



# Introducing Project Explorer

Doug Hennig

Stonefield Software Inc.

Email: [dhennig@stonefield.com](mailto:dhennig@stonefield.com)

Corporate Web sites: [www.stonefieldquery.com](http://www.stonefieldquery.com)

[www.stonefieldsoftware.com](http://www.stonefieldsoftware.com)

Personal Web site : [www.DougHennig.com](http://www.DougHennig.com)

Blog: [DougHennig.BlogSpot.com](http://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 (<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 4**). Project Explorer overcomes the shortcomings I listed above and others as well, and provides a more modern user interface.

### 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 (**Figure 1**), and choose Download ZIP. Unzip the downloaded file in any folder you wish.



**Figure 1.** You can download Project Explorer from GitHub.

- You can use the Thor Check for Updates function to download Project Explorer; see **Figure 2**.

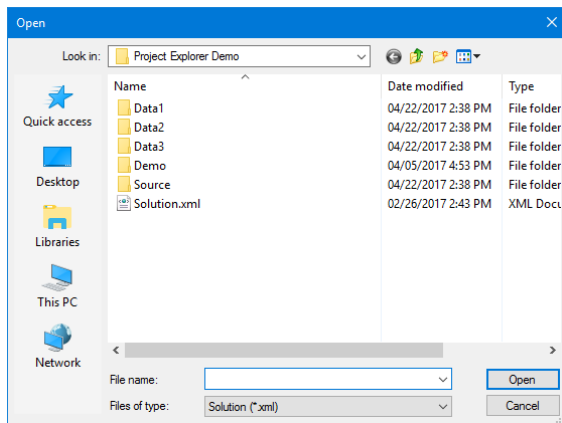


**Figure 2.** You can download Project Explorer using Thor.

## A tour of Project Explorer

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

- Run ProjectExplorer.app. If a single solution file (more about that in a moment) exists in the current folder, it's opened automatically. If a single project file exists in the current folder, a solution file is automatically created for it and opened. Otherwise, it displays a dialog prompting you to select a solution or a PJX file (**Figure 3**).
- Pass ProjectExplorer.app a path to a PJX file, a solution file, or a project object (that is, something like `_VFP.ActiveProject`). If you pass "?," it'll display a message showing the parameters you can pass it. Pass it .T. for the second parameter to display a dialog prompting you to select a solution or PJX file.



**Figure 3.** You are prompted to open a solution or a project.

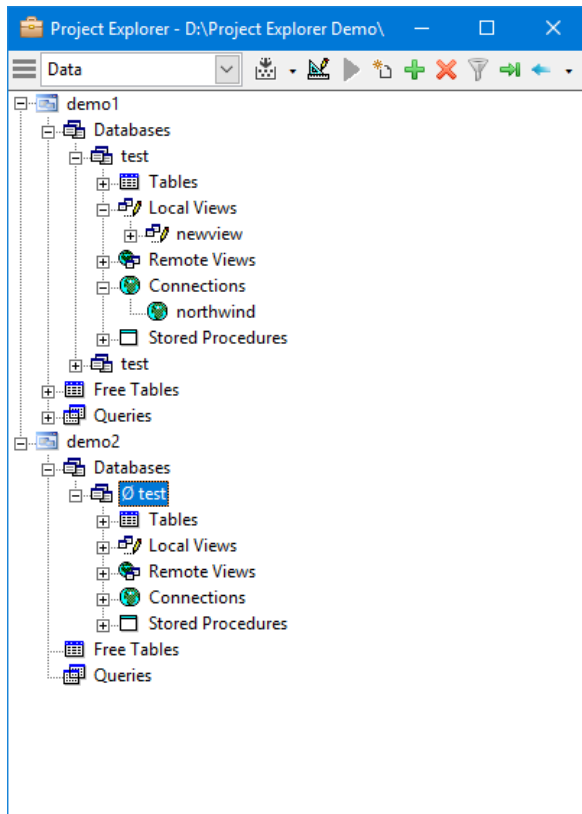
You can, of course, launch Project Explorer from a Thor menu item or hotkey. This is discussed later in this document.

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.

Project Explorer works with PJX files just like the Project Manager does. It also creates a solution file in the project folder. In earlier versions of Project Explorer, the solution file was named Solution.xml and only one was allowed per folder. Newer versions name it by default after the project file (although you can change the name) with an SLX extension, and you can have as many solution files in a folder as you wish. The solution file simply contains the list of projects and a few settings. We'll look at the structure of a 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 4**). 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:

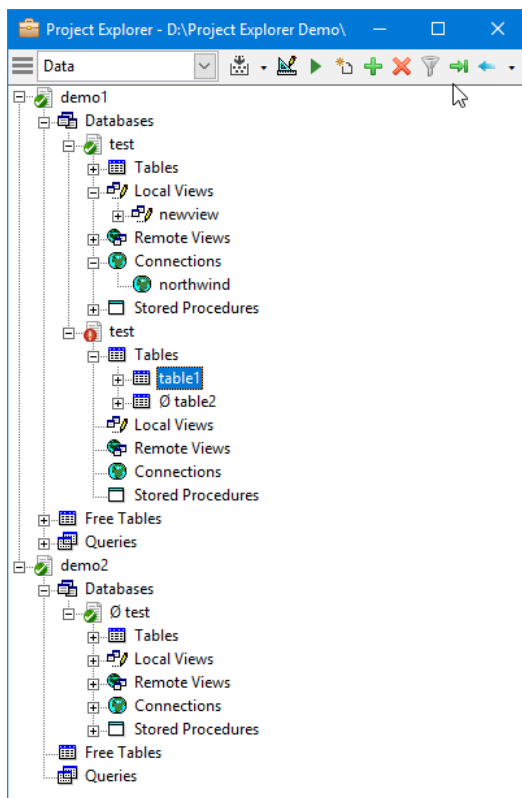


**Figure 4.** The user interface for Project Explorer is more modern than the Project Manager.

- 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 (click the "hamburger" button at the left of the toolbar) 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 4 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.

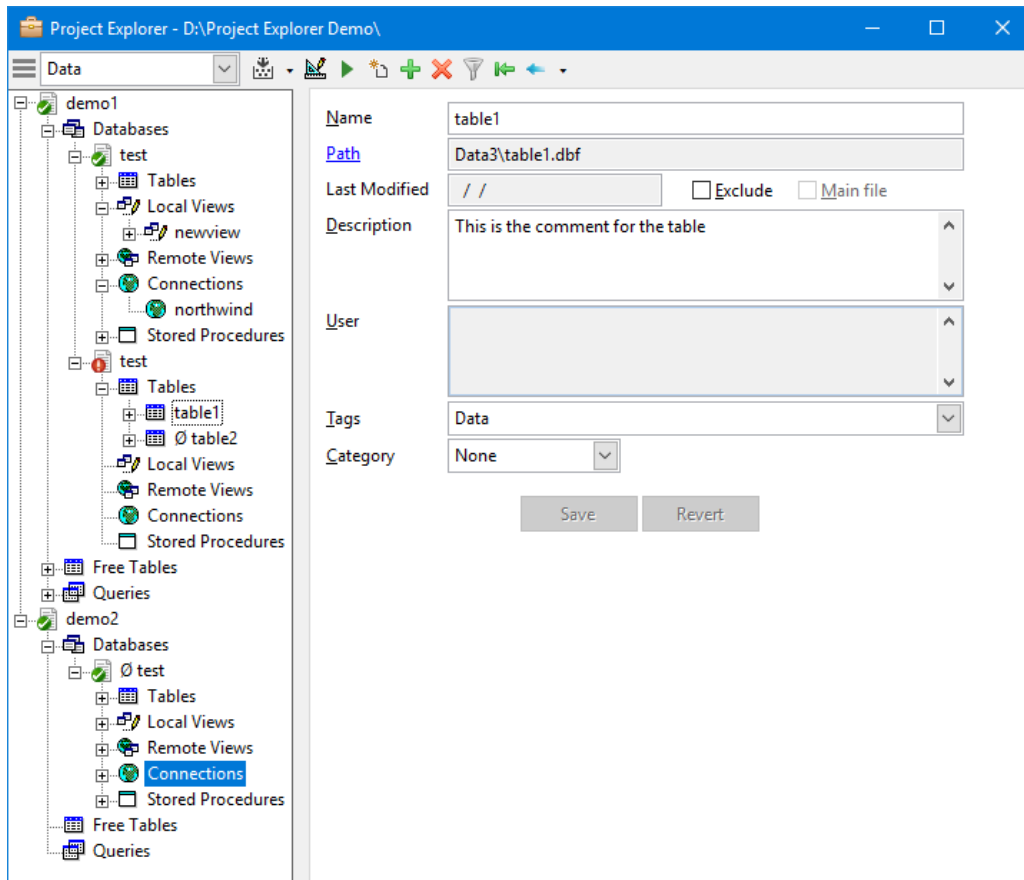
- The TreeView uses the same icons as the Project Manager does for file types, including Ø for excluded files, with a one exception: as you can see in **Figure 5**, when a solution is under version control, the icon for a file displays its status rather than type. The two projects and the first database 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) appears as  because it's been modified and not yet committed.



