Tutorial

How to use Project Analyzer

© Copyright 1999 Aivosto Oy
vbshop@aivosto.com
http://www.aivosto.com/project/project.html

This document is version 1.1 and was written for Project Analyzer 5.1

1Introduction

This tutorial tells you how to use Project Analyzer for optimizing and documenting your Visual Basic programs.

1.1Executive summary. Why should we use Project Analyzer?

A tool for software engineers, Project Analyzer makes large software projects easier to understand. It documents commented and even uncommented code. It also contains numerous optimization and measurement features that help in fine-tuning and monitoring software development.

Project Analyzer quickly pays itself back in easier maintenance, less errors, improved software quality, and faster moving of projects from a programmer to another. It is most useful for analyzing middle to large sized programming projects. A lot of large and small companies around the world use Project Analyzer in some part of their software development cycle.

1.1.1What are the main features of Project Analyzer?

Project Analyzer is a tool that reads Visual Basic source code. The main features are related to code optimization, producing professional documentation, and measuring programming efforts in terms of output and quality.

Project Analyzer detects several types of programming problems related to optimization, style and functionality, like unused variables, dead code, missing variable types, name conflicts, objects with wrong scope, procedures missing code and buttons that do nothing. It also analyzes the design quality with industry standard software engineering metrics like complexity, understandability and reusability. In addition, Project Analyzer can document source code by creating source code manuals and print-outs with cross-reference information.

1.1.2What are some special professional characteristics of the software?

Project Analyzer uses a powerful source code analysis engine that supports lots of the professional features of VB code. Supported VB features include name shadowing, compiler constants (#Const, #If..#End if), ActiveX projects, .dll files, and binary extension files (.frx). Project Analyzer even supports a special syntax for fine-tuning the analysis called comment directives.

Project Analyzer reports several levels of problems and optimization suggestions. For example, it understands different levels of deadness definitions – including called-by-dead-only deadness for procedures, and assigned-only deadness for variables. (If these words don't ring a bell yet, read on.)

1.2How do we get more help?

Because this is a tutorial, all features are not included. For a detailed help file, see project.hlp.

If you find something missing from this tutorial, please email to vbshop@aivosto.com with your wish – it may be included in a later version. Your questions about the use of the program are always welcome. We feel that providing good support is essential for this kind of software.

1.3Before we start: How to use the reports

Project Analyzer produces lots of different reports. All the reports work in the same way: first you select the report type, then you make the report.

You select the report type using the menu command Options|Report to. Available types are:

1. Display – the default
2. Paper – to print your reports
3. File – available file types are:
   - Plain text file
   - Rich text format file (RTF)
   - RTF for Help File, which you can use with Microsoft's Help Compiler to turn your reports into normal Windows Help files.
   - Hypertext file (HTML)

2How to optimize your code

When you make large programs, it often happens that extra code is left in the project. This extra code can be unused subs and functions, old variables, unnecessary constants, even Types and Enums. Extra code takes up disk space, slows down the program and it also makes it harder to read.

Because this code is not needed, it is also called *dead code*. The opposite of dead code is *live code*. With Project Analyzer, you can find the dead code to remove.

Project Analyzer helps optimizing live code too. It reports questionable programming practices and code that could be enhanced by adding an As Type or ByVal definition, for example.

2.1Problem view

The key to optimizing VB programs with Project Analyzer is its Problem view feature. The view is visible at the bottom of the main window.

Nr	Problem	Module	Procedure	Type
214	Error handler missing	TestClass	Style% [Property Get]	Functionality
215	Uncommented code	TestClass	Style% [Property Get]	Style
216	Parameter with type character: vNewValue%	TestClass	Style [Property Let]	Style
217	Error handler missing	TestClass	Style [Property Let]	Functionality
218	Uncommented code	TestClass	Style [Property Let]	Style
219	ByVal/ByRef missing: vNewValue%	TestClass	Style [Property Let]	Style
220	Error handler missing	TestClass	MyActiveColor_Click	Functionality
221	Uncommented code	TestClass	MyActiveColor_Click	Style
222	Default control naming: Shape1	ActiveColor		Style
223	Option Explicit missing	ActiveColor		Style

Time: 00:13 Phase: 2/2

2.1.1Problem categories

Project Analyzer divides problems into 4 categories:

1. *Optimization*. These problems affect the speed and size of the resulting program negatively.
2. *Style*. These problems are related to the programming style. They don't necessarily lead to problems in the short run, but they often lead to worse understandability and errors in the future.
3. *Metrics* is a sub-category of Style. You can set target values for different metrics and monitor if some part of your program exceeds the limits. Read more about metrics in chapter 4.
4. *Functionality*. These problems affect the run-time functionality of the program. The reason is often that somebody forgot to do something.

