



Introducing Project Explorer

Doug Hennig

Stonefield Software Inc.

Email: dhennig@stonefield.com

Corporate Web sites: www.stonefieldquery.com

www.stonefieldsoftware.com

Personal Web site : www.DougHennig.com

Blog: DougHennig.BlogSpot.com

Twitter: [DougHennig](https://twitter.com/DougHennig)

The Project Manager is one of the oldest tools built into VFP, and it has showed its age for a long time. For example, it doesn't provide integration with modern distributed version control systems (DVCS) such as Mercurial and Git, it doesn't have a way to filter or organize the list of items, and it can only work with one project at a time.

Project Explorer is a VFPX project that replaces the Project Manager with a modern interface and modern capabilities. It has most of the features of the Project Manager but adds integration with DVCS (including built-in support for FoxBin2PRG and optional auto-commit after changes), support for multiple projects within a "solution," allows you to organize your items by keyword or other criteria, and has support for easy "auto-registering" addins that can customize the appearance and behavior of the tool.

This document introduces Project Explorer and shows how it can make you more productive than working with the Project Manager. It starts by going through the interface and functionality of Project Explorer, then looks at its internals to see how it's designed, and finally shows how to write addins that extend the functionality or customize the user interface.

Introduction

I've wanted a replacement for the VFP Project Manager for a long time. There are many shortcomings to the Project Manager. Here are just a few:

- It doesn't have a way to filter or organize the list of items. Some of my projects are quite large. For example, SFQuery.pjx, the main project for Stonefield Query, has 1,335 items in it, most of which are classes. It takes a lot of scrolling to find the specific item I'm looking for. What would be nice would be a way to just display the five items I worked on today, or just see code specific to this project (that is, exclude framework classes, which I rarely look at), or just see items that for one reason or another seem to change frequently.
- It can only work with one project at a time. Stonefield Query consists of more than ten separate projects. It would be nice to be able to build them all with one mouse click rather than having to open a project, click Build, click OK, click Save, close the project, and repeat for the next project (in actuality, I have a BuildProjects.prg that programmatically builds all of the projects, but you get the point). Visual Studio has the concept of a solution, which consists of one or more projects that you can build one at a time or all at once.
- VFP was written back when SourceSafe and Vault were the big names in version control. Today, most developers use distributed version control systems (DVCS) such as Mercurial and Git. Unfortunately, VFP's source code control doesn't support DVCS so you end up either managing source code outside VFP (using the command line, Tortoise, or some other tool) or using tools such as Lutz Scheffler's Bin 2 Text Extension (<https://github.com/lscheffler/bin2text>) or Mike Potjer's VFP Git Utils (0 <https://github.com/mikepotjer/vfp-git-utils>) to provide integration inside VFP.

After thinking about this for many years, I decided that 2017 was the year when I finally did something about this. Project Explorer is the result.

Project Explorer sort of replaces the VFP Project Manager. I say "sort of" because behind the scenes, PJX files are still used and are still opened in the Project Manager, except the Project Manager window isn't visible; instead, you work in the Project Explorer window (**Figure 2**). Project Explorer overcomes the shortcomings I listed above and others as well, and provides a more modern user interface.

*** SEE *** IN REST OF DOC

Installing Project Explorer

Project Explorer is a VFPX project; its repository is located at <https://github.com/DougHennig/ProjectExplorer>. To install Project Explorer, do one of the following:

- If you use Git, create a folder on your system where you want Project Explorer to go, right-click that folder, and choose Git Clone. Specify <https://github.com/DougHennig/ProjectExplorer> as the URL to clone from.

- To download as a ZIP file, navigate your browser to <https://github.com/DougHennig/ProjectExplorer>, click the *Clone or download* button, and choose Download ZIP. Unzip the downloaded file in any folder you wish.
- You'll soon be able to install it using Thor Updater.

A tour of Project Explorer

There are a couple of ways you can open a project in Project Explorer:

- Run ProjectExplorer.app. It displays a dialog prompting you to select a solution (more about that in a moment) or a PJX file.
- Pass ProjectExplorer.app a path to a PJX file, a solution file, or a project object (that is, something like `_VFP.ActiveProject`).

You can, of course, launch Project Explorer from a Thor menu item or hotkey. This is discussed later in this document.

Another interesting way to launch Project Explorer is with a project hook that automatically runs ProjectExplorer.app when a project using that project hook is opened with MODIFY PROJECT. ProjectExplorerProjectHook in ProjectExplorerProjectHook.vcx shows an example of that. It simply has this code in the Activate event:

```
if vartype(_vfp.ActiveProject) = 'O' and _vfp.ActiveProject.Visible
    _vfp.ActiveProject.Visible = .F.
    lcPath = addbs(justpath(This.ClassLibrary))
    do (lcPath + 'ProjectExplorer.app') with _vfp.ActiveProject
endif vartype(_vfp.ActiveProject) = 'O' ...
```

You'll notice the first difference about Project Explorer when you run it without a parameter: it prompts you to open a solution or optionally a project (**Figure 1**). A solution is a set of projects that are managed together. For example, Stonefield Query consists of more than ten separate projects, each of which creates an EXE. By putting them into a solution, I can build all of the EXEs at once or just one if I want.

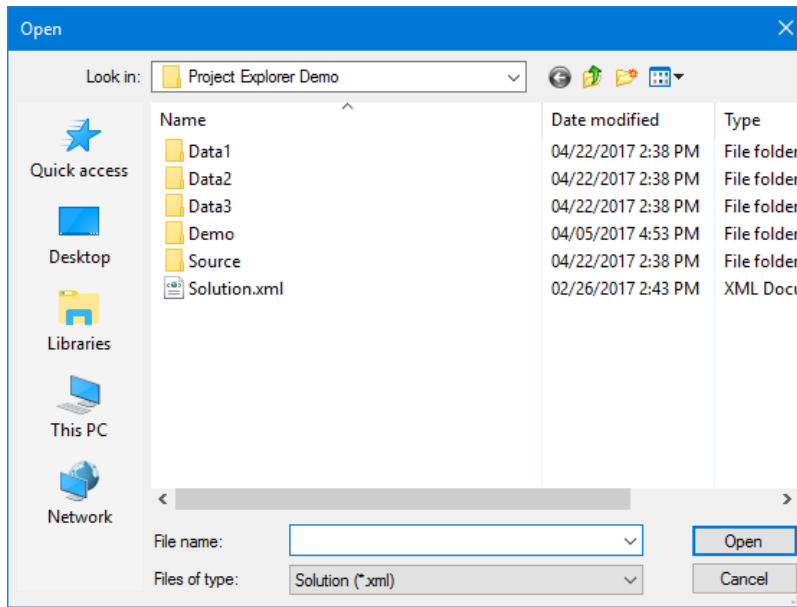


Figure 1. You are prompted to open a solution or a project.

Project Explorer works with PJX files just like the Project Manager does. It also creates a solution file named `Solution.xml` in the project folder. The solution file simply contains the list of projects and a few settings. We'll look at the structure of the solution file later.

The first time you open a project with Project Explorer, it takes a moment or two as it builds the meta data for the items in the project. Subsequent startups are much quicker.

The next difference you'll notice is the user interface (**Figure 2**). Project Explorer is similar to the Project Manager in that it displays a TreeView list of items in the project but there are several other differences that are immediately apparent:

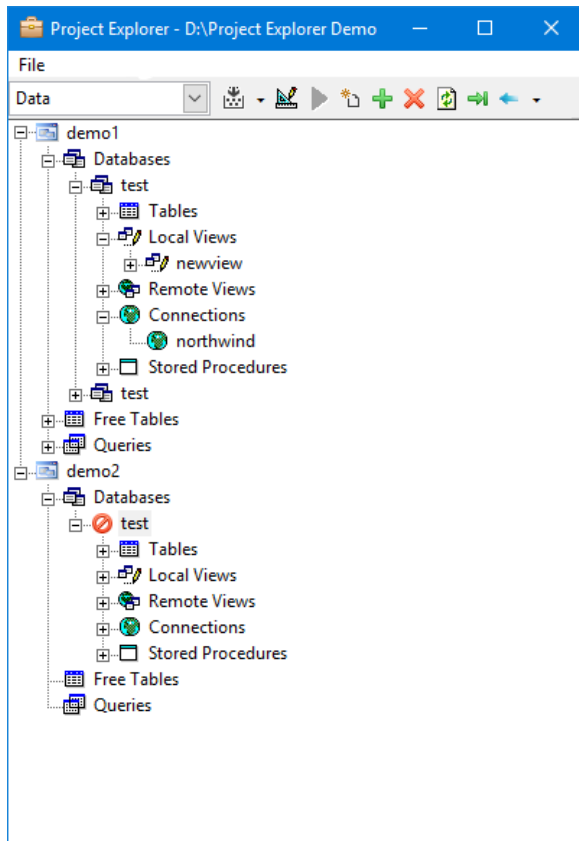





Figure 2. The user interface for Project Explorer is more modern than the Project Manager.

- The interface is more modern, with colorful picture buttons instead of text buttons.
- It's also more compact: the buttons are at the top in a toolbar rather than at the side, there are no tabs, and the information area at the bottom is gone. As a result, for a similarly sized window, you can see a lot more items in Project Explorer.
- Speaking of tabs, the equivalent mechanism to select a different set of files is with a combo box. Not only does it take up less space, it's also data-driven so it can organize items into more categories than just Data, Documents, Classes, Code, and Other. We'll discuss this in more detail later. A minor downside is that it takes two mouse clicks (the down arrow and then the desired choice from the combo box) rather than the one it takes with tabs.
- The associated menu is in the window rather than in the VFP system menu so the window is self-contained.
- Because the window has the Desktop property set to .T., the window can be dragged outside the VFP window, such as onto another monitor, giving you more workspace in the VFP window. This can be turned off in the Options dialog if you want the window to live inside the VFP window.
- The solution open in Figure 2 consists of two projects: Demo1.pjx and Demo2.pjx in D:\Project Explorer Demo. The folder is shown in the window title bar (something

you can't see in the Project Manager) and the contents of both projects are shown in the TreeView.

- The TreeView uses the same icons as the Project Manager does for file types with a couple of exceptions. First, because the TreeView control can only display one image for an item, files excluded from the project are displayed as  rather than as the file type icon followed by a second “no” icon. Second, as you can see in **Figure 3**, when a solution is under version control, the icon for a file displays its status rather than type. The two projects and the first database named Test appear as , meaning they are “clean” or unmodified but the second Test database (although they're in different folders, I named them both Test to show Project Explorer would work with duplicate names) and Table1 table appear as  because they have been modified and not yet committed.