**Figure 5.** 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 6** (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 6.** 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. Click the link to open a File Explorer window for the folder the item is in.
- **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. Click the link to jump to the class item.
- **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. Click the link to jump to the class library item.
- **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 or form. This is always read-only and only displays for classes and forms. Click the link to jump to the include file.
- **OLEPublic:** turned on if the class is a COM server. This only displays for classes.
- **Icon:** the custom icon for the class displayed in the Project Manager, the Class Browser, and Project Explorer. This is only displays for classes. Click the image to select the desired icon file; if you select Cancel, you are asked if you want to remove the icon for the class.
- **Toolbar icon:** the custom icon for the class displayed in the Form Control Toolbar. This is only displays for classes. Click the image to select the desired icon file; if you select Cancel, you are asked if you want to remove the icon for the class.
- **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 fields and indexes in free tables and stored procedures since there's nowhere to store it.

You can right-click the editbox and choose Zoom to display the content in a form you can resize and change the font for more convenient editing.

- **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. Like *Description*, you can right-click and choose Zoom to display the content in an editing form.
- **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.

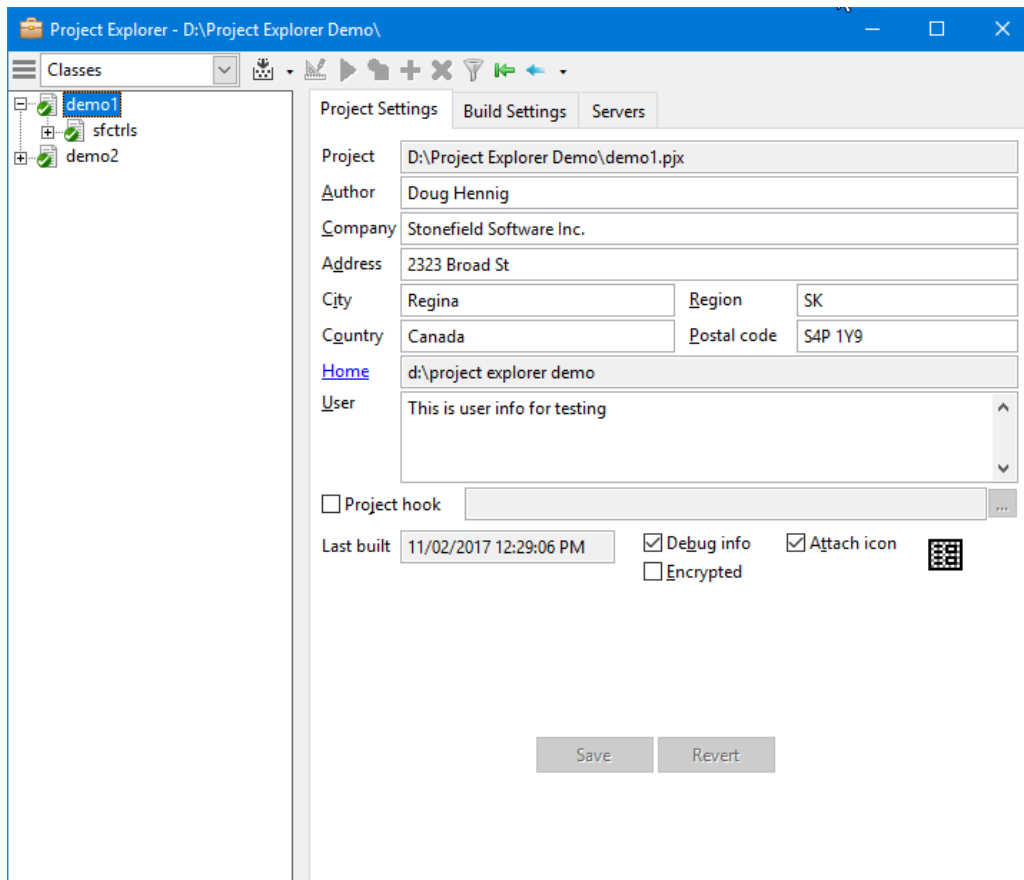
Click the Save button to save any changes or Revert to restore the previous property values. If the item is a class library, a database, a table, or a view and you changed Tags or



Category, you're asked whether you want the child items (for example, classes in the class library) to have the same tags and category; if so, Project Explorer updates those items as well.

### Project properties

When you select a project in Project Explorer, it displays the properties of the project at the right (**Figure 7**). 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:

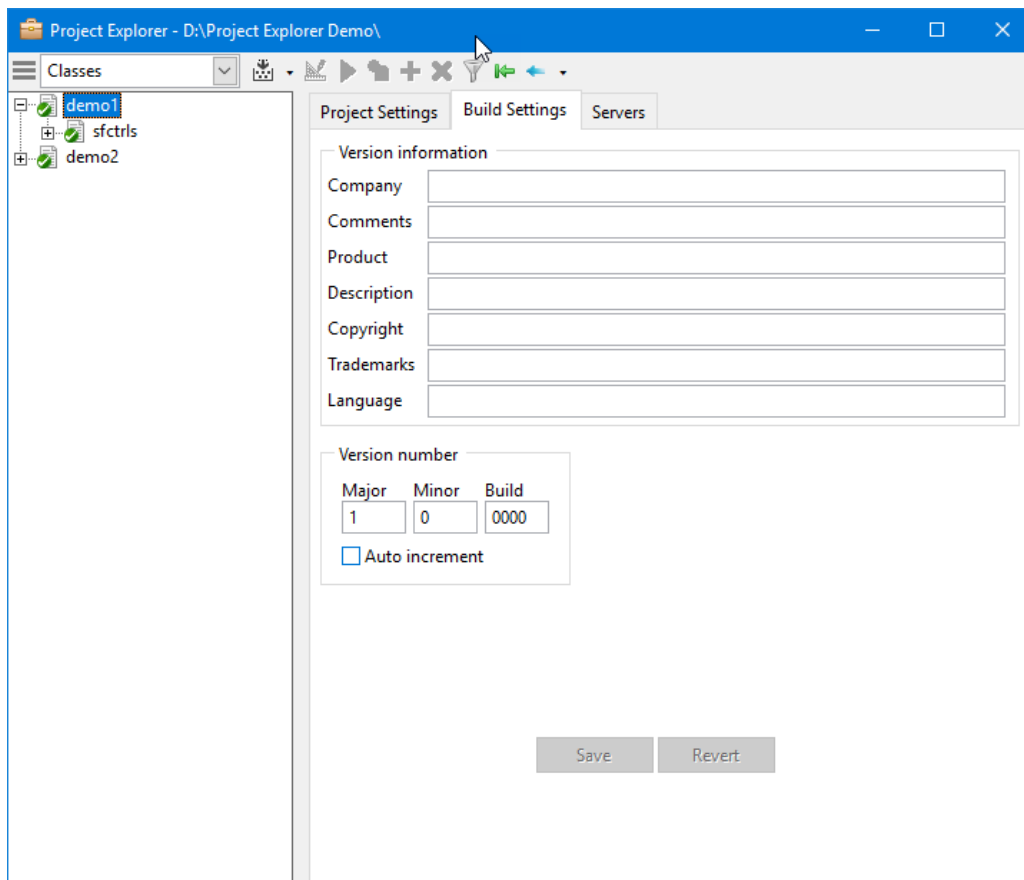


**Figure 7.** Project Explorer replaces several modal dialogs with the properties of the selected project.

- **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. Click the link to open a File Explorer window for the folder the project is in.
- **User:** user-defined information for the project itself. This is stored in the “H” record of the PJX file. You can right-click the editbox and choose Zoom to display the content in a form you can resize and change the font for more convenient editing.

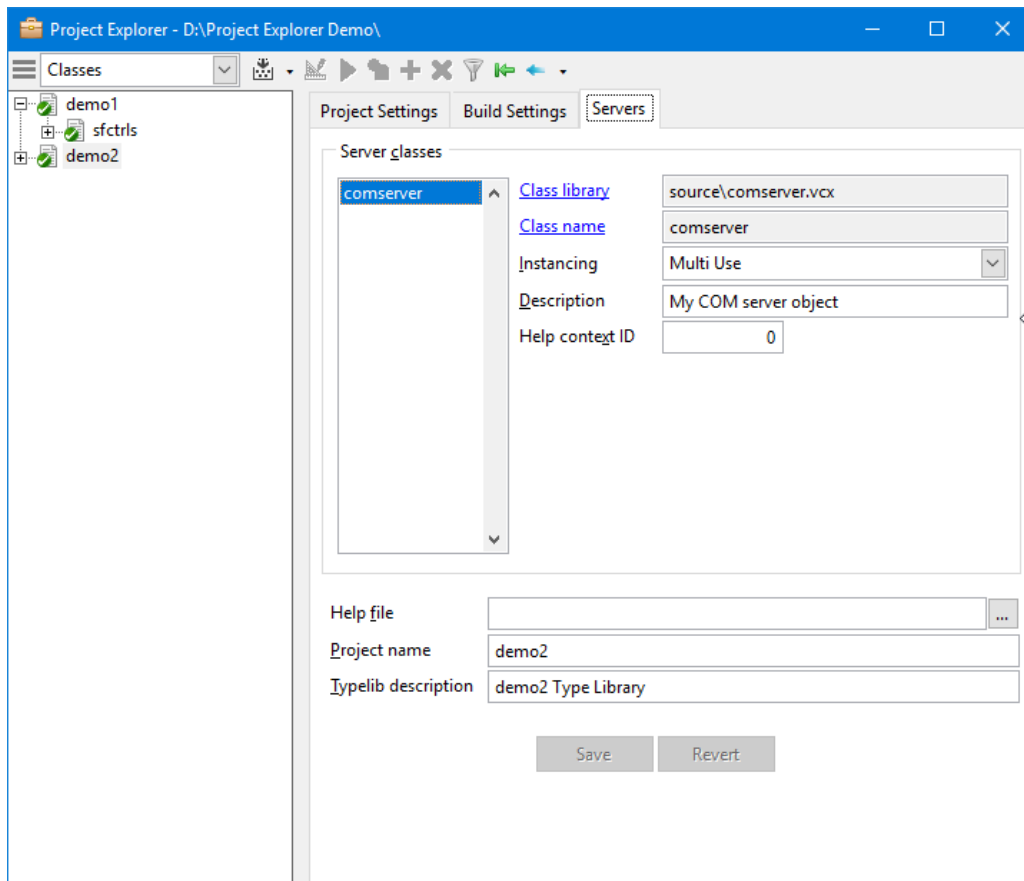
- **Project hook:** to add a project hook to a project, turn this on, click the button with the ellipsis (...) beside the project hook class name, and select a VCX and a class. You'll get a warning if the class you select isn't a subclass of ProjectHook. To remove the project hook for the project, turn 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 8**) 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 8.** You have to display two modal dialogs to get at these settings in the Project Manager.

The properties on the Servers page (**Figure 9**) 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. For *Class library* and *Class name*, click the link to jump to the class library or class item.



**Figure 9.** 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.

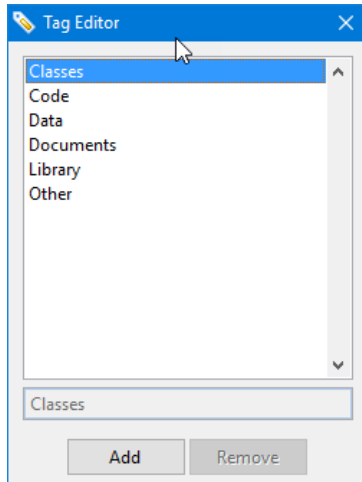
Project Explorer provides two ways to categorize items: by tags and by category.

### 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 Project Explorer menu (**Figure 10**).



**Figure 10.** The Tag Editor allows you to define your own tags.

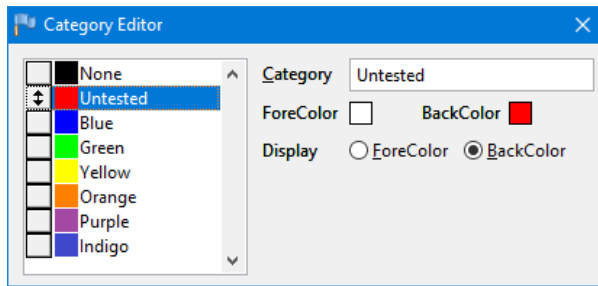
The “built-in” tags (the ones representing Project Manager tabs) can’t be edited or removed, but you can click the Add button and enter a tag name to create a new one or select one and change its name or remove it. Tags are stored in a table named ProjectExplorerTags.dbf in the Project Explorer folder.

The Tag combobox in the toolbar at the top of the window tells Project Explorer to display only those items containing that tag (you can also choose All to select all tags). 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 (it affects both the foreground and background colors). 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 11**), available from the Project Explorer menu, to change the names and even the colors.



**Figure 11.** 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 squares to select new foreground and background colors (the *Display* setting determines which color is used in the listbox). 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 as 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.