2.1.2Filter configuration

You may be interested in only 1 or 2 problem categories. Or you may think that the use of Goto isn't that bad after all. That's why Project Analyzer lets you choose which problems to show and which ones to hide. You can create quick check filters and strict filters for giving your code a final polish. The predefined filters are:

Dead code	Check deadness only
Disable checking	Disable all problem checking
Functionality	Report problems affecting functionality only
Optimizations	Report problems related to optimization only
Strict	Report all problems
Style	Report problems related to style issues only

Press the button to configure filters.

2.1.3Removing dead code

Dead means unused. Unused files, procedures (i.e. subs, functions & properties), variables, constants, user-defined types and Enums... Sometimes dead code is required for a later phase of your development, but most of the time you can remove it without any problems.

Because there are different kinds of dead things, you are not supposed to remove all dead code mechanically.

Note: When you have made changes to your source code, remember to check if the modifications were OK. Don't save your modifications, but let VB first check if everything is OK. Pressing Ctrl+F5 in Visual Basic makes VB to check everything and try to run the program. If it works OK, you're safe to save your modifications.

Unused files. These are files that no other files refer to in the source code. Check if you really need these files and remove them from your project.

Dead procedures. These things usually make up most of the extra code. Some people have reported their project consisted of 30 – 40% of these things!

There are two kinds of dead procedures. The first ones are those that no other procedure calls. Let's call this the primary case. You can safely remove any single dead procedure.

Then there is the secondary case: There may be a group of procedures that only call other procedures in this group. However, no calls come from outside this group. So in effect, these procedures are dead! This kind of deadness is marked with a C in the Problem report. You need to remove all procedures in such a group at once, otherwise VB may complain about some missing procedures.

Dead variables and constants are usually left behind because you alter some routines and don't remember to remove the extra declarations. There are two types of dead variables: Those that are not used at all, and those that are used only in assignment. This means that they are given a value, but that value is not used anywhere in the program. Dead constants have only one type: those that are not used at all.

To remove a variable or constant that is not used at all, just delete its declaration (the Dim statement). To remove a variable that is used only in assignment, you need to find the statement where it is used and determine what to do with the statement. Usually, it's a function call like this:

```
x = MyFunction(y, z)
```

If you don't need the return value x and you're using VB version 4.0 or later, you can modify this statement to

```
MyFunction y, z
```

There is one thing you need to do to find dead variables. You need to use Option Explicit at the start of each module. This makes Visual Basic require variable declaration. Using this option is good programming practice and widely recommended by VB experts, because it forces you to declare all your variables and makes your code more readable and require less RAM space.

Dead Types and Enums are there because of similar reasons as dead variables and constants. You can remove a dead Type or Enum if you're sure you don't need it at a later time.

Procedures with no code. This is a special case of extra code. There may be a reason for an empty procedure, but most of the time it's just taking up space. Check if you really need this kind of a procedure and remove it if you don't.

2.1.4Further optimization with Problem report

Variables with no type. If you don't declare your variables as a specific type, VB uses the default data type Variant. A Variant can contain any kind of data: numbers, text, object references, tables, ... In some cases is great. But in most cases that's not required, and you know beforehand what kind of data a variable contains.

Why is a Variant bad? The first reason is because it takes far more memory space than other data types. You can save memory if you declare your variables as Integer, Single, String, etc. The second reason is that using specific data types removes run-time errors. You never know what a Variant contains, but an Integer always contains a number. This is especially important when declaring procedure parameters.

Here is another reason why you should use Option Explicit. If you don't declare your variables, they become Variants by default.