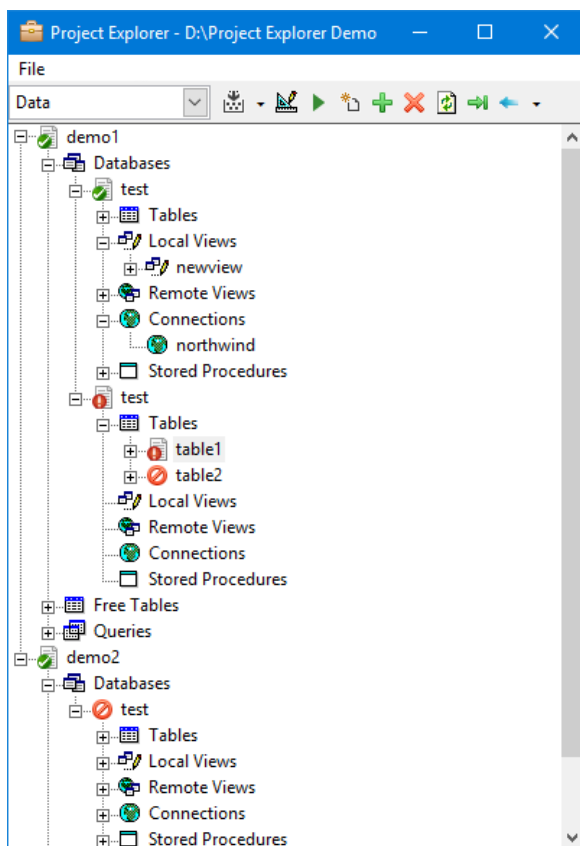




Figure 3. The icon for a file shows its version control status.

Item properties

What if I want to see the path or description of a file? To do that, click the Expand button in the toolbar () to show the full window, shown in **Figure 4** (the Expand button icon changes to ; clicking it again collapses the window). Project Explorer displays a lot more properties of the selected item than just path and description:

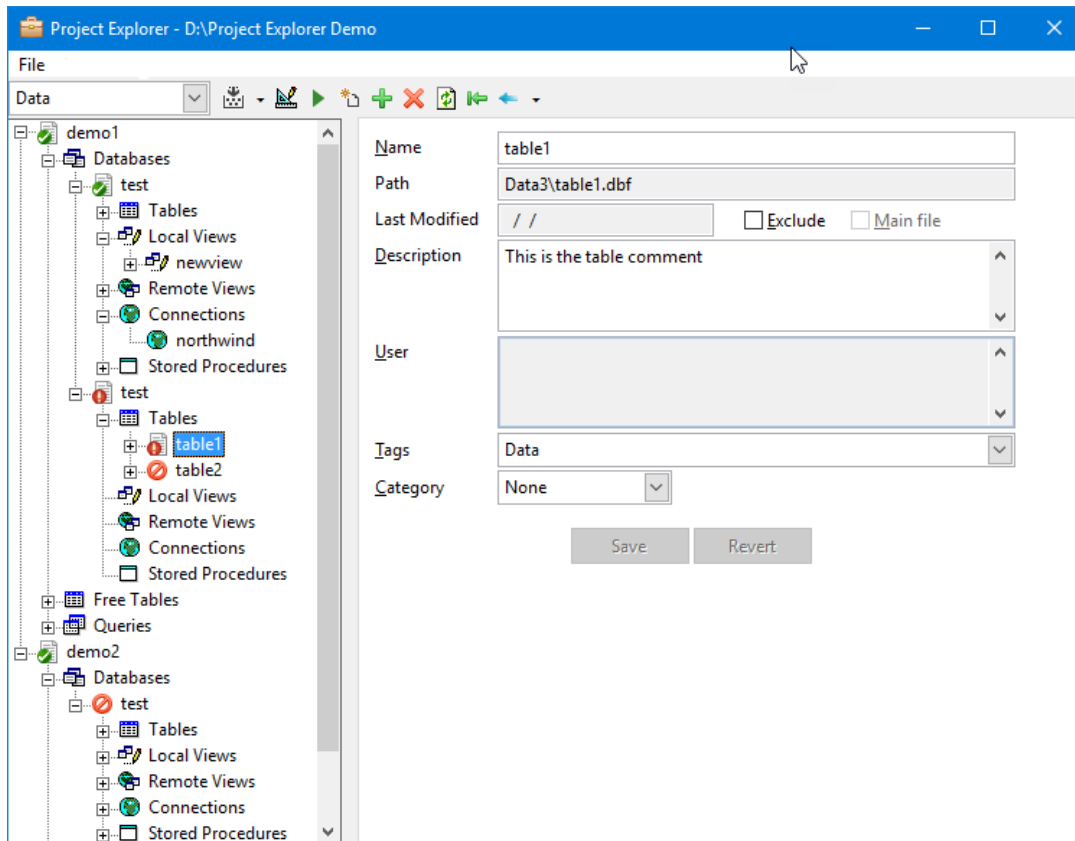


Figure 4. The full Project Explorer window shows all the properties for the selected item.

- **Name:** the name of the item. To rename an item, enter a new name. Project Explorer won't let you enter an invalid name and gives a warning if it's a duplicate name when you try to save. Not all item types can be renamed: to rename a field or index, use the Table or View Designer; to rename a stored procedure, edit the stored procedures; and a database can't be renamed because the backlink to the DBC in the DBF header of every table in the database has to be updated. (You can't rename a database in the Project Manager either; it looks like you can but when you click OK, you get a message that the database is open.)
- **Path:** the path for the item. In the case of a field or index, it's the path for the table or the name of the view. For views, connections, and stored procedures, it's the path for the database. For a class, it's the path for the VCX. For all other types, it's the path for the file. This is always read-only.
- **Last Modified:** the date the item was last modified. This isn't maintained for all item types so in those cases it's blank. This is always read-only.
- **Exclude:** turn this on to exclude the item from the project or off to include it; you can also right-click the item and choose Exclude from the shortcut menu to toggle the setting. Only items that are files (for example, not classes) can be included.

- **Main file:** turn this on to mark the selected item as the main file for the project; you can also right-click the item and choose Set Main from the shortcut menu to toggle the setting. This is only available for forms and programs.
- **Class:** the class the form or class is based on. This is always read-only and only displays for forms and classes.
- **Library:** the class library for the class the form or class is based on. This is always read-only and only displays for forms and classes.
- **Base class:** the base class for the class. This is always read-only and only displays for classes.
- **Include file:** the include file for the class. This is always read-only and only displays for classes.
- **OLEPublic:** turned on if the class is a COM server. This is always read-only and only displays for classes.
- **Icon:** the custom icon for the class displayed in the Project Manager, the Class Browser, and Project Explorer. This is always read-only and only displays for classes.
- **Toolbar icon:** the custom icon for the class displayed in the Form Control Toolbar. This is always read-only and only displays for classes.
- **Description:** the description for the item. For files, this is stored in the PJX file. For classes, it's stored in the VCX file. For tables, views, connections, and fields and indexes in views or tables belonging to a database, it's stored in the DBC file. (Note that there's no usual way to get or save the description for an index; although it can be stored in a DBC, DBGETPROP and DBSETPROP don't support it, so Project Explorer manually reads from and writes to the PROPERTY memo of the index's record in the DBC.) This is disabled for field and indexes in free tables and stored procedures since there's nowhere to store it.
- **User:** user-defined information for the item. For files, this is stored in the PJX file. For classes, it's stored in the VCX file. This is disabled for all other item types.
- **Tags:** keywords that apply to the item. Tags are discussed in more detail later.
- **Category:** a color coding for the item. Category is discussed in more detail later.

Click the Save button to save any changes or Revert to restore the previous property values.

Project properties

When you select a project in Project Explorer, it displays the properties of the project at the right (**Figure 5**). This replaces several modal dialogs in the Project Manager. The properties on the Project Settings page are those from the Project page of the Project Manager's Project Information dialog; see the *Project Tab, Project Information Dialog Box* topic in the VFP help for details. Some comments about these properties:

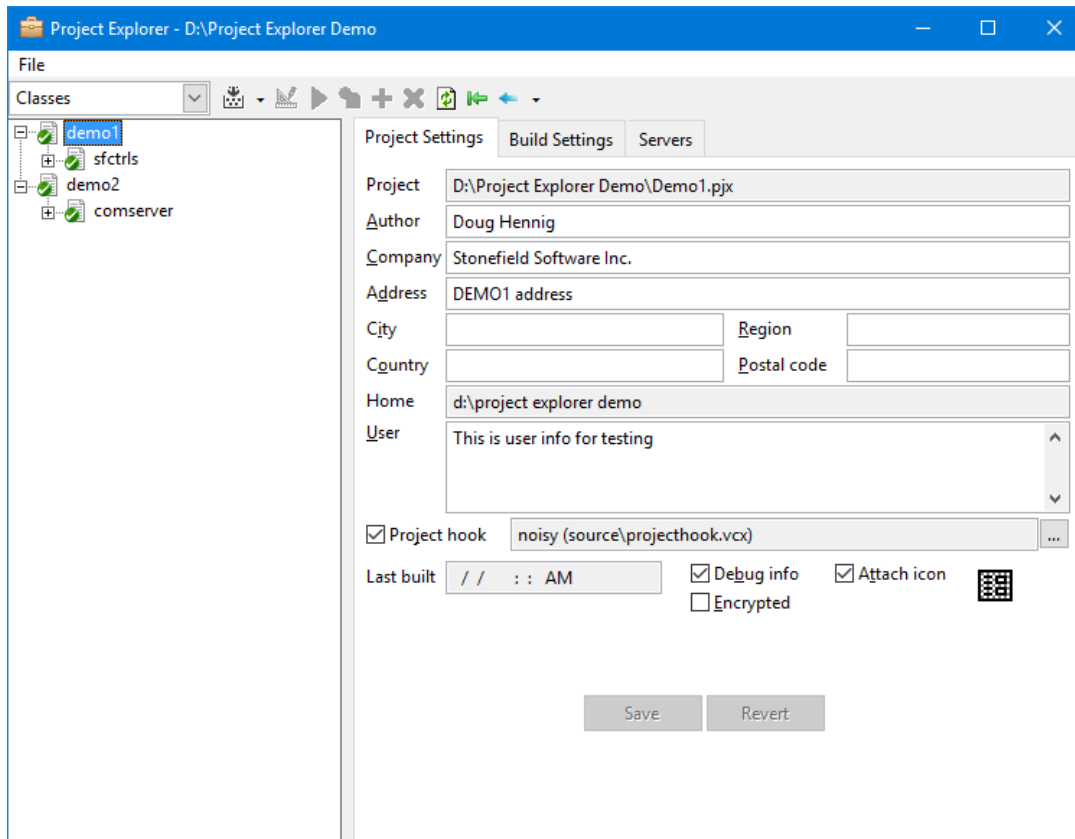


Figure 5. Project Explorer replaces several modal dialogs with the properties of the selected project.