You can assign one or more tags and/or a category to a group of items using the Add or Remove Tags/Categories function in the Project Explorer menu (**Figure 12**). Turn on *Tags* and select the tags to use if you want to process tags. Turn on *Category* and select the category you want to use if you want to process a category. You can do both at the same time if you wish. Specify whether you want to add or remove the tags and/or category. If you want the changes applied to child items, such as classes in a class library or tables in a database, turn on *Apply to child items of matching items*. Enter a VFP expression into the Filter editbox to specify which items to add the tags to. 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 add the tags to forms, use the following filter expression:

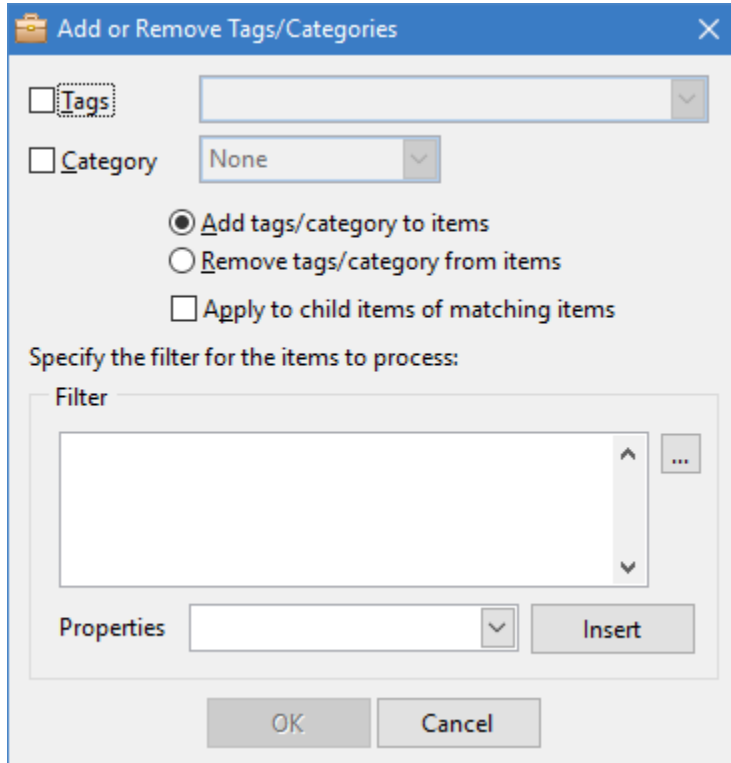
```
Item.TypeName = 'Form'
```

Here's a filter expression for adding a tag to all graphics files:

```
inlist(lower(justext(Item.Path)), 'bmp', 'png', 'jpg', 'jpeg', 'ico', 'gif')
```

Click the “...” button to display the VFP Expression Builder dialog. You can also right-click the editbox and choose Zoom to display the content in a form you can resize and change the font for more convenient editing.


Item filters are discussed in more detail in the Sorting and filtering section.

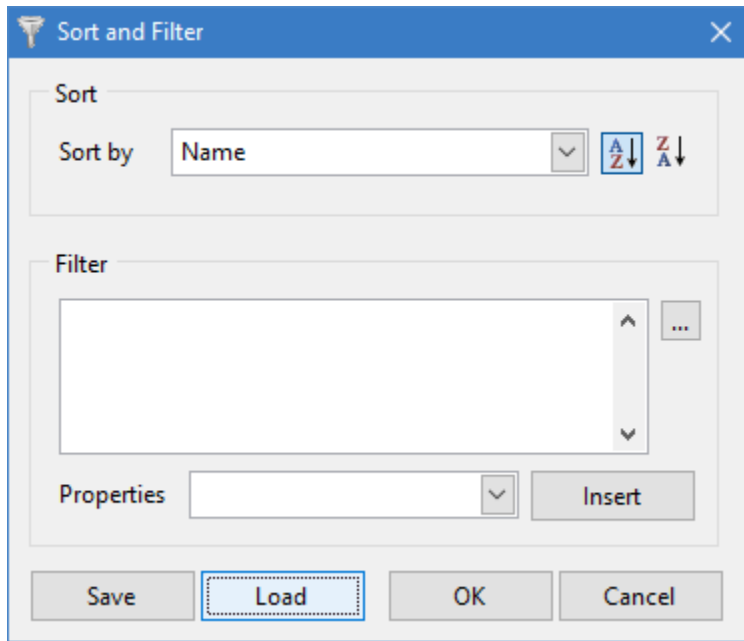


**Figure 12.** You can assign tags to a group of items using the Assign Tags to Items function.

Click OK to add the tags to all items matching the filter condition.

### Sorting and filtering

The Sort and Filter function (**Figure 13**), available in the Project Explorer menu and the toolbar (  ), gives you control over which items appear in the TreeView and in what order.



**Figure 13.** 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
```

Click the “...” button to display the VFP Expression Builder dialog. You can also right-click the editbox and choose Zoom to display the content in a form you can resize and change the font for more convenient editing.

**Table 1** lists the more useful properties.

**Table 1.** The more useful properties of project items.

Property	Description
CategoryName	The category for the item (“None” by default)
Exclude	.T. if the item is excluded from the project
IncludeFile	The name of the include file for the class or form
IsBinary	.T. if this is a VFP binary file


IsFile	.T. if this is a file (for example, .F. for a class)
ItemBaseClass	The base class for the class or form
ItemClass	The class for the form
ItemLibrary	The VCX for the class specified in ItemClass
ItemName	The name of the item
ItemParentClass	The parent class for the class
ItemParentLibrary	The VCX for the class specified in ItemParentClass
LastModified	The date and time the item was last modified
MainFile	.T. if this is the main file for the project
ParentPath	The path for the item's parent (for example, the DBC in the case of a table in a database)
ParentType	The parent type (see Type)
Path	The path for the item
Tags	A comma-delimited list of tags
Type	The item type; for example, "P" for program, "K" for form, etc. See the FILETYPE_* constants in FoxUser.h and ProjectExplorer.h for the codes for each type
TypeName	The full name of the item type; for example, "Program" or "Form"
User	The value of the User property
VersionControlStatus	The item's version control status; see the ccVC_STATUS_* constants in ProjectExplorer.h for the codes (the same as Mercurial and Git use)

When a filter has been set or the sort changed, the tooltip for the Sort & Filter button in the toolbar displays the current filter and sort settings. To reset them to default, right-click the button. Alternatively, to clear the filter, choose Sort and Filter from the Project Explorer menu again, clear the editbox, and click OK.

To save the sort and filter for future use, click the Save button and specify the name and path for a .filter file (an XML file containing the settings). To load a sort and filter, click the Load button and choose the desired .filter file.



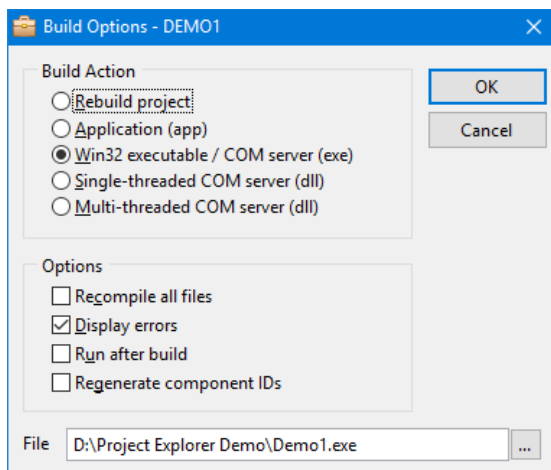
## Building

Click the  button in the toolbar to display the Build Options dialog (**Figure 14**) 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 14.** 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.

Note: if you get an error during the build process, such as "File cannot be closed because outstanding references exist," close Project Explorer, CLEAR ALL and CLOSE ALL, then open Project Explorer and try again.


### 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* or right-click the item and choose Modify from the shortcut menu. 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* or right-click the item and choose Run from the shortcut menu (it appears as "Browse" for tables and views). In the case of programs, forms, menus, and applications, the item is run. For non-form classes, the class is instantiated and added to \_SCREEN at position 0, 0. For form classes, the class is instantiated and a reference to it added to \_SCREEN. 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.

Here are some tips for running code from Project Explorer:


- For obvious reasons, any code run from Project Explorer should not do CLOSE ALL or CLEAR ALL.
- I found running an application using controls from Carlos Alloti's ctl32 library a bit problematic. The issue is that some of the controls aren't completely released until later than you think and if your clean up code does something like SET PROCEDURE TO, it can cause errors and even crashes. The solution is to use RemoveObject to specifically remove the controls in the Destroy method of your forms. For example, if you use Emerson Reed's ThemedExplorerBar control from his Themed Controls VFPX project, you may wish to add the following to the Destroy method of the control:

```
This.RemoveObject('ctl32_scrollbar')
```

- Error handling can be a little more complicated. Project Explorer wraps the DO command in a TRY structure. My error handler allows the program to continue (depending on the severity of the error) by using RETURN TO to return to the method containing the READ EVENTS statement for the application. The problem is that you can't use RETURN TO within a TRY; attempting to do so is an untrappable error. Complicating this is that SYS(2410), which theoretically is supposed to tell you if a TRY is in effect, is essentially useless because it's easily fooled by a

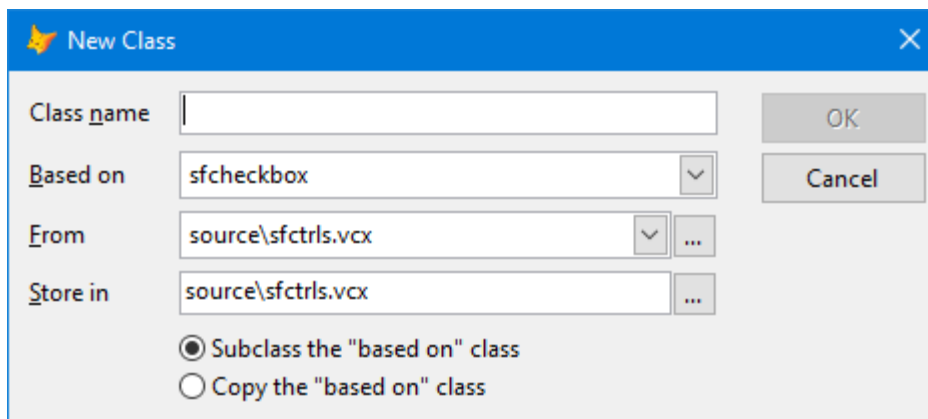
combination of ON ERROR, Error method, and TRY error handling strategies. So, Project Explorer creates a private variable, which is visible to the called code, named `plInsideTry` and sets it to `.T`. Before using RETURN TO in your code, check whether `plInsideTry` exists and if so whether it contains `.T`. In that case, do not use RETURN TO.

### Creating

Click the New button (  ) to create a new item of the same type as the selected item (if the *Add and New allow any file type* option, discussed later, is turned off) or of any type (if that option is turned on). This button is disabled for application, field, index, API library, and “other” items (if that option is turned off) and enabled for all other types. For connections and views, you’re prompted for a name. For other types except classes and forms, 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.

When you click this item for classes, the New Class dialog (**Figure 15**) appears. It has similar functionality to that dialog in the Project Manager, with these additional features:

- *Based on* is set to the name of the selected class if there is one. This makes it easy to subclass an existing class by simply selecting it and clicking the New button.
- *From* is set to the selected VCX but it’s a combobox containing the ten most recently used class libraries, so you can select one from the list. *Based on* adjusts to display the classes in the selected library. The libraries are listed in most recent to least recent order.
- You can create a new class by subclassing the *Based on* class or by copying it (the equivalent of dragging a class from one VCX to another and then renaming it in the Project Manager).

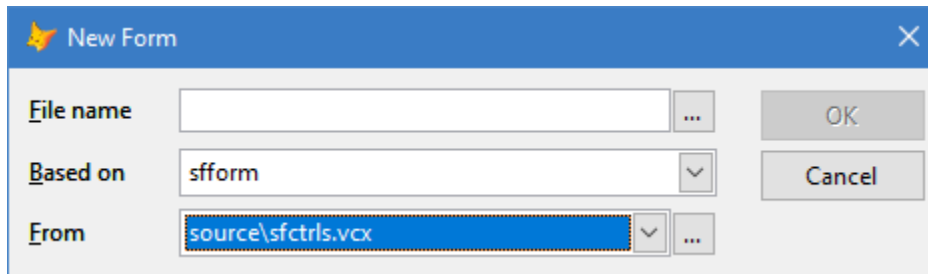


**Figure 15.** The New Class dialog displays when creating a new class.

There are a couple of ways to choose which class to base a new form on:

- When a form class is selected, right-click the class and choose Create Form from Class, then specify the name and path of the new form in the file dialog that appears.

- When forms are displayed, click the New button to display the New Form dialog (**Figure 16**). The *Based on* and *From* controls work the same as they do in the New Class dialog.



**Figure 16.** The New Form dialog appears when creating a new form.

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

### Adding

Click the Add button (+) to add an item of the same type as the selected item (if the *Add and New allow any file type* option, discussed later, is turned off) or of any type (if that option is turned on) to the project. You can select as many files to add as you wish. This button is only enabled for item types that are files (if that option is turned off). When you add a table to a database, it's added both to the database and to the project.

### Removing

Click the Remove button (X) 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 Project Explorer menu has several functions for managing solutions:

- *Open Solution* displays a dialog so you can open a project or solution.
- *Add Project to Solution* adds a project to the solution. Note that you can only add a project in the same folder structure as the solution; that is, a PJX file in the same folder as the solution file or a subdirectory of that folder.
- *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.

If you want to have more than one solution in a folder, the solution files must have an SLX extension rather than be named Solution.xml because if there's a file with that name,

Project Explorer automatically opens it without prompting you. When you open Project Explorer for the first time in a certain folder, you may be prompted to select a project. Doing so creates a solution file named for that project but with an SLX extension. To create another solution file for a different project in the same folder, run Project Explorer and pass it .T. for the second parameter. For example:







```
do <path>\ProjectExplorer.app with '', .T.
```

Project Explorer prompts you to select a project and then creates a solution file for that project. From now on, Project Explorer prompts you to select which solution file to work with when you open it.

### 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 2** for a list of the icons and their meanings.

**Table 2.** Version control status icons.

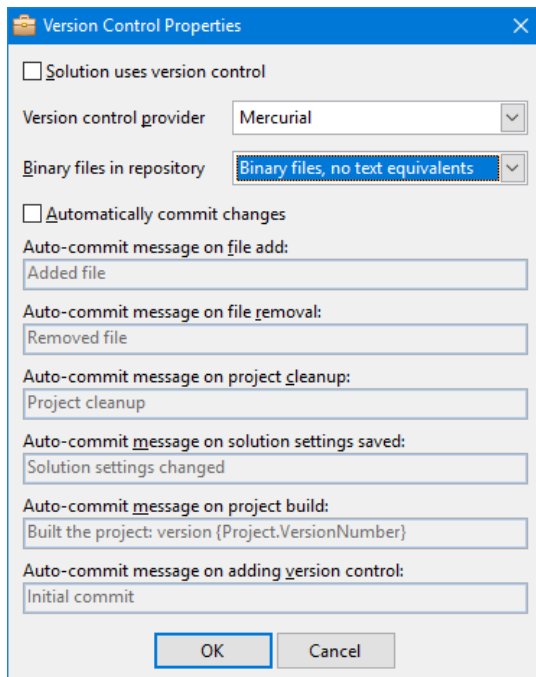
Icon	Meaning
	Unversioned
	Ignored
	Added
	Removed
	Modified
	Clean (unmodified)

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.
- **Convert Binary to Text:** creates a text equivalent of the file. This is useful, for example, for generating the text equivalent of a table after browsing it and added, editing, or deleting records. Because this is a potentially destructive operation (the text equivalent is overwritten), you are asked to confirm that you want to proceed.

- **Convert Text to Binary:** recreates the file from its text equivalent. This can be used after merging or updating the text equivalent from another branch or different repository so you get the updated binary file. Because this is a potentially destructive operation (the binary file is overwritten), you are asked to confirm that you want to proceed.
- **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 Project Explorer menu to display the dialog shown in **Figure 17**. This dialog has the following settings:



**Figure 17.** 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.
- **Binary files in repository:** specifies whether binary files are included in the repository. This is discussed in detail later.
- **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 whether binary files are included in version control, as specified by the *Binary files in repository* setting:

- **Binary files, no text equivalents:** 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, no binary files:** 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.

- Include both in repository: all files in the project plus the text equivalents of VFP binary files are included in version control.

If *Binary files in repository* is set to anything but *Binary files, no text equivalents*, 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 *Include both in repository* 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 *Include both in repository* is used.
- When you revert changes to the class library and *Text equivalents, no binary files* is used, the VC2 file is reverted and FoxBin2PRG is told to regenerate the VCX and VCT files from it. If *Include both in repository* 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.

After you click OK in the Version Control Properties dialog, you are asked if you want Project Explorer to create a repository for the solution. Choose No if a repository already exists, in which case you're asked to locate the folder containing the repository folder (that is, the parent of the .git or .hg folder).

One thing to note about version control is that some providers are case-sensitive for filenames. I'm not clear about VFP's rules about filename case; sometimes I see files with a "prg" extension and sometimes it's "PRG," and sometime a file that was "prg" is now "PRG." Although Project Explorer has code to handle filename case, I recommend turning off case sensitivity if your provider supports it. For Git, use these command line directives to turn it off:

```
git config --global core.ignorecase true
git config --system core.ignorecase true
```

Mercurial respects the case-sensitivity of the operating system, so for Windows, it's case-insensitive.

Also note that Project Explorer requires the latest version of FoxBin2PRG as it fixes a bug in handling PJX files. As of this writing, that version isn't available through the Thor Check for

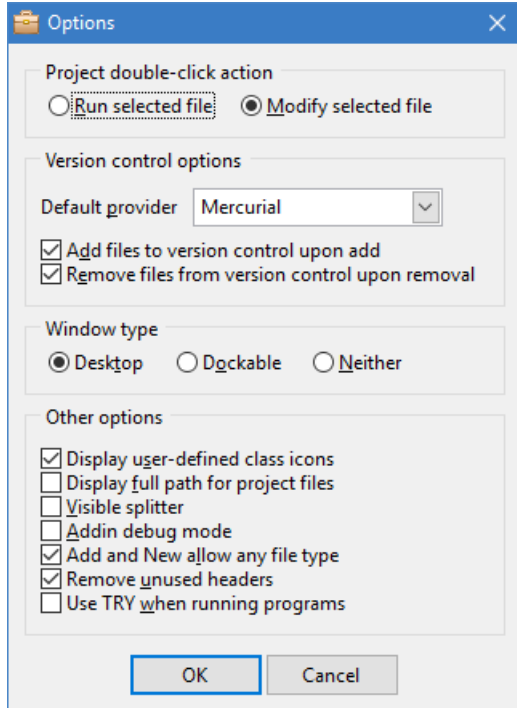


Updates process, so be sure to get the latest version from the FoxBin2PRG repository (<https://github.com/fdbozzo/foxbin2prg>).

### Other functionality

#### Project Explorer options

The *Options* function in the Project Explorer menu displays the dialog shown in **Figure 18**. It has the following options:



**Figure 18.** 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.
- **Window type:** choose *Desktop* to allow the Project Explorer window's Desktop property set to .T. Choose *Dockable* to allow the Project Explorer window to be

docked with other windows such as the Command window. In both cases, the window can be moved outside the VFP desktop and it's always on top. Choose *Neither* if you want the window inside the VFP desktop, which means other windows can be on top of it.

- **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.
- **Add and New allow any file type:** turn this on to allow the Add and New functions to display a dialog in which you can choose any file type. Turn it off to only allow a file of the selected type to be chosen; for example, if a form is currently selected, Add and New only allow you to add or create a form.
- **Remove unused headers:** turn this on to remove headers that don't have any items under them in the TreeView, such as "Labels" if there aren't any labels in the project.
- **Use TRY when running programs:** turn this on to use a TRY structure when running a program or application. If your code uses an error handler that uses RETURN TO, this will cause a problem because RETURN TO isn't allowed within a TRY, so in that case, turn this setting off.

### 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, but if you drag a control to the column of a grid (the column must be selected in the Properties window first for this to work correctly) and that column contains the default Text1 object, you're asked if you want it removed and if so, the new control becomes the current one in the grid.

- 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. Right-click the TreeView and choose Refresh from the shortcut menu 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.

### Invoking the project builder

Right-click the TreeView and choose Builder from the shortcut menu to invoke the same builder or builder dialog you would see in the Project Manager.

### 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.
- The Files page of the Project Information dialog.
- 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.

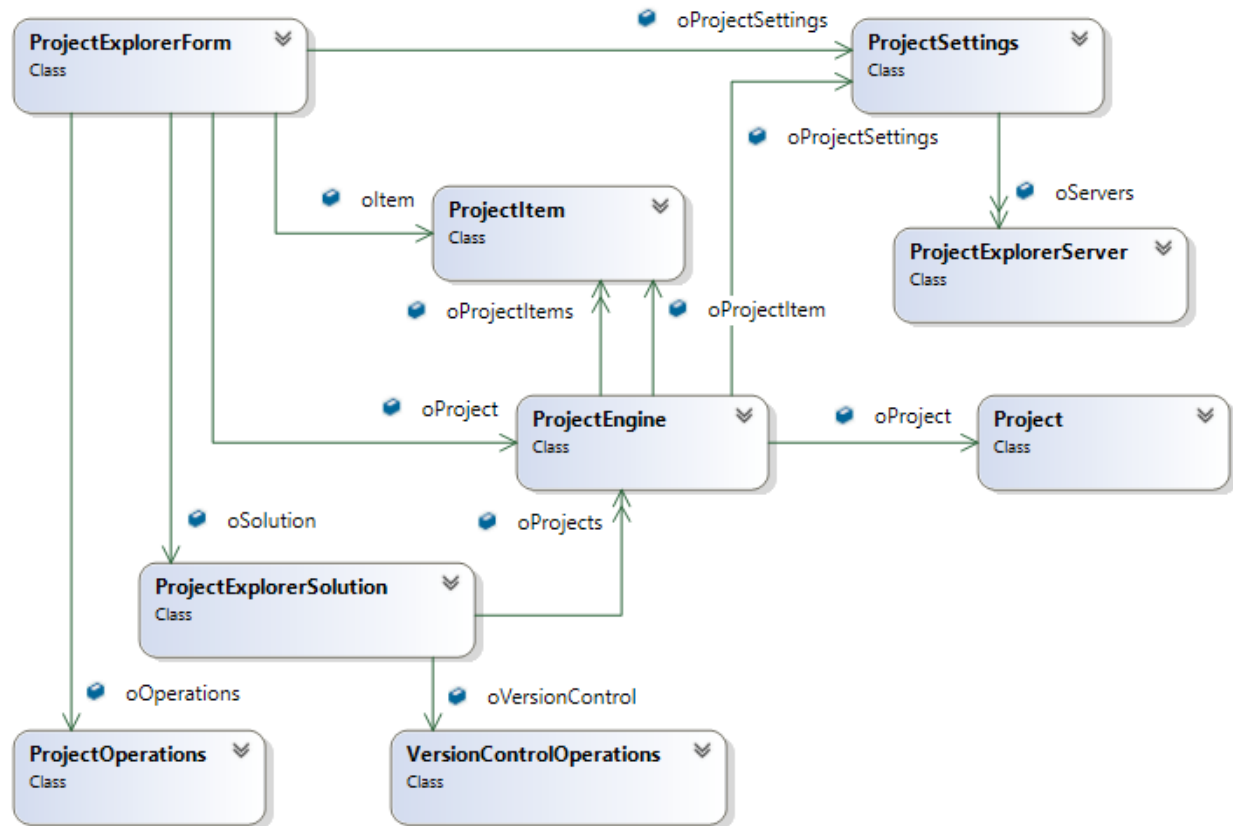
`ProjectOperations` is a small class that performs operations on a project. It has methods such as `AddItem`, `EditItem`, `RunItem`, and `RemoveItem` which do what their names imply.

`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 and *Dockable* settings turned off), `ProjectExplorerFormDesktop` (used if *Window has Desktop turned on* is turned on), and `ProjectExplorerFormDockable` (used if *Dockable* 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. The class diagram shown in **Figure 19** shows the relationship between the main classes.



**Figure 19.** Class diagram for main Project Explorer classes.

ProjectExplorerForm is a subclass of ProjectExplorerExplorerFormTreeView, a renamed version of an Explorer-style class I've discussed before (see my white paper titled "Creating Explorer Interfaces in Visual FoxPro" at <http://doughennig.com/papers/default.html>). This class and several related classes in ProjectExplorerExplorer.vcx and ProjectExplorerTreeView.vcx provide the basics of TreeView and Explorer-style form 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 name and path for the solution as the key (you can also use an index number). For example, if I open C:\My Projects\Customer A\Main Application\MyProject.slx, I can reference the open Project Explorer using:

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

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)
```

