

Using the VFPX Code Analyst

Andrew MacNeill, AKSEL, 2007

The VFPX project encompasses a variety of tools for FoxPro developers. Some of these tools, such as the Status Bar or Outlook view (discussed in the past months) are designed for use in applications. However, there are other tools to enhance the Visual FoxPro Development Environment (IDE). Some improve on existing tools included by Microsoft whereas others are new (see the sidebar). In this article, I look at one of the VFPX tools that aim to make application development easier.

Code Analyst

In the April 2006 of FoxPro Advisor, I presented some guidelines for refactoring code. Since writing that article, however, I have seen many instances where refactoring is required yet the developer in question is at a loss of where to start. Enter the Code Analyst, a project that I started on VFPX but have since been joined by some other developers to enhance it further.

Code Analyst combines extensibility with a Code References-like interface, analysing pieces of code or entire projects and identifying areas for refactoring. Run ANALYST.APP and select a project, program, form or class library. When it is run the first time, it is also added to the Tools menu in the FoxPro IDE. A progress bar identifies which component is being analysed. When completed, a results window appears. (see figure 1)

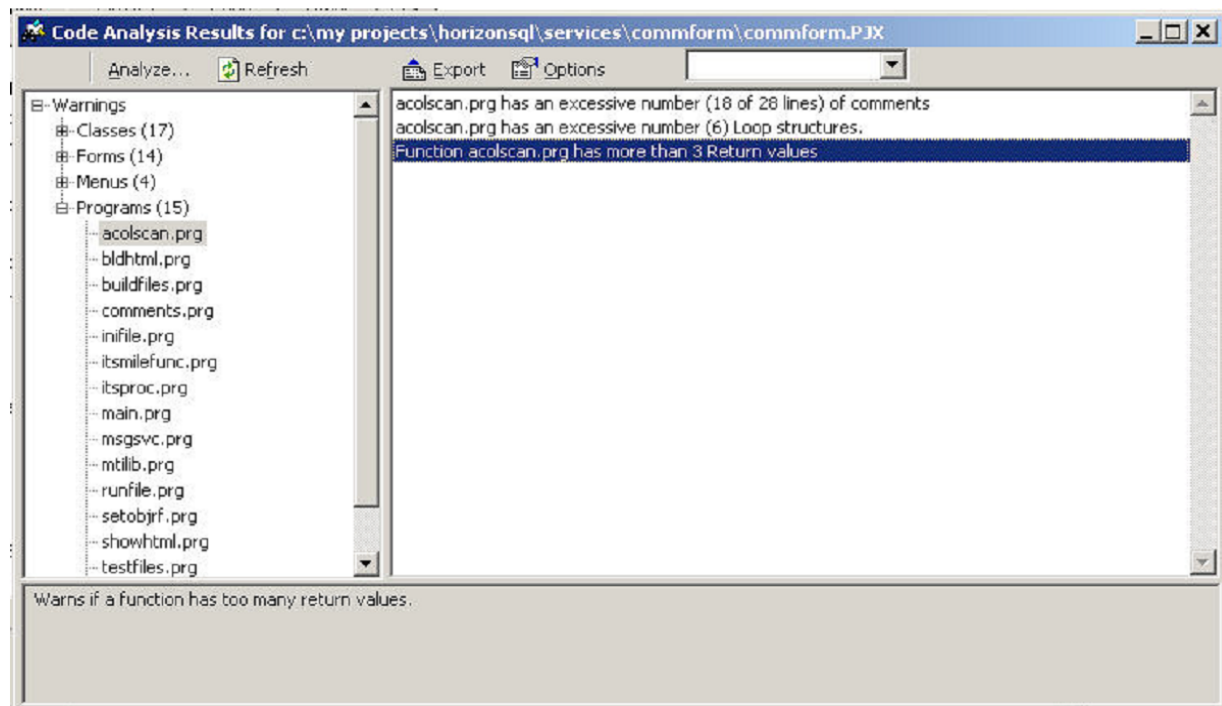


Figure 1 – Analyzing Your Code. The Code Analyst window lets you review individual pieces of code, highlighting areas ripe for refactoring.

When viewing a project, the results are broken into types of components. Select an individual component and the list on the right shows the results for that piece of code. Highlight a particular warning and an explanation as to why it was flagged appears in the panel at the bottom.