- **Project:** the name and path of the project. This is always read-only.
- **Home:** the home folder for the project. This is always read-only.
- **User:** user-defined information for the project itself. This is stored in the “H” record of the PJX file.
- **Project hook:** to add a project hook to a project, turn this on, click the button with the ellipsis (...) beside the project hook class name, and select a VCX and a class. You’ll get a warning if the class you select isn’t a subclass of ProjectHook. To remove the project hook for the project, turn this off. Note that project hook changes take effect immediately, unlike with the Project Manager where you have to close and reopen the project.
- **Last built:** the date and time the project was last built. This is always read-only.
- **Attach icon:** to specify an icon for the compiled file, turn this on, click the image, and select the desired icon file.

The properties on the Build Settings page (**Figure 6**) are those from the Version dialog displayed when you click the Version button in the Build Options dialog, which is displayed when you click the Build button in the Project Manager. See the *EXE Version Dialog Box* topic in the VFP help for details.

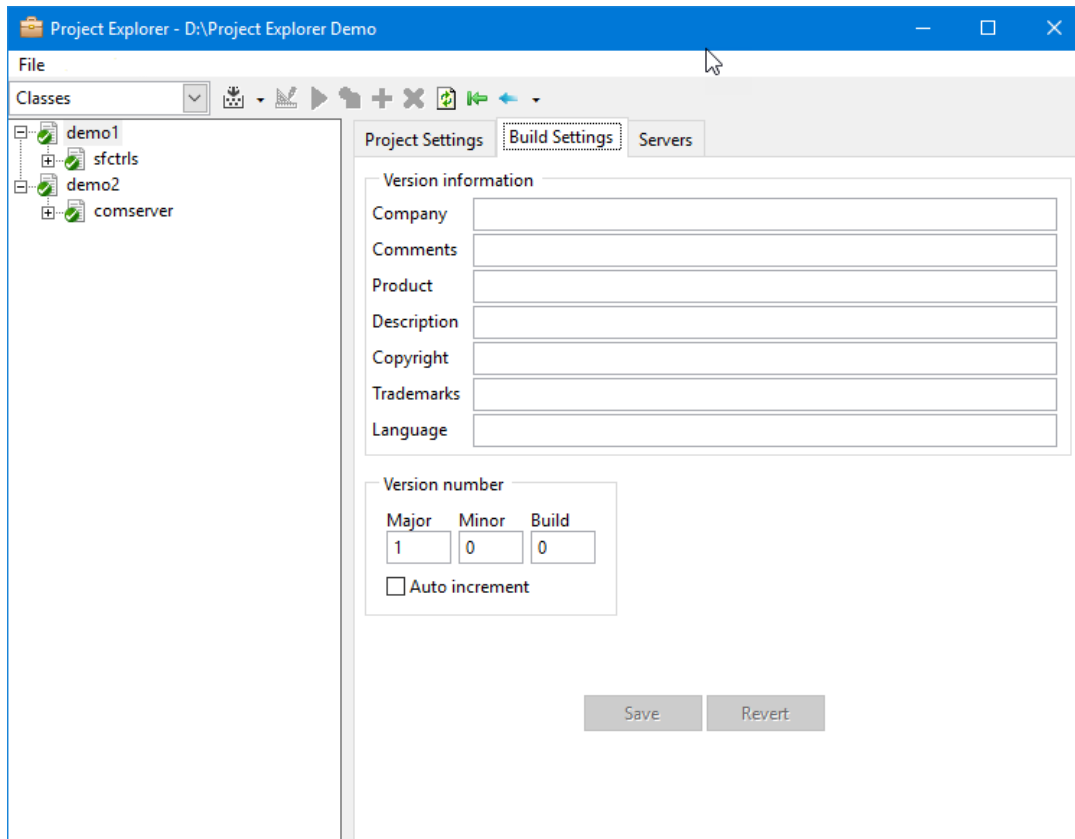


Figure 6. You have to display two modal dialogs to get at these settings in the Project Manager.

The properties on the Servers page (**Figure 7**) are those from the Servers page of the Project Manager's Project Information dialog; see the *Servers Tab, Project Information Dialog Box* topic in the VFP help for details.

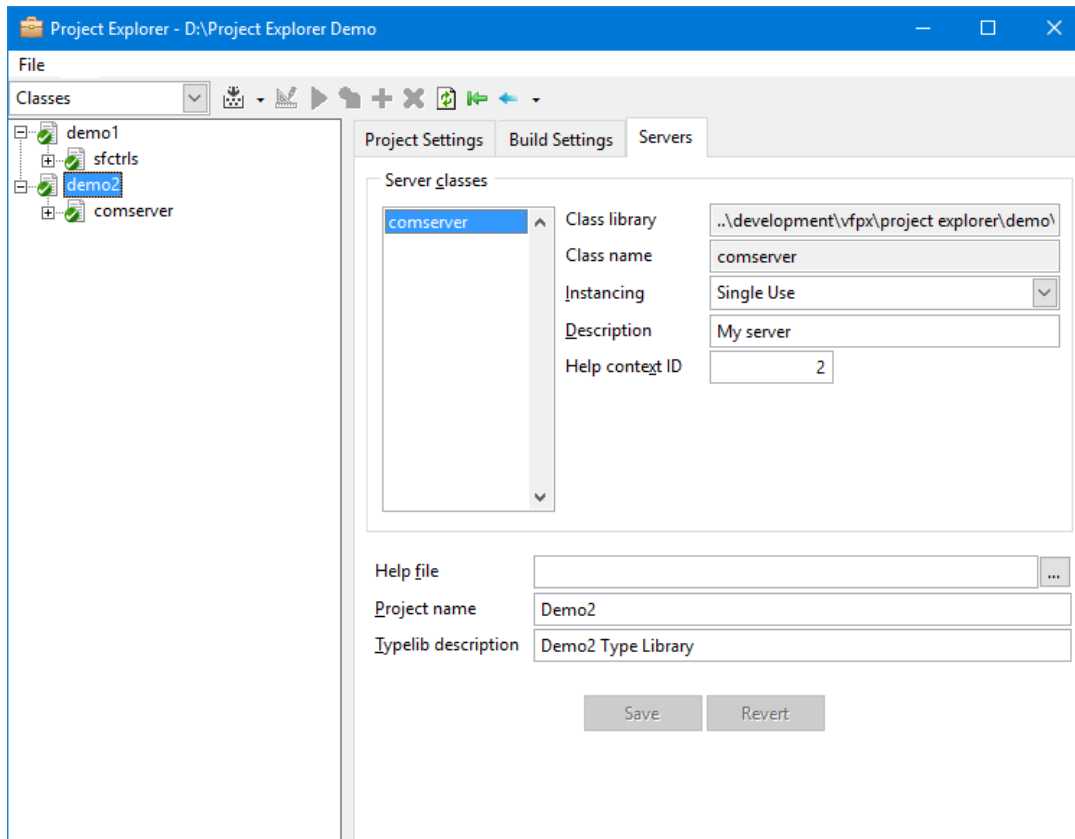


Figure 7. The Servers page has the settings from the Servers tab of the Project Information dialog.

Item organization

The Project Manager provides one way to organize item: by type. The tabs at the top allow you to see either all items (the All tab) or just those of the specified type.

One thing I have always wanted is to way to organize items in other ways. For example, I use an in-house framework. Sometimes I want to see framework items but often I just want to see the items specific to the project. Occasionally, I'd like to see only those items I'm currently working on, such as the various classes and programs in a certain module. In other words, I need user-defined categorization.

Project Explorer provides two ways to categorize items: by tags and by category.

Tags

Tags are keywords that apply to an item. The default tag for an item is the Project Manager tab it appears in: Data for free tables, queries, databases, tables, fields, indexes, views, connections, and stored procedures; Documents for forms, reports, and labels; Classes for class libraries and classes; Code for programs, applications, and API libraries; and Other for menus, text files, and other types of files including images. However, when you click the down arrow for the Tag control, you see a list of the available tags with checkboxes so you can select all the tags that apply to the item.

To define your own tags, choose Tag Editor from the File menu (**Figure 8**).

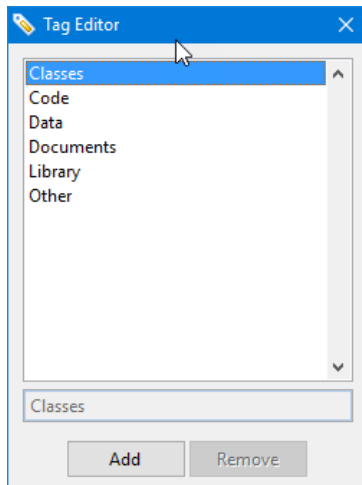


Figure 8. The Tag Editor allows you to define your own tags.

The “built-in” tags (the ones representing Project Manager tabs) can’t be edited or removed, but you can click the Add button and enter a tag name to create a new one or select one and change its name or remove it. Tags are stored in a table named ProjectExplorerTags.dbf in the Project Explorer folder.

The Tag combobox in the toolbar at the top of the window tells Project Explorer to display only those items containing that tag. Without any custom tags, it acts like the tabs in the Project Explorer. However, if you create a custom tag and, for example, use that tag for a few classes and programs, when you select that tag from the combobox, only those classes and programs appear in the TreeView. So, to use my earlier examples, I could tag all my project-specific items as “Project”, and then choose “Project” from the combobox when I only want to see those items.

Category

Category is a color coding for an item. I took the inspiration for this feature from Microsoft Outlook, which allows you to assign a category to an item to color-code it.

The Category combobox for an item allows you to select a single category for the item. Selecting one changes the color of the item’s node in the TreeView to the color for the category. There are eight categories available. Initially, the categories are named for their colors (none [black], red, blue, green, yellow, orange, purple, and indigo) but you can use the Category Editor (**Figure 9**), available from the File menu, to change the names and even the colors.

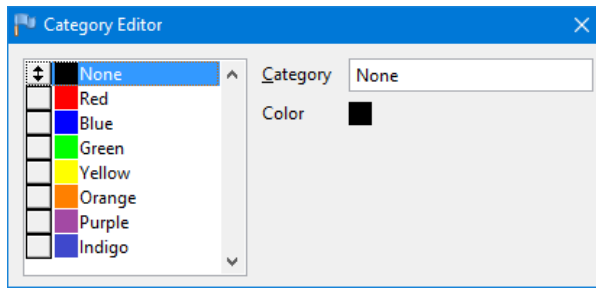


Figure 9. The Category Editor allows you to customize the labels and colors of categories.