So to make your program require less memory and get rid of run-time errors, see the Problem report for Variables with no type. Give a specific type to all Variants you can, like this:

```
Dim x
=> Dim x As Integer
```

In the following example, x is really a Variant. That's very difficult to notice without Project Analyzer!

```
Dim x, y As Integer
=> Dim x As Integer, y As Integer
```

Functions with no type. Like variables, functions require a type definition too. If you don't define the type, a function returns a Variant value. Example:

```
Function Calculate(ByVal Value As Single)
=> Function Calculate(ByVal Value As Single) As Single
```

There is nothing like Option Explicit to require a type definition for functions. So the only way to check it is to use an analyzer.

2.1.5Style issues

You may feel that Project Analyzer often accuses you of using "wrong" programming style. Don't let it make you feel bad! There are a number of ways to do certain things, and if done consistently, no problems arise. However, you should pay attention to the style rules and make your decision of what rules to obey.

Scope declaration missing. VB's default scope rules are somewhat complicated and may lead to excess scope or a scope that is too limiting. Always give your procedures, variables etc. a scope. Don't just write

```
Sub Calc()
```

It is not clear if Calc may be called from outside of its own module. Instead, write either

```
Private Sub Calc()
```

or

```
Public Sub Calc()
```

A Private thing cannot be called from outside of its own module. A Public thing is part of the interface of the module and can be accessed from outside. In the case of Public classes and UserControls, Public things may even be called from other projects.

Excess scope. A part of your program is accessible from other modules because it was deliberately or accidentally made Public. However, other modules don't use it. Check if you should declare the part as Private instead of Public. It is good programming practice to use as tight a scope as possible to prevent other parts of the program calling and modifying parts that they shouldn't have anything to do with.

Global found, use Public. Earlier versions of VB didn't have a Public keyword. Later, Public and Private were introduced for defining the scope of things. Nowadays it is suggested that you use Public instead of Global. They are synonymous words, so no harm is done.

ByRef/ByVal missing. VB's default for procedure parameters isn't that useful. If you haven't declared a parameter either ByVal or ByRef, VB treats it as ByRef. Unfortunately, ByRef can lead to unexpected results if it's not used correctly.
The cure is simple. Always use ByVal when possible! Like this:

```
Sub DoThings(Text)
=> Sub DoThings(ByVal Text As String)
```

What exactly are ByVal and ByRef? ByVal tells VB to pass a parameter by value. This means, whatever changes you make to the value of that variable, the changes take effect only inside the (sub-)procedure.
If you use ByRef, the parameter is passed by reference. This means that not only the value, but the actual variable is passed to the procedure. When you change the value of the parameter, the change is reflected in the calling procedure too! This may lead to errors that are very hard to notice. Example:

```
Sub DoThings(Text As String) ' Text is ByRef by default
   Text = "Hello, world!"
End Sub

Sub Main()
   Dim Text As String
   Text = "Hello, fellow!"
   DoThings(Text)
   MsgBox Text ' Shows Hello, world!
End Sub
```

If you used ByVal, the result would differ:

```
Sub DoThings(ByVal Text As String)
   Text = "Hello, world!"
End Sub

Sub Main()
   Dim Text As String
```

```
        Text = "Hello, fellow!"
        DoThings(Text)
        MsgBox Text ' Shows Hello, fellow!
    End Sub
```

2.1.6Other problems

The Problem view lists lots of other problems too. You can see descriptions of these problems in project.hlp.

2.2Optimizing project groups with Super Project Analyzer

Super Project Analyzer is an add-in feature to Project Analyzer. It analyzes project groups that share some files. These project groups are also called super projects. Super Project Analyzer analyzes them. It reports which procedures, variables and constants, Types and Enums are shared between projects, or dead. This is quite similar to what Problem report does for a single project.

Why do you need a separate analyzer then? A procedure that's not used in one project may be used in another project. Super Project Analyzer takes the output of separate analyses and forms a general picture of what can be removed and what can not.

To use Super Project Analyzer, first analyze all the projects in the super project separately and save the results with the command Save data for Super Project Analyzer in the Add-Ins menu. After that, start Super Project Analyzer and load this data to see the ultimate results.

Project Graph is an optional add-in for Project Analyzer. See the help file for more information.

2.3Comment directives: Bypassing certain optimizations

Sometimes a procedure (or variable etc.) is dead, but you know you'll need it later. Project Analyzer has a special feature to ignore the deadness of given objects.

You do this by writing special comments in your code. These are called *comment directives*. They don't affect Visual Basic in any way, but they tell Project Analyzer to behave in non-standard ways. See topic Comment directives in project.hlp for the exact syntax.