One other class you may be interested in is `ProjectExplorerWindowManager`. This class is used so Project Explorer knows when a designer window, such as the Form Designer, is closed so it can take the appropriate action. This class' `SetBinding` method locates the desired designer's window by its caption and uses `BINDEVENT()` to bind to its `Destroy` event. When that event occurs, the class raises its own `WindowDestroyEvent`. The `EditItem` and `NewItem` methods of `ProjectExplorerForm` bind to `WindowDestroyEvent` so the form receives notice when the designer is closed. Because the user could open more than one designer at a time, a collection of the handles of the windows and the items they're editing is stored in the `oWindows` property of the form.

### 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.
- `ProjectExplorerSettings.xml`: contains the list of version control providers supported by Project Explorer and the classes that implement their behavior plus the path for the binary to text converter (currently only `FoxBin2PRG` is supported). To support another version control provider, subclass `VersionControlOperations` and add an entry for it to this file. To specify that the path for the binary to text converter is an expression that must be evaluated, surround it with curly braces. To specify that you're not using any binary to text converter, set the path to blank. 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>
  <textconverter 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 VF PX project).
- The ThorUpdater folder contains files necessary to support the Thor Check for Updates process; see <https://vfpx.github.io/thorupdate/> for details on how that works.

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

- The solution file: the file, named either Solution.xml or a file with an SLX extension, 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 beyond 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.

For items that aren't files, such as classes, which don't have a record in the PJX, the KEY value stored in the meta data table is as follows:

- For a class, the ID is *ParentID~ClassName*. For example, for a class named MyClass in MyClasses.vcx (which has an ID of \_4WK001AXW), KEY is \_4WK001AXW~myclass.
- For a field or index in a free table, the ID is *Field~ParentID~FieldName* or *Index~ParentID~TagName*. For example, for a field named CUSTOMERID in Customers.dbf (which has an ID of \_4WK001AYY), KEY is Field~\_4WK001AYY~customerid.
- For a table, view, connection, or stored procedure in a database, the ID is *ParentID~Name*. For example, for a table named Customers in MyDatabase.dbc (which has an ID of \_4WD11QLRS), KEY is \_4WD11QLRS~customers.
- For a field or index in a table or view in a database, the ID is *Field~DatabaseID~TableOrViewName.FieldName* or *Index~DatabaseID~TableName.TagName*. For example for a field named CUSTOMERID in the CustomerInvoices view in MyDatabase.dbc (which has an ID of \_4WD11QLRS), KEY is Field~\_4WD11QLRS~customerinvoices.customerid.

### Running Project Explorer from Thor

If you install Project Explorer from the Thor Check for Updates function, it automatically creates a “tool” for Project Explorer and adds it to the Thor Tools, Applications menu. If you instead downloaded Project Explorer from its repository or cloned the repository, copy Thor\_Tool\_ProjectExplorer.prg from the ThorUpdater subdirectory of the Project Explorer folder to the Thor\Tools folder of your Thor folder, then edit that PRG and replace the folder specified for the FolderName property and the DO command with the correct location of Project Explorer on your system.

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.

You can also create a MyAddins subdirectory of the Addins subdirectory and put addins there. The benefit of that folder is that it isn't touched when an updated version of Project Explorer is installed. For example, if you enable or modify one of the stock addins that comes with Project Explorer, it's disabled and any changes lost when you install a new version of Project Explorer because the PRG is overwritten. Instead, copy the PRG to the MyAddins folder and modify the copy. Since it isn't overwritten when a new version is installed, it stays enabled or modified.

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:   toParameter1 - 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 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 = '*** specify method ***'
    toParameter1.Active = .T. && set to .F. to disable addin
    toParameter1.Name   = '*** specify descriptive name (optional) ***'
    toParameter1.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 3** 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 3.** The Project Explorer addin methods.

Addin	When Executed	Parameters	Success Flag
<b>Projects and solutions</b>			
<b>BeforeOpenSolution</b> <b>AfterOpenSolution</b>	When a solution is opened	ProjectExplorerSolution object, solution file name and path	
<b>BeforeGetProjectSettings</b>	When settings are retrieved from the	ProjectSettings object	

Addin	When Executed	Parameters	Success Flag
<b>AfterGetProjectSettings</b>	project: at startup, after a project is built, and when the project is reverted		
<b>BeforeLoadTreeView</b> <b>AfterLoadTreeView</b>	When the TreeView is loaded (e.g. when selecting a tag to display items for)	ProjectExplorerForm object	
<b>AfterCreateMenu</b>	When clicking the menu button	ProjectExplorerForm object, ProjectExplorerShortcutMenu object	
<b>AfterCreateShortcutMenu</b>	When right-clicking the TreeView	ProjectExplorerForm object, ProjectExplorerShortcutMenu object	
<b>OnActivate</b>	When the Project Explorer window gets focus	ProjectExplorerForm object	
<b>OnStartup</b>	When Project Explorer is started	ProjectExplorerForm object	
<b>BeforeSaveProjectSettings</b> <b>AfterSaveProjectSettings</b>	When you click Save for the project properties	ProjectSettings object	
<b>BeforeBuildProject</b> <b>AfterBuildProject</b>	When a project is built	Project object	Y
<b>BeforeAddProjectToSolution</b> <b>AfterAddProjectToSolution</b>	When a project is added to a solution	ProjectExplorerSolution object, path for PJX	Y
<b>BeforeRemoveProjectFromSolution</b> <b>AfterRemoveProjectFromSolution</b>	When a project is removed from a solution	ProjectExplorerSolution object, path for PJX	Y
<b>BeforeAddVersionControl</b> <b>AfterAddVersionControl</b>	When a solution is placed under version control	ProjectExplorerSolution object	Y
<b>BeforeChangeVersionControl</b> <b>AfterChangeVersionControl</b>	When the version control settings are changed	ProjectExplorerSolution object	Y
<b>BeforeSaveSolution</b>	When a solution is	ProjectExplorerSolution	