To edit a category, select it in the list and enter a new name or click the colored square to select a new color. Categories are stored in a table named `ProjectExplorerCategories.dbf` in the Project Explorer folder.

Using tags and categories

What's the difference between a tag and a category? You can use these categorization features any way you wish, but I think of tag is what the item is, which is usually permanent, and category as a short-term description. For example, you may use a schema where red means Unfinished, blue means Untested, and black means Completed. Once an item is finished, you change its category from Unfinished to Untested, and once testing is done, from Untested to Completed. You can quickly tell at a glance which items belong in which category by the node color.

The tags and category for each item in a project are stored in the meta data table for the project, named *ProjectName_MetaData.dbf*, where *ProjectName* is the name of the project, in the solution's folder.

While the tag combobox allows you to display only those items having a certain tag, to display only items in a certain category, you have to use filtering.

Sorting and filtering

The Sort and Filter function in the File menu (**Figure 10**) gives you control over which items appear in the TreeView and in what order.

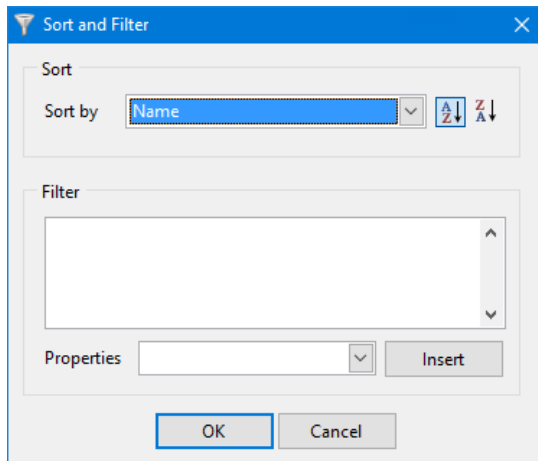


Figure 10. The Sort and Filter dialog gives you control over which items appear in the TreeView and in what order.


The choices for sorting are by Name (the item name, case-insensitive), Last Modified Date, and Category. You can choose between ascending and descending order.

Enter a VFP expression into the Filter editbox to filter the items in the TreeView. The Properties combobox assists with entering a filter based on item properties: select a property from the list and click the Insert button to add it to the editbox. For example, to display only those items modified in the past 30 days, use the following filter expression:

```
ttod(Item.LastModified) >= date() - 30
```

To clear the filter, choose Sort and Filter from the File menu again, clear the editbox, and click OK.

Building

Click the  button in the toolbar to display the Build Options dialog (**Figure 11**) to build the selected project (this button is disabled if there's no main file for the project). Alternatively, click the arrow beside the button to display additional build choices:

- **Build Project:** builds the selected project without displaying the dialog and using the previous build settings.
- **Build Solution:** builds all projects in the solution without displaying the dialog. You can also right-click the TreeView and choose Build Solution from the shortcut menu.
- **Rebuild Project:** builds the project with the RECOMPILE option without displaying the dialog
- **Rebuild Solution:** builds all projects in the solution with the RECOMPILE option without displaying the dialog.

The first time a project is built, the Build Options dialog appears even if you chose a build function from the menu.

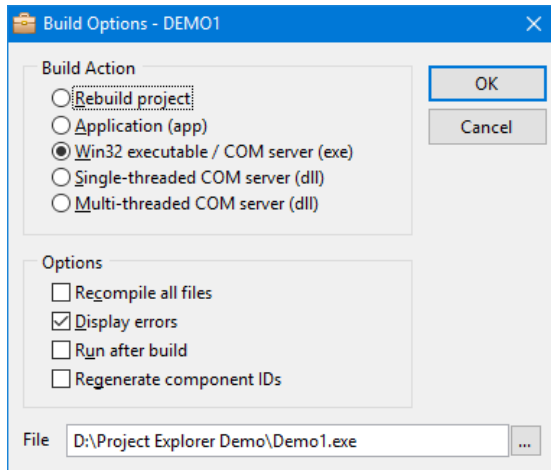



Figure 11. The Build Options dialog allows you to specify build settings.

The options in the Build Options dialog are the same as they are in the Project Manager's Build Options dialog except there's no Version button since the version settings are available in the Build Settings page of the project properties and there's a control for the output file name rather than another dialog for the file name. Project Explorer remembers these settings on a project-by-project basis so after you've built a project or solution the first time, you can just choose Build Project or Build Solution from the build button menu to use the same settings without displaying the dialog.


Managing items

The toolbar has functions to manage project items.

Modifying

Click the Modify button () to display the editor for the selected item. Alternatively, you can double-click the item if the *Project double-click action* setting in the Options dialog (discussed later) is set to *Modify selected file*. This is available for all item types except API libraries. There's special handling for images, which display the registered application for the type of image file, for applications, which open Project Explorer for the project that builds to the application if that project exists, and for class libraries, which open that VCX in the Class Browser.

Running

Click the Run button () to "run" the item. Alternatively, you can double-click the item if the *Project double-click action* setting in the Options dialog (discussed later) is set to *Run selected file*. In the case of programs, forms, menus, and applications, the item is run. For reports and labels, the item is previewed. For tables, views, and queries (including fields and indexes in tables and views), the item is opened and displayed in a BROWSE window. For other item types, the Run button is disabled.

For obvious reasons, any code run from Project Explorer should not do CLOSE ALL or CLEAR ALL.

Creating

Click the New button (🌟) to create a new item of the same type as the selected item. This button is disabled for application, field, index, API library, and “other” items and enabled for all other types. When you click this item for classes, the New Class dialog (**Figure 12**) appears, which has the same functionality that dialog has in the Project Manager. For connections and views, you’re prompted for a name. For other types, a file dialog appears so you can specify the name and path of the new item. Notice that this is different than the Project Manager, which asks you for the name when you save the item.

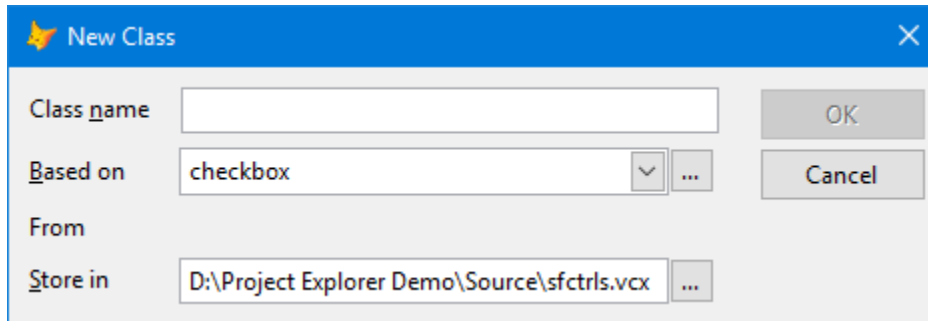


Figure 12. The New Class dialog displays when creating a new class.

As with modifying, an event fires when the editor is closed. We’ll discuss that later when we discuss addins.

Adding

Click the Add button (+) to add an item of the same type as the selected item to the project. This button is only enabled for item types that are files. When you add a table to a database, it’s added both to the database and to the project.

Removing

Click the Remove button (✖) to remove the selected item. This button is only enabled for item types that are files as well as classes, connections, views, and tables in a database. You’re prompted whether you want to remove the item from the project (in the case of a class, from the VCX, or in the case of an item in a database, from the database) or if you want it both removed and deleted.

Managing solutions




The File menu has several functions for managing solutions. *Add Project to Solution* adds a project to the solution and *Remove Project from Solution* removes the selected project from the solution. Note that you can’t remove the last project from a solution. *Cleanup Solution* cleans up each project (basically packing the PJX) and packs the meta data tables.

Version control

Project Explorer is integrated with modern distributed version control systems (DVCS) such as Mercurial and Git. This integration takes several forms. First, as mentioned earlier,

the icon for a file displayed the TreeView indicates its version control status. See **Table 1** for a list of the icons and their meanings.

Table 1. Version control status icons.

Icon	Meaning
	Unversioned
	Ignored
	Added
	Removed
	Modified
	Clean (unmodified)

Second, if a solution is under version control, the shortcut menu for items in the TreeView has several functions related to version control:

- **Add File to Version Control:** adds the file to version control. This function is disabled if the item is already in version control or isn't a file.
- **Remove File from Version Control:** removes the file from version control. This function is disabled if the item isn't in version control or isn't a file.
- **Commit File:** commits the file. This function is disabled if the item isn't in version control, isn't added, removed, or modified, or isn't a file.
- **Commit All:** commits all changes, not only to items in the project but to the project itself.
- **Revert:** reverts the file. This function is disabled if the item isn't in version control, is unversioned or ignored, or isn't a file.
- **Revision History:** displays the revision history for the item. In the case of a VFP binary file, it's actually the revision history of the text equivalent that's displayed. This function is disabled if the item isn't in version control or isn't a file.
- **Visual Diff:** displays the visual diff dialog for the item. In the case of a VFP binary file, it's actually the diff for the text equivalent. This function is disabled if the item isn't in version control, isn't added, removed, or modified, or isn't a file.
- **Repository Browser:** display the repository browser.

To put a solution under version control, select the *Version Control Properties* function in the File menu to display the dialog shown in **Figure 13**. This dialog has the following settings:

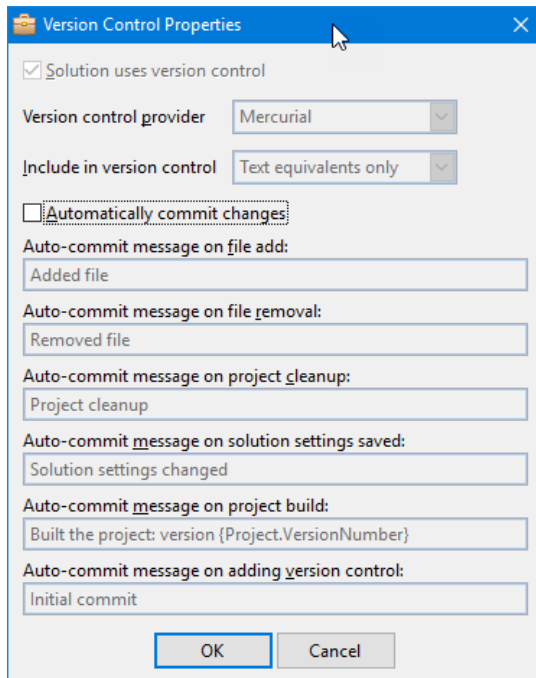


Figure 13. The Version Control Properties dialog manages version control settings for the solution.