3How to document your code

For many programmers, documentation is the last and the most boring job in a programming project. It often gets forgotten because it doesn't seem to add any value. But what if someone else has to continue with the undocumented code? Even the simplest documentation can save a lot of money in such cases.

Project Analyzer includes a lot of features, mostly reports, for documentation purposes. Although writing comments to code is essential for professional documentation, Project Analyzer can help even in the case of uncommented code.

3.1Listing procedures with comments

A commented listing of procedures is the most basic form of code documentation. Project Analyzer makes this listing by command Report|Procedure list. For each procedure, this command can list comments and cross-reference information. Example:

```
Function CheckName(Byval Person As String) As Boolean
' Checks if Person exists in the database
' Returns True if successful

Called by:                    This section is optional
- AddEmployee
- UpdateEmployee
Calls:                        This is optional too
- CheckDatabaseField
```

To fully use this feature, you need to provide comments in your code. All comments immediately before and after the procedure declaration line are included in the report. This means all comments until an empty line or normal code is found. Example:

```
' This sub was created by N.N.
Sub MySub (Byval x As Integer)
' This sub does the following:
' It takes the parameter x and ...

' This line is not shown any more
Form1.Print x + 5

End Sub
```

The cross-reference information, calling and called procedures, is useful when you need to see which procedures work together.

Comment manual is an enhanced version of listing procedures with comments. This is an option in the Project Printer add-in. It is described later in this document.

3.2Listing variables and constants

Procedures are not everything a project contains. Global and module-level variables and constants are equally important for documentation. To get a listing, open the Variables and constants window in the View menu.

By pressing the Report button, you get a listing of all the variables and constants that are on the screen. By checking the options, you can list global constants only, or all live variables and constants, for example.

3.3Project Printer

The most thorough documentation of a project is its source code. Project Printer is an add-in tool for Project Analyzer that makes documenting the whole source code easy. Project Printer is not just an ordinary text file printer. Optionally, it formats code for easier reading, generates a table of contents, and includes cross-reference information and various metrics too.

Besides reporting all source code, Project Printer can also create a *comment manual*, which means formatted documentation with comments in the code.

Project Printer is an optional add-in for Project Analyzer. See the help file for more information.

3.3.1Documenting all source code with Project Printer

There are two ways to use Project Printer that can be recommended. You can either print all the code on paper, or alternatively, put it all into an ordinary Windows Help file. The Help file option is something that can be used to store code in a format that is easy to browse and to give colleagues for learning.

Using Project Printer. What is special about Project Printer is that you first select the report type before you actually select all the other options. This is because the report type has an effect on the available options. For example, you cannot use syntax-enhanced reports with keyword bolding if you're putting it all into a plain text file.

3.3.2Creating a comment manual with Project Printer

Comment manual is a feature of the Project Printer add-in. It creates a manual-like report based on comments in your source code. It does not print all the code, just the contents of the comments. This functions is an enhanced version of listing procedures with comments. What is different is that you can use a special comment syntax to get a nice-looking, formatted manual.

Comment manual takes all comments immediately before and after your Sub/Function/Property declaration line and makes a report out of them. All comments before the Sub/Function/Property line are taken, as well as all comments after it up to an empty line or a normal code line. This latter rule holds for comments at the start of the (declarations) section too.

For the exact syntax, please see topic *Comment manual* in project.hlp.

3.4Documenting cross-references with call trees

A program is not just a collection of procedures, it's very much calls from a procedure to another. These calls are cross-references. Calls from a procedure to another, from there to the next one form call trees.

There are two kinds of call trees: module call trees and procedure call trees. A *module call tree* tells which other modules a module needs. A *procedure call tree* is more exact: it tells which procedures are called by a given procedure.

3.4.1Ordinary call trees

Here is an example of a call tree for a function called RegExpr:

```
- RegExpr
    RaiseError
    RemoveMatches
  - ProcessParentheses
      GetParentheses
    + CreateMatch
```

```
            RaiseError
```

RegExpr calls RaiseError, RemoveMatches, and ProcessParentheses. ProcessParentheses calls GetParentheses, CreateMatch, and RaiseError. CreateMatch calls other procedures, but they are not shown now (+ denotes a branch that could be expanded).