Addin	When Executed	Parameters	Success Flag
<b>AfterSaveSolution</b>	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	object	
<b>BeforeCleanupSolution</b> <b>AfterCleanupSolution</b>	When a solution is cleaned up	ProjectExplorerSolution object, .T. if object code is removed	
<b>OnExit</b>	When Project Explorer is closed	ProjectExplorerForm object	
<b>Item management</b>			
<b>BeforeModifyItem</b> <b>AfterModifyItem</b>	When an item is modified	ProjectItem object, ProjectExplorerForm object (AfterModifyItem only)	Y
<b>BeforeRunItem</b> <b>AfterRunItem</b>	When an item is run	ProjectItem object	Y
<b>BeforeRemoveItem</b> <b>AfterRemoveItem</b>	When an item is removed from the project	ProjectItem object	Y
<b>BeforeAddItem</b> <b>AfterAddItem</b>	When an item is added to the project	Project object, filename	
<b>BeforeNewItem</b> <b>AfterNewItem</b>	When an item is created	Project object, ProjectItem object	
<b>BeforeRenameItem</b> <b>AfterRenameItem</b>	When an item is renamed	ProjectItem object, new name	
<b>BeforeSaveProjectItem</b> <b>AfterSaveProjectItem</b>	When you click Save for an item's properties	ProjectItem object	Y
<b>GetDefaultMetaDataForItem</b>	When an item is loaded for the first time	ProjectItem object	
<b>Version control</b>			

Addin	When Executed	Parameters	Success Flag
<b>BeforeCreateRepository</b> <b>AfterCreateRepository</b>	When a repository is created	Folder for repository	
<b>BeforeAddFilesToVersionControl</b> <b>AfterAddFilesToVersionControl</b>	When files are added to version control	Array of file names	
<b>BeforeRemoveFilesFromVersionControl</b> <b>AfterRemoveFilesFromVersionControl</b>	When files are removed from version control	Array of file names	
<b>BeforeCommitFiles</b> <b>AfterCommitFiles</b>	When files are committed	Commit message, array of file names	
<b>BeforeRevertFiles</b> <b>AfterRevertFiles</b>	When files are reverted	Array of file names	
<b>BeforeCommitAllFiles</b> <b>AfterCommitAllFiles</b>	When all changes are committed	Commit message, array of project file names	
<b>BeforeRenameFileInVersionControl</b> <b>AfterRenameFileInVersionControl</b>	When a file is renamed	Original file name and path, new name (stem only)	
<b>BeforeRevisionHistory</b> <b>AfterRevisionHistory</b>	When revision history for a file is displayed	File name and path	
<b>BeforeVisualDiff</b> <b>AfterVisualDiff</b>	When visual diff for a file is displayed	File name and path	
<b>BeforeRepositoryBrowser</b> <b>AfterRepositoryBrowser</b>	When the repository browser is displayed	None	
<b>BeforeConvertBinaryToText</b> <b>AfterConvertBinaryToText</b>	When Convert Binary to Text is chosen	ProjectItem object, ProjectExplorerForm object	
<b>BeforeConvertTextToBinary</b> <b>AfterConvertTextToBinary</b>	When Convert Text to Binary is chosen	ProjectItem object, ProjectExplorerForm object	

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 3) 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 3 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 3 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 `AddWLCProjectBuilderButton.prg` written by Rick Borup, adds a button to Project Explorer's toolbar that, when clicked, opens the White Light Computing (WLC) Project Builder written by Rick Schummer. 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). The call to `SetToolbarControlPosition` positions it as the right-most control in the toolbar and adjusts `MinWidth` so the control is always visible when the form is collapsed.

**Listing 4.** This `OnStartup` addin adds a button to the toolbar to open the WLC Project Builder.

```
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 to open the White Light Computing (WLC) Project
* Builder dialog for the active project.
* Requires the WLC Project Hook class library cprojecthook5.vcx (download from
* http://whitelightcomputing.com/prodprojectbuilder.htm)
* To use: 1) Set toParameter1.Active = .T. (above)
*          2) Set the value of WLCProjectBuilderButton.cWLCProjectBuilderClass
*          (below)

loToolbar = toParameter1.oProjectToolbar
try
    loToolbar.AddObject('cmdWLCProjectBuilder', 'WLCProjectBuilderButton')
    loButton      = loToolbar.cmdWLCProjectBuilder
    loButton.Height      = loToolbar.cmdBack.Height
    loButton.Width       = 30
    loButton.Caption     = 'PB'
    loButton.ToolTipText = 'Open WLC Project Builder Dialog'
    loButton.Visible     = .T.
    toParameter1.SetToolbarControlLocation(loButton)
    l1OK = .T.
catch
    loToolbar.RemoveObject('cmdWLCProjectBuilder')
    l1OK = .F.
endtry
return l1OK
```

```
define class WLCProjectBuilderButton as CommandButton

* Set cWLCProjectBuilderClass to the path and file name of the WLC
* cprojecthook5.vcx class library (or your subclass library).

    cWLCProjectBuilderClass = '\Development\Tools\VFP\WLCProjectBuilder\' + ;
        'cProjectHook5.vcx'
    oWLCProjectBuilderToolbar = null

    function Init
        This.oWLCProjectBuilderToolbar = newobject('tbrProjectTools', ;
            This.cWLCProjectBuilderClass)
        return vartype( This.oWLCProjectBuilderToolbar) = 'O' and ;
            not isnull( This.oWLCProjectBuilderToolbar)
    endfunc

    function Click
        This.oWLCProjectBuilderToolbar.cmdProjectBuilder.Click()
    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 when the user clicks the menu button because it's an addin for the AfterCreateMenu method. As with AfterCreateShortcutMenu handlers, the second parameter is a reference to a ProjectExplorerShortcutMenu object. 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 AfterCreateMenu 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 = 'AfterCreateMenu'
    toParameter1.Active = .T.
    return
endif

* Add an item to the Project Explorer menu to run the main program for the project.

tuParameter2.AddMenuBar('R\<un Main Program', ;
    'loForm.RunItem(loForm.oProject.oProject.MainFile)', ;
    'empty(loForm.oProject.oProject.MainFile)', ;
    '', ;
    tuParameter2.nBarCount - 1)
return .T.
```

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.
```

The addin in **Listing 8**, taken from SetFont.prg, sets the font size when the Project Explorer is started.

**Listing 8.** This addin changes the font size for Project Explorer.

```
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

* Set the font to the desired size

toParameter1.SetAll('FontSize', 12)
```

## 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:

- 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.
- Currently only `FoxBin2PRG` is supported for binary to text conversion. A little work is needed to support other converters. Project Explorer could also use the value of `_SCCTEXT` as the location of the converter.
- Source control providers that integrate with VFP, such as `SourceSafe` and `SourceGear Vault`, aren't currently supported.
- 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.

I'd like to thank the beta testers for Project Explorer whose bug reports and enhancement requests made it much better: Paul Mrozowski, Matt Slay, Rick Borup, Eric Selje, Phil Sherwood, Mike Potjer, Matthew Olson, Hans-Peter Grözinger, and Lutz Scheffler.

## 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*. He wrote over 100 articles in 10 years for FoxRockX and FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe.

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 conference (<http://www.swfox.net>). He is one of the administrators for the VFPX VFP community extensions Web site (<http://vfp.org>). 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>).