- **Solution uses version control:** turn this on to put the solution under version control. Note that once you’ve turned it on, you can’t turn it off.
- **Version control provider:** the version control provider to use: Mercurial or Git (others can be supported as discussed later). Once you’ve selected a provider, you can’t change it.
- **Include in version control:** specifies what to include in version control. This is discussed in detail later. Once you’ve made a selection, you can’t change it.
- **Automatically commit changes:** turn this on to automatically commit changes to files (discussed in more detail later). You can turn this on or off as desired.
- **Auto-commit messages:** these are the commit messages to use when some types of changes are automatically committed: when a file is added or removed, the project is cleaned up, the solution settings are saved, a project is built, or version control is added to a solution. The build message supports text merge using Project as a reference to the project being built and “{” and “}” as the text merge delimiters. For example, the default message of “Built the project: version {Project.VersionNumber}” includes the version number of the build.

There are also some settings related to version control in the Options dialog, discussed later.

If *Automatically commit changes* is turned on, when you add, remove, modify, or make changes to the properties of items, the changes are automatically committed. In some cases, you’re prompted for a description of the changes and in others, the message you specified in the Version Control Properties dialog is used. Some changes affect only the file for the

item, such as when you modify the item or in the case of a class, create it or change its Description or User. Some changes affect another file, such as adding or removing fields or indexes in a table that belongs to a database, in which case both the table and the database container are affected. Other changes affect the project, such as when files are added, created, or removed or you change the Description or User property of a file-based item. Project Explorer knows which files are affected with each type of change and automatically commits those files.

You have three choices about what's included in version control, as specified by the *Include in version control* setting:

- **Binary files only:** if this is selected, Project Explorer includes all files in the project, including VFP binary files such as VCX, VCT, SCX, and SCT files in version control. The project files themselves (PJX and PJT) are also included.
- **Text equivalents only:** if this is selected, VFP binary files are not included in version control but all other file types and the text equivalents of the binary files are. Project Explorer uses FoxBin2PRG, a VFPX (<http://vfpx.org>) project available through Thor Updates or for download from VFPX, to convert VFP binary files to their text equivalents and vice versa. This setting means, for example, that MyClasses.vc2 (the text equivalent of MyClasses.vcx) is included in version control but MyClasses.vcx and MyClasses.vct aren't.
- **Both:** all files in the project plus the text equivalents of VFP binary files are included in version control.

If *Include in version control* is set to anything but *Binary files only*, here's what happens when you do various things to a class (as an example) in Project Explorer:

- When you add a class library to a project, Project Explorer tells FoxBin2PRG to create the text equivalent of the class library (VC2) and the project (PJ2). If auto-commit is turned on, the text equivalents are committed, as are the VCX, VCT, PJX, and PJT files if *Both* is used.
- When you modify the class and save it, Project Explorer tells FoxBin2PRG to create the text equivalent of the class library the class belongs to. If auto-commit is turned on, the text equivalent is committed, as are the VCX and VCT files if *Both* is used.
- When you revert changes to the class library and *Text equivalents only* is used, the VC2 file is reverted and FoxBin2PRG is told to regenerate the VCX and VCT files from it. If *Both* is used, the VC2, VCX, and VCT files are reverted.
- When you remove the class, Project Explorer removes the VCX and VCT files from the project (and optionally deletes them), deletes the VC2 file, and removes the VCX, VCT, and VC2 files from version control.

Other functionality

Project Explorer options

The *Options* function in the File menu displays the dialog shown in **Figure 14**. It has the following options:

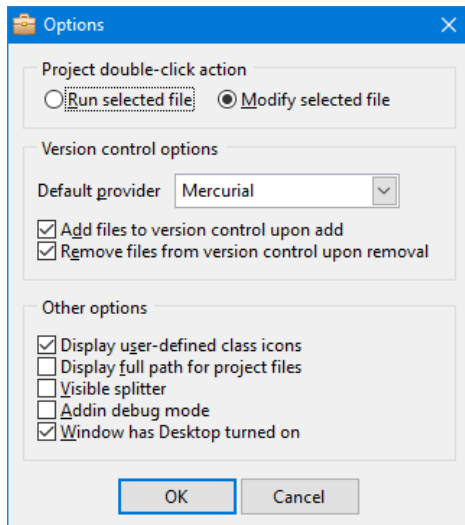


Figure 14. The Options dialog has Project Explorer settings.

- **Project double-click action:** determines whether double-clicking an item runs or modifies it. This is the same setting as in the Projects page of the VFP Options dialog so changing it in one place changes it in the other as well.
- **Default provider:** the default provider for the Version Control Properties dialog.
- **Add files to version control upon add:** turn this on to automatically add a file to version control when it's created or added to a project. This is the same setting as in the Projects page of the VFP Options dialog so changing it in one place changes it in the other as well.
- **Remove files from version control upon removal:** turn this on to automatically remove a file to version control when it's removed from a project. This is the same setting as in the Projects page of the VFP Options dialog so changing it in one place changes it in the other as well.
- **Display user-defined class icons:** turn this on to display the custom icon for a class (the one specified in the *Container icon* setting in the Class Info dialog and the *Icon* property in Project Explorer) in the TreeView or off to display an icon representing the class' base class. This is the same setting as in the Projects page of the VFP Options dialog so changing it in one place changes it in the other as well.
- **Display full path for project files:** turn this on to display the full path for files or off to display the path relative to the location of the project.


- **Visible splitter:** turn this on to display the splitter between the TreeView and the properties pane as a grey bar with four dots or off to not display the splitter (it's still there; it just doesn't have a visible appearance).
- **Addin debug mode:** turn this on to enable debug mode for addins: Project Explorer shows information as addins are loaded and executed.
- **Window has Desktop turned on:** turn this on to allow the Project Explorer window to be moved outside the VFP desktop. In that case, the menu is inside the window and the window is always on top. Turn this off if you want the window inside the VFP desktop, which means other windows can be on top of it and that the Project Explorer menu appears in the VFP system menu.

Drag and drop


Project Explorer supports the same drag and drop functionality that the Project Manager does:

- You can drag a file from File Explorer to the TreeView to add it to the project.
- You can drag a class to a Form Designer or Class Designer window to add an instance of the class to the form or class (if permitted).
- You can drag a class to another class library to copy it.
- The Project Manager doesn't support this, and the "no" icon makes it look like it won't work, but you can also drag a class to _SCREEN to add an instance of the class there.

Refreshing the TreeView

If you make changes to files outside of Project Explorer (for example, modifying a program using Notepad or when the project isn't open in Project Explorer), the TreeView may not display the current status of files. Click the Refresh button () in the toolbar to reload the TreeView.

Moving to a previous item

Click the Back button () in the toolbar to move back to the previously selected item. You can also click the down arrow beside the Back button to display a list of previously selected items and select one to go back to that item. This is a quick way to jump back and forth between frequently edited items.

Project hook support

Because Project Explorer opens the project behind the scene (that is, the Project Manager is actually open but invisible), the project appears in the _VFP.Projects collection and is contained in _VFP.ActiveProject. In addition, project hooks are supported for the same actions in Project Explorer that would trigger them in the Project Manager. Here are some comments about that:

- Events such as QueryRemoveFile and QueryRunFile fire for files and classes but not items in a database (tables, views, connections, and stored procedures), although there's some inconsistency because QueryNewFile does fire for new items in a database.
- In the Project Manager, QueryRemoveFile fires before the prompt about removing the file appears. In Project Explorer, QueryRemoveFile fires after the prompt.
- Unlike the Project Manager, an event fires when an editor is closed after creating or modifying an item, albeit not a project hook event since there isn't one. We'll discuss that later when we discuss addins.
- Activate and Deactivate don't fire because the Project Manager isn't visible.
- While project hooks are supported, addins are more powerful, easier to create and install, and there are more of them. See the Addins section for more details.

Unimplemented Project Manager features

The following are features of the Project Manager that weren't implemented in Project Explorer:

- Tearing off tabs into their own windows.
- Shrinking down the window to just show tabs.
- Docking as a toolbar.
- The Files page of the Project Information dialog.
- Launching the Application Builder or Web Services Publisher from the Builder item in the shortcut menu.
- Selecting the code page for an item from the shortcut menu.
- Choosing Save As from the File menu to save to a new project.

Inside Project Explorer

Let's look under the hood and see how Project Explorer was built.

Classes

All of the VCXs and most of the classes used in Project Explorer have "ProjectExplorer" or "Project" as a prefix. The reason for that is to avoid conflict with classes and class libraries that exist in your projects. For example, if Project Explorer had a class named BaseTextBox and you have a class named BaseTextBox, Project Explorer would be confused when you try to edit yours from within Project Explorer.

The first classes we'll look at are those in ProjectExplorerItems.vcx. ProjectItem is the base class for an item in a project. It has quite a few properties (see the About method for a list of them and their descriptions), some of which match properties from a VFP File object (such as Description, Exclude, and LastModified), others that describe what operations can

be performed on it (such as `CanEdit`, `CanRemove`, and `CanRun`), and still others that describe where it fits into a hierarchy (such as `HasChildren`, `HasParent`, and `ParentPath`). It also has a `Tags` property which contains a collection of tags for the item, a `ForeColor` property which is the category for the item (originally I called this attribute “color” rather than “category”), and a `VersionControlStatus` property which contains a single letter indicating the version control status of the item (such as “M” for modified).

`ProjectItem` has several methods, some of which perform an action on the item and are abstract in this class (such as `EditItem`, `RemoveItem`, and `RunItem`) and others which provide management functions (such as `GetTagString` and `SaveTagString`).

A subclass of `ProjectItem`, `ProjectItemFile`, is the parent class for those items that are files, such as programs and class libraries but not classes or views. Many of `ProjectFile`’s methods use the appropriate method of the VFP File object to perform the operation, such as `RunItem`, shown in **Listing 1**, which calls the `Run` method of the File object.

Listing 1. `ProjectItemFile.RunItem` calls the `Run` method of the VFP File object.

```
lparameters toProject
local loFile, ;
    llReturn, ;
    loException as Exception
This.cErrorMessage = ''
if This.CanRun
    try
        loFile = toProject.Files.Item(This.Path)
        llReturn = loFile.Run()
    catch to loException
        This.cErrorMessage = loException.Message
    endtry
endif This.CanRun
return llReturn
```

The remainder of the classes in `ProjectExplorerItems.vcx` are specific for each item type. For example, `ProjectItemConnection` is for connections in a database and `ProjectItemMenu` is for a menu. These items have the properties set appropriately for the item type. For example, `ProjectItemApplication`, which represents an application item, has `CanInclude` set to `.F.` because an application cannot be included in a project, `CanRun` set to `.T.` because an application can be run, `DefaultTags` set to “Code” because that’s the default tag associated with applications, and `TreeViewImage` set to “application” because that’s the key of an `ImageList` image used for the `TreeView` node for an application. It also has code in `CanEdit_Access` that returns `.T.` if a project can be found for the application and code in `EditItem` that uses Project Explorer to open that project.