You can get call trees like this by using the Call tree feature. You can find this in the View menu. Use the Report button to get a report of what you see on the screen.

Call trees can get very large and practically useless. Therefore it is recommended that you document only those call trees that are of special importance. You can always reconstruct call trees by running Project Analyzer again.

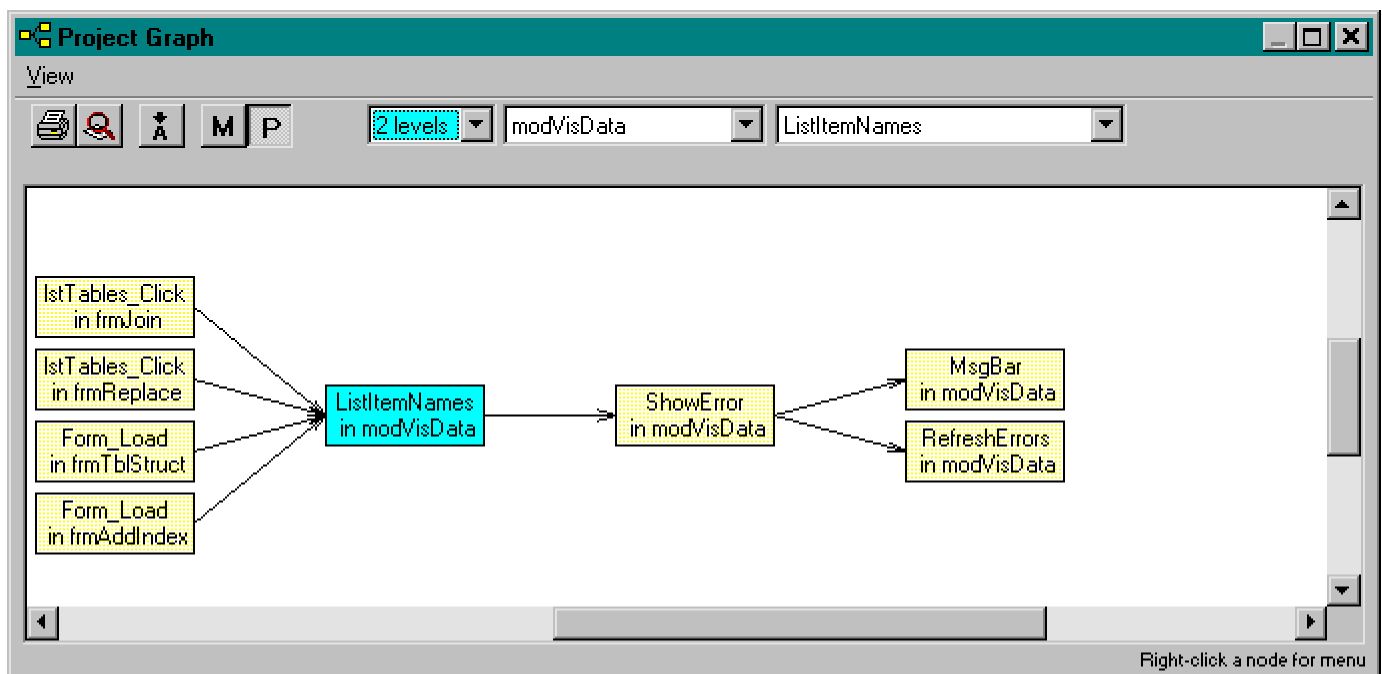
3.4.2Need report (closure of a call tree)

When you take all the branches and leaves in a call tree, you get a *need report*, available in the Report menu. Need report tells you all the parts in a program that procedure A requires to work. It lists all required procedures, variables, constants, user-defined types and Enums.

You can take a need report for one procedure, or a group of procedures. If you take it for a group of procedures, you simply get a list of everything that the group needs. This is useful if you want to copy certain routines from a project to another.

3.4.3Graphical call trees (Project Graph)

The most intuitive call trees are graphical. Project Graph is an add-in feature of Project Analyzer that can produce graphical call trees.



Project Graph generates both ordinary, or forward call trees, and backward call trees in the same picture. Forward call trees tell what the computer must do to complete a procedure. Backward call trees are the opposite: they tell which procedures need a given procedure.

Because graphical call trees easily get huge, the size of a graph is automatically limited to max 3 levels forwards and max 3 levels backwards. You can get more levels by right-clicking a node and selecting Expand.