The combo box on the top shows a list of all alerts found (see figure 2). Select one and the Results window only displays those areas of code with that alert. For example, selecting "RETURN Within WITH" will show pieces of code where a RETURN command was found within a WITH...ENDWITH statement. Many developers use this approach in their code; however, it has been known to sometimes cause crashes and C5 errors.

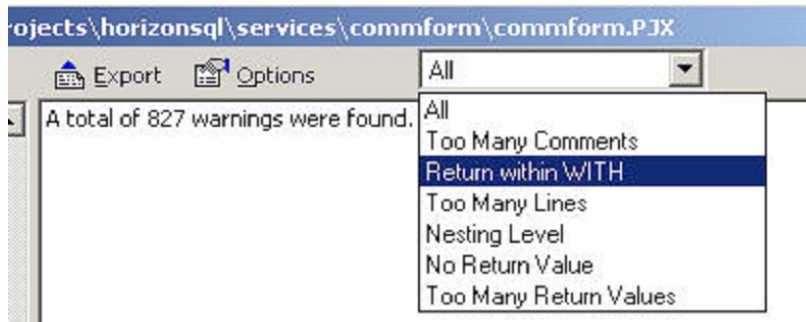


Figure 2 – Focus on a Problem. Use the Alerts combo to only look at particular refactoring alerts.

Right-click on a node on the tree on the left and choose Open to open the piece of code. Forms and programs may also be opened automatically by double-clicking the alert in question.

Once changes have been made, click Refresh and the analysis will be performed again. Alternately, click Analyze... and select another FoxPro code component to review.

Where Do The Rules Come From?

Click Options to review which rules are applied. The Code Analyst uses a table that may be found in the root FoxPro folder named CODERULE.DBF to store its rules. The Configuration window lets you review this table (see figure 3). The structure for this table is shown in table 1. Some rules may not apply to your coding standards or the program in question. For example, one of the default rules identifies code with more than 150 lines. This is an arbitrary number based on some recommendations found in programming books. Another rule expects that every method, function or procedure should have a RETURN value. If a particular doesn't apply, clear the Active checkbox.

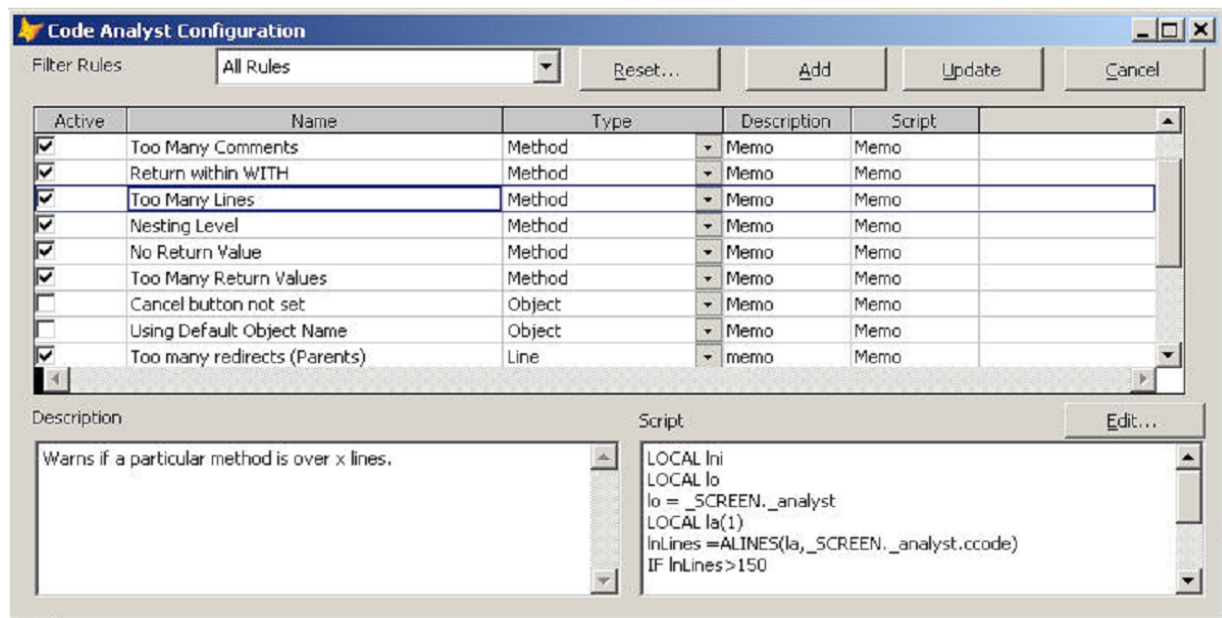


Figure 3 – Choose Your Rules. Since developers can have their own coding conventions, you can choose which rules you want to analyze.