The next set of classes we’ll look at are in `ProjectExplorerEngine.vcx`. `ProjectSettings` contains settings for a project, many of which (such as `Encrypted`, `Icon`, and `MainFile`) come directly from the VFP Project object. The properties associated with the Version dialog of the Build Options dialog, such as `Author`, `Company`, and `Address`, have to be parsed from the `DEVINFO` field in the `PJX` file. `ProjectSettings` also has an `oServers` property which

contains a collection of `ProjectExplorerServer` objects; these objects contain properties about the COM servers in the project, such as `Description`, `Instancing`, and `ProgID` (the properties available in the `Server` tab of the `Project Information` dialog).

`ProjectEngine` represents a project. It has an `oProjectItems` property that contains a collection of `ProjectItem` subclasses, one for each item in the project. It also has an `oProjectItem` property that contains a `ProjectItemFile` object for the PJX file itself (mostly used for its `VersionControlStatus` property), an `oProjectSettings` property that contains the `ProjectSettings` object for the project, and an `oProject` property that contains a reference to the VFP Project object, and a `cProject` property that contains the name and path for the PJX file. See the `About` method for a complete list of properties and their descriptions.

The main method in `ProjectEngine` is `GetFilesFromProject`, which adds a `ProjectItem` subclass for each file in the project to `oProjectItems`. It uses `AddFileToCollection` to do most of the work and calls `GetClasses`, `GetDatabaseItems`, and `GetTableItems` to add items for the classes, items in databases, and fields and indexes in free tables, respectively, to the collection. It also has some helper methods, such as `GetItemForFile`, which returns the `ProjectItem` subclass for the specified file, and `GetItemParent`, which returns the `ProjectItem` subclass that's the parent for the specified item (for example, the item for the VCX that the specified class belongs to).

`ProjectExplorerSolution` represents a solution. It has an `oProjects` property that contains a collection of `ProjectEngine` objects, one for each project in the solution, and a `cSolutionFile` property that contains the name and path to the solution file. It also has an `oVersionControl` property that contains a subclass of `VersionControlOperations` to provide version control services and several properties, such as `lAutoCommitChanges`, that specify the version control behavior. It has numerous methods for dealing with projects, including `OpenProjects`, `CloseProjects`, `CleanupSolution`, `AddProject`, and `RemoveProject`. See the `About` method for a complete list of members and their descriptions.

`ProjectOperations` is a small class that performs operations on a project. It has methods such as `AddItem`, `EditItem`, `RunItem`, and `RemoveItem` which do what their names imply.

`ProjectAddins` provides addin support. This is discussed in the `Addins` section.

`VersionControlOperations` is an abstract class that provides version control services, such as creating a repository, adding, removing, committing, and reverting files, and getting the status of files. Two of its subclasses, `MercurialOperations` and `GitOperations`, implement the actual behavior for Mercurial and Git support. Both of these use the command line API, calling `hg.exe` and `git.exe`, respectively, with the appropriate parameters for the operation. `VersionControlOperations` has a `cFoxBin2PRGLocation` property that contains the path to `FoxBin2PRG`, a VFPX project (<https://github.com/fdbozzo/foxbin2prg>) that converts VFP binary files to their text equivalents and vice versa. `VersionControlOperations'` `nIncludeInVersionControl` property determines what files are included in version control: binary files only, text equivalents only, or both.

The final set of classes we'll look at are in ProjectExplorerUI.vcx. This class library contains the UI classes for Project Explorer, such as the various dialogs and of course the Project Explorer form itself, ProjectExplorerForm (used if the *Window has Desktop turned on* setting is turned off) and ProjectExplorerFormDesktop (used if *Window has Desktop turned on* is turned on).

ProjectExplorerForm uses many of these classes. Its oSolution and oOperations properties contain instances of ProjectExplorerSolution and ProjectOperations, respectively. oItem contains the ProjectItem subclass for the currently selected item and oProject contains a reference to the ProjectEngine object that item belongs to. oProjectSettings contains a reference to the ProjectSettings object that belongs to the current project, just for data binding purposes.

*** DISCUSS TREEVIEW CLASSES

*** DISCUSS WINDOWS EVENT HANDLING

After Project Explorer opens, it adds a new member to _screen: oProjectExplorers, which is a collection of Project Explorer instances (you can open more than one at a time). You can access the Project Explorer form using the folder for the project or solution as the key (you can also use an index number). For example, if I open a solution from C:\My Projects\Customer A\Main Application, I can reference the open Project Explorer using:

```
_screen.oProjectExplorers(' C:\My Projects\Customer A\Main Application')
```

This is handy if you want to change something about the form (although that's better done with addins, as I'll discuss in the Addins section) or want to access some member. For example, the following code displays the total number of items in all projects in the solution:

```
lnItems = 0
loExplorer = _screen.oProjectExplorers(1)
for each loProject in loExplorer.oSolution.oProjects foxobject
    lnItems = lnItems + loProject.oProjectItems.Count
next loProject
messagebox(lnItems)
```

The class diagram shown in **Figure 15** shows the relationship between the main classes.

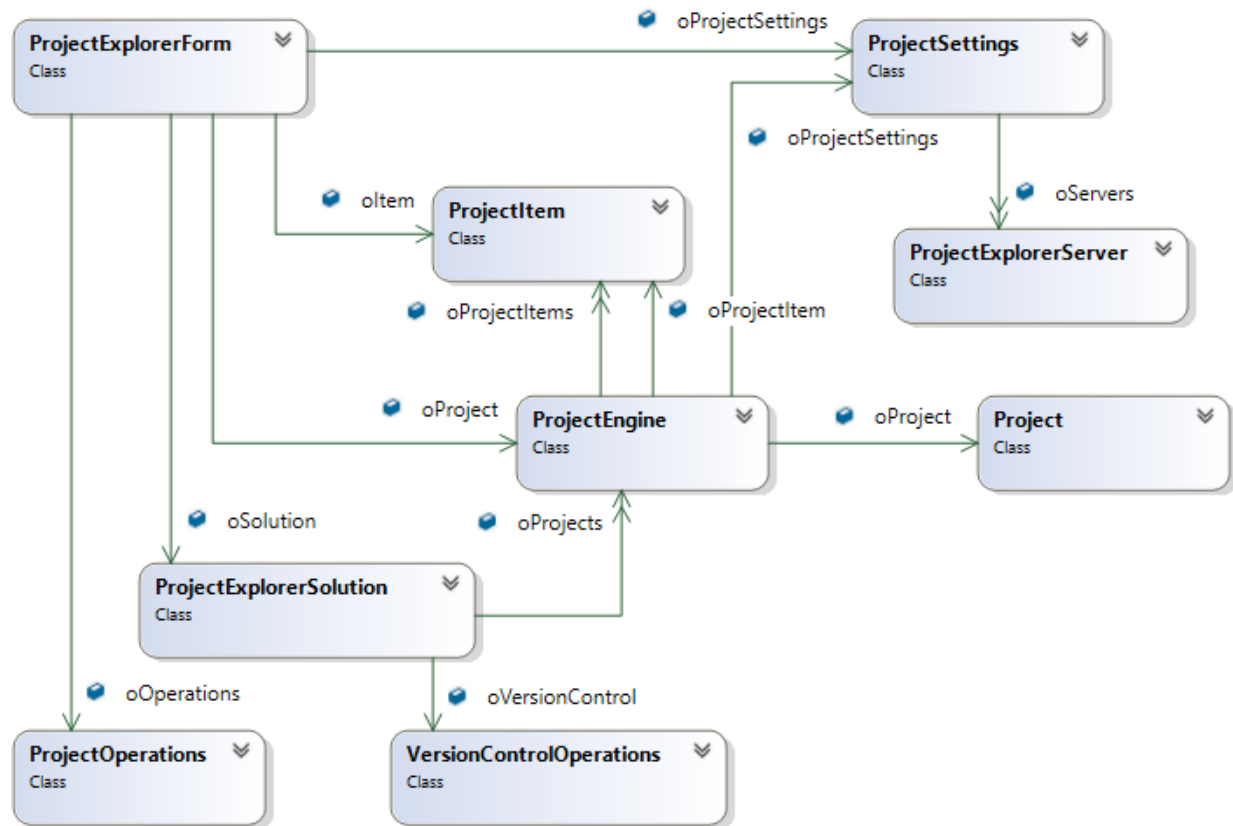


Figure 15. Class diagram for main Project Explorer classes.

Files

ProjectExplorer.app is the Project Explorer application. In the same folder as the app are the following files:

- **ProjectExplorer.pjx** and **pjt**: the project for Project Explorer.
- **System.app**: the GDIPlusX system file (GDIPlusX is another VFPX project); Project Explorer uses it to create images files of colored squares for the Category combo box.
- **ProjectExplorerCategories.dbf** and **cdx**: contains the category definitions. This file is created by copying **CategoriesSource.dbf** from the Source folder if it doesn't exist.
- **ProjectExplorerTags.dbf** and **cdx**: contains the tag definitions. This file is created by copying **TagSource.dbf** from the Source folder if it doesn't exist.
- **ProjectExplorerProjectHook.vcx**: contains **ProjectExplorerProjectHook**, a project hook that automatically runs **ProjectExplorer.app** when a project using that project hook is opened with **MODIFY PROJECT**.
- **ProjectExplorerSettings.xml**: contains the list of version control providers supported by Project Explorer and the classes that implement their behavior plus the expression used to determine the location of **FoxBin2PRG**. To support another

version control provider, subclass VersionControlOperations and add an entry for it to this file. Here's the content of this file:

```
<settings>
  <versioncontrol>
    <provider name="Mercurial" class="MercurialOperations"
      library="ProjectExplorerEngine.vcx" />
    <provider name="Git" class="GitOperations"
      library="ProjectExplorerEngine.vcx" />
  </versioncontrol>
  <foxbin2prg path="execscript(_screen.cThorDispatcher,
    'Thor_Proc_GetFoxBin2PrgFolder')" />
</settings>
```

The subdirectories of the Project Explorer main folder are:

- The Addins folder contains addin PRG files.
- The Source folder contains all the source code for Project Explorer.
- The Tests folder contains FoxUnit tests (FoxUnit is another VFPX project).
- The Thor folder contains a couple of PRGs used to add Project Explorer to Thor; see the next section for details.

Project Explorer creates the following files in the folder where your project exists:

- Solution.xml: the file listing the projects in the solution and their settings. Here's an example of this file:

```
<solution>
  <projects>
    <project name="demo1.pjx" buildaction="3" recompile="false"
      displayerrors="true" regenerate="false" runafterbuild="false"
      outputfile="Demo1.exe" />
    <project name="demo2.pjx" buildaction="5" recompile="false"
      displayerrors="true" regenerate="false" runafterbuild="false"
      outputfile="Demo2.dll" />
  </projects>
  <versioncontrol class="MercurialOperations"
    library="ProjectExplorerEngine.vcx" repository=""
    includeinversioncontrol="2" autocommit="true"
    fileaddmessage="Added file" fileremovemessage="Removed file"
    cleanupmessage="Project cleanup"
    savedsolutionmessage="Solution settings changed"
    buildmessage="Built the project: version {Project.VersionNumber}" />
</solution>
```

- *Project_Metadata.dbf*, *cdx*, and *fpt* (where *Project* is the name of the project file): the meta data for each item in the project. There is one table per project in the solution. The columns in this table are KEY, which contains a unique ID for the item, FORECOLOR, which contains the category number, TAGS, which is a carriage return delimited list of tags for the item, and CURRENT, which is used for internal purposes

(deciding which items have been removed from the project outside Project Explorer).

Project Explorer works with PJX files just like the Project Manager does. It only makes one change to a PJX file above what the Project Manager does: it assigns a unique ID (SYS(2015) value) to each file in the project and puts the ID into the DEVINFO field in the PJX file. As far as I can tell, DEVINFO is only used by the “H” record (the one for the project itself) so putting values into it doesn’t affect anything.

Running Project Explorer from Thor

Two PRGs in the Thor folder make it easy to run Project Explorer from Thor: Thor_Tool_ProjectExplorer.prg, which adds Project Explorer to the Thor Tools, Applications menu, and Thor_Proc_GetProjectExplorerFolder.prg, which returns the folder where Project Explorer is installed. Copy both of these PRGs to the Thor\Tools\My Tools subdirectory of your Thor folder. If you wish to assign a hot key to Project Explorer (I use Ctrl+Alt+P), use The Tool Definitions tab of the Thor Configuration dialog.

Project Manager bugs

While working on Project Explorer, I came across some bugs in the project object model that Project Explorer has to deal with:

- Calling Files.Add for an EXE adds it as type “x” (other file) rather than “Z” (application). Project Explorer makes sure it’s the correct type.
- Calling Files.Add for a table in a database returns the file object for the database not the table.
- A table in a database doesn’t have a record in the PJX file unless the table is included in the project. However, you can’t do that programmatically. Project Explorer handles it by manually adding or removing a record for the table from the PJX when you change the exclude status of the table.
- Calling Files.Add for a VCX sets it to the main file for the project if there isn’t one already. This is a throwback to the days when ActiveDoc was supported. Project Explorer turns that setting back off again.
- Speaking of main file, turning off the main file for a project doesn’t work properly: the MainFile property is read-only and while Project.SetMain("") works, it returns .F. and doesn’t clear MainFile until the project closed and reopened. Project Explorer hacks the PJX file to handle this.
- File.Run works from the Command window but not in code so Project Explorer has to manually run the item.

Addins

As I mentioned earlier, while project hooks are supported, addins are more powerful, easier to create and install, and there are more of them. Like project hooks, addins can prevent the normal execution of a function under certain conditions. However, addins can

do a lot more, such as adding buttons or other controls to the user interface or adding items to the shortcut or Project Explorer menu.

Addins auto-register themselves with Project Explorer. To add an addin to Project Explorer, create a PRG with the appropriate code in the Addins subdirectory of the Project Explorer folder. At startup, Project Explorer examines all PRGs in that folder and automatically registers any that are active addins.

Template.txt in the Addins folder (see **Listing 2**) shows what the content of an addin should contain (the header comments obviously aren't required). Note that this is procedural code rather than a class. This was a deliberate design decision because classes stay open even after they're no longer used and you have to close VFP or use CLEAR ALL to be able to edit the addins in that case.

Listing 2. Template.txt has the content an addin should contain.

```
*=====
* Program:      *** PUT NAME HERE
* Purpose:      *** PUT TEMPLATE PURPOSE HERE
* Author:       *** PUT AUTHOR NAME HERE
* Last Revision:*** PUT LAST REVISION DATE HERE
* Parameters:   tuParameter1 - a reference to an addin parameter object if
*               only one parameter is passed (meaning this is a
*               registration call) or a reference to an object; see the
*               documentation for the type of object passed for each
*               method
*               tuParameter2 - see the documentation for what's passed for
*               each method
*               tuParameter3 - see the documentation for what's passed for
*               each method
* Returns:      .T. if the method being hooked should continue to execute
*               or .F. if not. You can also return 0 to not continue
*               and signal failure, 1 to not continue and signal
*               success, or 2 to continue
*=====

lparameters tuParameter1, ;
            tuParameter2, ;
            tuParameter3

* If this is a registration call, tell the addin manager which method we're
* an addin for.

if pcount() = 1
    tuParameter1.Method = '*** specify method ***'
    tuParameter1.Active = .T. && set to .F. to disable addin
    tuParameter1.Name    = '*** specify descriptive name (optional) ***'
    tuParameter1.Order  = 1    && specify order to process (optional)
    return
endif

* This is an addin call, so do it.
```

```
*** put code here
```

```
*** return appropriate value; see notes above  
return .T.
```

The code structure was inspired by Thor, which uses a similar mechanism. At startup, Project Explorer calls each PRG in the Addins folder as a “registration” call to determine what the addin is for. It passes a single parameter (an execution call passes three parameters, even if some of them are .F.) which is a registration object with Method, Active, Name, and Order properties. As noted in Listing 2, you only have to set the Method property to indicate which Project Explorer method this should be an addin for. **Table 2** lists the different methods available. You can have as many addins for a particular method as you wish; the Order property determines in which order the addins are executed if there’s more than one for a given method.

Table 2. The Project Explorer addin methods.

Addin	When Executed	Parameters	Success Flag
Projects and solutions			
BeforeOpenSolution AfterOpenSolution	When a solution is opened	ProjectExplorerSolution object, solution folder	
BeforeGetProjectSettings AfterGetProjectSettings	When settings are retrieved from the project: at startup, after a project is built, and when the project is reverted	ProjectSettings object	
BeforeLoadTreeView AfterLoadTreeView	When the TreeView is loaded (e.g. when selecting a tag to display items for)	ProjectExplorerForm object	
AddNodeToTreeView	When an item is added to the TreeView	ProjectItem object, TreeView Node object	
AfterAddMenu	After the menu has been created	ProjectExplorerForm object	
AfterCreateShortcutMenu	When right-clicking the TreeView	ProjectExplorerForm object, ProjectExplorerShortcutMenu object	
OnActivate	When the Project Explorer window gets focus	ProjectExplorerForm object	
BeforeSaveProjectSettings AfterSaveProjectSettings	When you click Save for the project properties	ProjectSettings object	
BeforeBuildProject AfterBuildProject	When a project is built	Project object	Y
BeforeAddProjectToSolution AfterAddProjectToSolution	When a project is added to a solution	ProjectExplorerSolution object, path for PJX	Y
BeforeRemoveProjectFromSolution	When a project is removed	ProjectExplorerSolution object,	Y

Addin	When Executed	Parameters	Success Flag
AfterRemoveProjectFromSolution	from a solution	path for PJX	
BeforeAddVersionControl AfterAddVersionControl	When a solution is placed under version control	ProjectExplorerSolution object	Y
BeforeSaveSolution AfterSaveSolution	When a solution is created, a project is added or removed from a solution, you click OK in the Version Control Properties dialog, you click OK in the Build Options dialog, or you click Save for the project properties	ProjectExplorerSolution object	
BeforeCleanupSolution AfterCleanupSolution	When a solution is cleaned up	ProjectExplorerSolution object, .T. if object code is removed	
OnExit	When Project Explorer is closed	ProjectExplorerForm object	
Item management			
BeforeModifyItem AfterModifyItem	When an item is modified	ProjectItem object	Y
BeforeRunItem AfterRunItem	When an item is run	ProjectItem object	Y
BeforeRemoveItem AfterRemoveItem	When an item is removed from the project	ProjectItem object	Y
BeforeAddItem AfterAddItem	When an item is added to the project	Project object, filename	*** SEE TODO
BeforeNewItem AfterNewItem	When an item is created	Project object, ProjectItem object	
BeforeRenameItem AfterRenameItem	When an item is renamed	ProjectItem object, new name	
BeforeSaveProjectItem AfterSaveProjectItem	When you click Save for an item's properties	ProjectItem object	Y
GetDefaultMetaDataForItem	When an item is loaded for the first time	ProjectItem object	
Version control			
BeforeCreateRepository AfterCreateRepository	When a repository is created	Folder for repository	
BeforeAddFilesToVersionControl AfterAddFilesToVersionControl	When files are added to version control	Array of file names	
BeforeRemoveFilesFromVersionControl AfterRemoveFilesFromVersionControl	When files are removed from version control	Array of file names	

Addin	When Executed	Parameters	Success Flag
BeforeCommitFiles AfterCommitFiles	When files are committed	Commit message, array of file names	
BeforeRevertFiles AfterRevertFiles	When files are reverted	Array of file names	
BeforeCommitAllFiles AfterCommitAllFiles	When all changes are committed	Commit message, array of project file names	
BeforeRenameFileInVersionControl AfterRenameFileInVersionControl	When a file is renamed	Original file name and path, new name (stem only)	
BeforeRevisionHistory AfterRevisionHistory	When revision history for a file is displayed	File name and path	
BeforeVisualDiff AfterVisualDiff	When visual diff for a file is displayed	File name and path	
BeforeRepositoryBrowser AfterRepositoryBrowser	When the repository browser is displayed	None	

The code following the registration handler is the actual code executed when the addin is called. This code can do anything you wish and use the parameters passed in (identified in Table 2) as necessary. The return value of this code determines whether the method calling the addin continues or not. This is discussed in more detail in a moment.

Listing 3 is an example of a simple addin: it displays a message when an item is run. Note that you can disable the addin by simply setting Active to .F.

Listing 3. A simple addin that displays a message when an item is run.

```
lparameters toParameter1, ;
    tuParameter2, ;
    tuParameter3

* If this is a registration call, tell the addin manager which method we're
* an addin for.

if pcount() = 1
    toParameter1.Method = 'BeforeRunItem'
    toParameter1.Active = .T.
    return
endif

* Display a message.

messagebox('Before run addin for ' + toParameter1.ItemName)
return .T.
```

The Project Explorer addin mechanism was inspired by the Class Browser. Every method that supports an addin calls code like this:


```
This.oAddins.ExecuteAddin('MethodName', parameters)
```

or like this:

```
if not This.oAddins.ExecuteAddin('MethodName', parameters)
    return .F.
endif
```

or like this:

```
if not This.oAddins.ExecuteAddin('MethodName', parameters)
    return This.oAddins.lSuccess
endif
```

This.oAddins is a reference to a ProjectExplorerAddins object and *MethodName* is the name of the addin to call. *parameters* represents zero to three parameters; Table 2 specifies which parameters are passed to each addin.