Proactive Rules

There are also some rules that will change code for you. The Relabel Default Object Names rule looks at individual objects and if its name is the same name as its class, suggests a renaming of it. For example, if a command button's name is Command1, it renames it using its Caption property. If the object doesn't have a caption property or another object with the same name exists, it is reported as an alert.

Field Name	Type	Description
TYPE	Character (1)	Type of rule. Available options are: M – Method. F – File O – Object L – Line
NAME	Character (30)	The name of the rule being performed
ACTIVE	Logical	Identifies if a rule will be checked when reviewing code.
DESCRIPT	Memo	A description of what the rule does and how it may be corrected.
SCRIPT	Memo	The script that is applied to the piece of code being analyzed.
PROGRAM	Memo	Not used at this time
CLASSLIB	Character (30)	Not used at this time
CLASSNAME	Character (50)	Not used at this time
TIMESTAMP	Date/Time	The date/time that the rule was last updated.
UNIQUEID	Character (10)	A unique identifier to be used for sharing rules.

Table 1 – Refactoring Rules. With the CODERULE table, you can create your own rules to help identify common patterns in your code.

How Do Rules Work?

When the Code Analyst runs, it creates an `_Analyst` object to the `_SCREEN` object. While the Script field contains the logic for each rule, the `_Analyst` object contains the properties for each element of code being looked at (see table 2). Code is analyzed differently based on the type of rule.

Property	Description
CLINE	The line of code
NLINE	The line number
CFUNCNAME	The function or method name
OOBJECT	Pointer to the object (used for Object rules)
CCODE	The entire body of code
CFILENAME	The file name
LDISPLAYFORM	Logical property to indicate that the Code Analyst should display results.

Table 2 – Code Analyst properties. The

If a piece of code doesn't pass a particular rule, call the `AddWarning` method of the `_ANALYST` object with the message you want displayed. For example, here is the code to check for too many comments in a piece of code. This rule has a type of M – for Method so it will run against every method, function or procedure found. The full code is found in the `ccode` property.

It runs through every line of code and then if it finds more than a third of the lines of code have comments, it adds a warning. After all – while comments may be useful, too many comments may make code more complex.

```
LOCAL Ini
LOCAL lo
lo = _SCREEN._analyst
LOCAL la(1)
InLines = ALINES(la,_SCREEN._analyst.ccode)
LOCAL InComments
LOCAL InCode,InEmpty
STORE 0 TO InComments,InCode,InEmpty
LOCAL lcLine

FOR Ini = 1 TO InLines
  lcLine = la(Ini)
  IF EMPTY(lcLine)
    InEmpty = InEmpty + 1
  ENDIF

  IF ALLTRIM(STRTRAN(lcLine," ")= "*")
    InComments = InComments + 1
  ELSE
    InCode = InCode + 1
  ENDIF
ENDFOR
```

```
lo.aCode(ALEN(lo.aCode,1),3) = InCode
IF InComments>InCode/3
    lo.AddWarning(lo.aCode(ALEN(lo.aCode,1),1);
    +" has an excessive number - "+;
    LTRIM(STR(InComments)) +;
    " comments compared to "+;
    LTRIM(STR(InCode))+;
    " lines of code ")

    lo.LDisplayForm = .T.
ENDIF
```

At the end of the code, it sets the LDISPLAYFORM property to .T. to ensure the Code Analyst will display the results.

A Work in Progress

Like other VFPX projects, the Code Analyst is a work still in progress. It comes with 14 rules that can certainly highlight potential deficiencies in code but there are certainly many more ways to refactor code. The current version also doesn't have any reports that might be useful for code reviews, a valuable tool for development teams. These, among others, are features that have been discussed. Join the discussion by going to <http://www.codeplex.com> and searching for VFPX.