The addin can return .T. to indicate the method should continue or .F. to prevent it from continuing. The code can also return a numeric value: 0 means the method should not continue because the operation failed or should fail, 1 means the method should not continue because the addin handled it successfully so the normal behavior should not execute (sort of like NODEFAULT in the method of a class), and 2 means the method should continue. Not all methods respect the numeric value; 2 means continue and anything else means don't. The "Success Flag" column in Table 2 indicates which methods return .T. but don't continue if the addin returns 1 (that is, they use code like the third example above). Note that typically only the Before methods care about continuing or not; the After methods simply call the addin and ignore the return value since the operation is done at that point (in other words, they use code like the first example above).

Examples

The code in **Listing 4**, taken from AddFileExplorerButton.prg, adds a button to Project Explorer's toolbar that, when clicked, opens File Explorer for the folder the selected file is in. It executes at startup because it's an addin for the OnStartup method. The execution code adds a button to the toolbar (the first parameter is a reference to the Project Explorer form and its oProjectToolbar member is the toolbar container) and positions it to the right of the Back button. The Refresh method of the button enables the button when an item that's a file (for example, not a class or field) is selected and the Click method calls ExecuteFile, a wrapper for the Windows API ShellExecute function, that's included in Project Explorer.

Listing 4. This OnStartup addin adds a button to the toolbar to open File Explorer for the folder of the selected file.

```
lparameters toParameter1, ;
    tuParameter2, ;
    tuParameter3
```

* If this is a registration call, tell the addin manager which method we're

```
* an addin for.

if pcount() = 1
    toParameter1.Method = 'OnStartup'
    toParameter1.Active = .T.
    return
endif

* Add a button to the toolbar that opens File Explorer for the selected file.

loToolbar = toParameter1.oProjectToolbar
loToolbar.AddObject('cmdOpenFileExplorer', 'FileExplorerButton')
loButton   = loToolbar.cmdOpenFileExplorer
loButton.Visible = .T.
loButton.Top   = loToolbar.cmdBack.Top
loButton.Left  = loToolbar.cmdBack.Left + loToolbar.cmdBack.Width + 5
loButton.ToolTipText = 'Open File Explorer'
return .T.

define class FileExplorerButton as ProjectExplorerToolbarButton of ;
    Source\ProjectExplorerButton.vcx

    function Init
        This.Picture = Thisform.cMainFolder + 'Addins\folder.png'
    endfunc

    function Click
        ExecuteFile(justpath(Thisform.oItem.Path))
    endfunc

    function Refresh
        This.Enabled = vartype(Thisform.oItem) = 'O' and Thisform.oItem.IsFile
    endfunc
enddefine
```

The code in **Listing 5**, taken from `AddPackToShortcutMenu.prg`, adds a `Pack File` function to the `TreeView`'s shortcut menu. It executes when the user right-clicks the `TreeView` because it's an addin for the `AfterCreateShortcutMenu` method. The second parameter is a reference to a `ProjectExplorerShortcutMenu` object, which is an object-oriented wrapper for the VFP shortcut menu system. `ProjectExplorerShortcutMenu`'s `AddMenuBar` method expects several parameters: the caption for the menu item, a command to execute when the item is selected (`loForm` evaluates to the Project Explorer form, which has a `cMainFolder` property containing the path for the Project Explorer application folder and an `oItem` property, which is a `ProjectItem` object for the selected item), an expression that's evaluated when the shortcut menu is displayed that determines whether the menu item is enabled, the path for the image file for the menu item, and the position the item appears at in the menu. In this case, the command to execute is `PackFile.prg` in the `Addins\Functions` folder, which accepts a filename as a parameter and packs that file. The menu item is disabled if no item is selected or if the selected item isn't a VFP binary file.

Listing 5. This AfterCreateShortcutMenu addin adds a Pack File function to the shortcut menu.

```
lparameters toParameter1, ;
    tuParameter2, ;
    tuParameter3

* If this is a registration call, tell the addin manager which method we're
* an addin for.

if pcount() = 1
    toParameter1.Method = 'AfterCreateShortcutMenu'
    toParameter1.Active = .T.
    toParameter1.Name   = 'Add Pack to Shortcut Menu'
    return
endif

* This is an addin call, so add "Pack File" as the third item in the shortcut
* menu.

tuParameter2.AddMenuBar('Pack File', ;
    "do (loForm.cMainFolder + 'Addins\Functions\PackFile') with loForm.oItem.Path", ;
    "vartype(loForm.oItem) <> 'O' or not loForm.oItem.IsBinary", ;
    , ;
    3)
return .T.
```

RunMainProgram.prg, shown in **Listing 6**, adds a function to the Project Explorer menu that runs the main program for the project. This saves you having to select the Code tag and scroll to find the correct startup program. It executes at startup because it's an addin for the AfterAddMenu method. The first parameter is a reference to the Project Explorer form, which has an oMenu member that's an instance of ProjectExplorerMenu, an object-oriented wrapper for the VFP menu system. Its FilePad member is an instance of a ProjectExplorerPad object and is a reference to the File pad. Its AddBar method accepts three parameters: the class for a ProjectExplorerBar subclass that represents a menu bar, the library for that class, and the name for the bar. In this case, it specifies the FileRunMainBar class inside RunMainProgram.prg (SYS(16) is used so the name of the PRG can be changed if necessary). FileRunMainBar defines what the bar looks like and what command executes when the menu item is chosen. In this case, it calls the RunItem method of the Project Explorer form, passing it the path for the main file of the project (the form's oProject member contains a reference to the ProjectEngine object for the selected project, and its oProject member contains a reference to the open VFP Project object).

Listing 6. This AfterAddMenu addin adds a function to the menu to run the main program for the project.

```
lparameters toParameter1, ;
    tuParameter2, ;
    tuParameter3

* If this is a registration call, tell the addin manager which method we're
* an addin for.

if pcount() = 1
```

```
toParameter1.Method = 'AfterAddMenu'
toParameter1.Active = .T.
return
endif

* Add a menu function to the File pad to run the main program for the project.

toParameter1.oMenu.ProjectExplorerPad.AddBar('RunMainBar', sys(16), 'RunMain')
toParameter1.oMenu.ProjectExplorerPad.Refresh()
return .T.

define class RunMainBar as ProjectExplorerBar of ;
    Source\ProjectExplorerMenu.vcx
    cCaption          = [Run Main Program]
    cStatusBarText    = [Runs the main program in the project]
    cOnClickCommand = [_screen.ActiveForm.RunItem()] + ;
        [_screen.ActiveForm.oProject.oProject.MainFile]]
    cSkipFor          = [empty(_screen.ActiveForm.oProject.oProject.MainFile)]
    cBarPosition      = [before ProjectExplorerExit]
enddefine
```

I like to use a version numbering system which has as the last four digits the encoded date of the build. This allows me to programmatically decode the version number to get the build date, which can be used, for example, to determine whether a user is entitled to this build if they did not renew their software maintenance (they're entitled to any build up to the date their maintenance lapsed). The addin shown in **Listing 7**, taken from SetVersionNumber.prg, automatically sets the version number for the project whenever it's built because it's an addin for the BeforeBuildProject method. The first parameter is a reference to the VFP Project object, so this code sets the last four digits of its VersionNumber property just before the project is built.

Listing 7. This BeforeBuildProject addin updates the project's version number when it's built.

```
lparameters toParameter1, ;
    tuParameter2, ;
    tuParameter3

* If this is a registration call, tell the addin manager which method we're
* an addin for.

if pcount() = 1
    toParameter1.Method = 'BeforeBuildProject'
    toParameter1.Active = .T.
    toParameter1.Name    = 'Set version number on build'
    toParameter1.Order   = 1
    return
endif

* This is an addin call, so do it.

lnJulian = val(sys(11, date())) - val(sys(11, {^2000-01-01}))
lcJulian = padl(transform(lnJulian), 4, '0')
toParameter1.VersionNumber = left(toParameter1.VersionNumber, ;
```

```
    rat('.', toParameter1.VersionNumber)) + lcJulian  
return .T.
```

Other ideas

Here are some other ideas for addins:

- Additional actions on the selected item, such as backup or compare to another item (for example, call `BeyondCompare`).
- Additional actions on the project: backup, zip files, etc.
- A function to show files in project folders that aren't in the project and an option to delete them individually or all.
- Drag and drop a table or field for rapidly coding table operations that use a list of field names, such as `INSERT INTO`. This can leverage code written by Rick Schummer in the Data Explorer.
- Show project statistics: number of classes, programs, reports, etc.

Ideas

Here are some ideas for future development of Project Explorer:

- The Exclude icon takes precedence over version control status icons, so if a file is excluded, you can't see its status. Some way to see both would be useful. One possibility is to show excluded files in a different color, such as light grey, but then that would take precedence over the file's category color.
- Although most of the operational code is in operations classes, such as `ProjectExplorerSolution`, `ProjectEngine`, and `ProjectOperations`, there is some code in the form class, `ProjectExplorerForm`. Refactoring that code out would allow a different UI to be substituted if desired.
- Speaking of refactoring, it's possible that the `ProjectOperations` and `ProjectSettings` classes could be merged into `ProjectEngine` to minimize the number of classes needed.
- The Project Manager supports changing the font settings for the `TreeView`. That could easily be done with Project Explorer's `TreeView` but is more work for the other controls since control spacing and form `MinWidth` and `MinHeight` are all affected.
- Project hook methods could be called when items in a database are created, modified, or removed.
- Renaming a file and then reverting that change needs some work, as there can be a lot to undo (especially renaming a table that belongs to a database container).

Summary

Project Explorer overcomes the many shortcomings of the VFP Project Manager. I've been using it in my day-to-day development instead of the VFP Project Manager for many months now. I look forward to any feedback you have when you start using it with your projects.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of VFPX: Open Source Treasure for the VFP Developer, Making Sense of Sedna and SP2, the What's New in Visual FoxPro series, Visual FoxPro Best Practices For The Next Ten Years, and The Hacker's Guide to Visual FoxPro 7.0. He was the technical editor of The Hacker's Guide to Visual FoxPro 6.0 and The Fundamentals. All of these books are from Hentzenwerke Publishing (<http://www.hentzenwerke.com>). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (<http://www.foxrockx.com>).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox and Southwest Xbase++ conferences (<http://www.swfox.net>). He is one of the administrators for the VFPX VFP community extensions Web site (<http://vfp.codeplex.com>). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (<http://tinyurl.com/ygnk73h>).