To print a graphical call tree, use the Printer button. This prints a page with the graph that is currently shown on the screen.

Project Graph is an optional add-in for Project Analyzer. See the help file for more information.

4 How to measure your code

To monitor their programming efforts, software engineers often use some simple metric, like lines of code or EXE size. These are the most basic metrics, although they are not very sophisticated. You use them just fine, but Project Analyzer knows more. It can tell you about the *understandability*, *complexity*, and *reusability* of your code.

How to get metrics? You can see metrics in many different ways:
- The View|Metrics window is good to quickly point out complex procedures and modules.
- The Summary and Design quality reports are good to see the overall performance.
- VB Browser shows metrics for each module and procedure.

4.1 The different groups of metrics

4.1.1 Understandability

Bad understandability will most probably result in more errors and maintaining problems. You can estimate the understandability of your program with the following metrics:

Lines of code / procedure (View|Metrics, Design quality report)
Procedures longer than 50 lines are often too long. An alarm limit could be 100 lines.

Comment to code ratio and whitespace to code ratio (Design quality report)
The more comments in your code, the easier it is to read - and understand. Whitespace is also important for legibility.

Length of identifiers (Design quality report)
The longer your variable, procedure etc. names are, the more likely they are to be descriptive.

Cyclomatic complexity (View|Metrics, Design quality report)
Target for less than 10. See below.

Depth of conditional nesting (see View|Metrics)
Target for less than 5. See below.

4.1.2 Complexity

There are many kinds of software complexity:

- Structural complexity relates comes from conditionals and loops, that is, the flowgraph of the program.
- Psychological complexity means how difficult it is to understand a program. This is very much related to structural complexity.
- Informational complexity is about how much data comes into a procedure and how much goes out.
- Mathematical/computational complexity is about how much time and memory it takes to execute an algorithm.

High complexity may result in bad understandability and more errors. Complex procedures also need more time to develop and test.

The best measures for estimating the complexity of a procedure with Project Analyzer are:
- Lines of code
- Cyclomatic complexity
- Informational complexity
- Structural fan-out

These measures deal mostly with structural and informational complexity. See below for more information.

4.1.3 Reusability

Reusability is a magic word in programming. Measures for reusability are
- Reuse ratios reported by Super Project Analyzer (reuse in a group of projects)
- Structural fan-in. If it's high, the procedure/module is called (reused!) many times.
- Number of class modules

4.2 What do the metrics actually mean?

4.2.1 Lines of code

As simple as it may seem, lines of code is quite a good measure of how complex a program is. Project Analyzer calculates lines of code as follows:

Lines of code = Total lines - Commented lines - Empty lines

Lines of code does not include control declarations in a .frm file.

4.2.2 Cyclomatic complexity (McCabe)

Cyclomatic complexity is a measure of the structural complexity of a procedure. The higher the number, the more complex the procedure, and the harder it is to maintain it. Cyclomatic complexity for VB procedures is calculated as follows:

Cyclomatic complexity = Number of Branches + 1

Branches are caused by IF, SELECT CASE, DO...LOOP and WHILE...WEND statements.

"Normal" values for cyclomatic complexity range from 1 (very simple) to 9 (moderately complex). If cyclomatic complexity is more than 10, you may want to split the procedure. If it's over 20 you can consider it alarming.

Cyclomatic complexity is the minimum number of test cases you must have to execute every statement in your procedure. This is important information for testing. Carefully test procedures with the highest cyclomatic complexity values.

An average cyclomatic complexity, as well as the distribution of complexity, is reported on the Design quality report.

Note: Cyclomatic complexity often gives quite high values for procedures with long SELECT CASE statements.

4.2.3 Nested conditionals

Nested conditionals metric is related to cyclomatic complexity. Whereas cyclomatic complexity deals with the absolute number of branches, nested conditionals is only interested in how deeply nested these branches are.

If you have a procedure with deeply nested conditionals, you should consider splitting it up. Those procedures can be very hard to understand, and they are quite error-prone too.

4.2.4Nested loops

Nested loops is a very rough estimate of the mathematical complexity of a procedure. The more nested loops there are in a procedure, the more likely it is that those loops take up a significant amout of time to execute.

4.2.5Structural fan-in/fan-out (Constantine & Yourdon)

For procedures:

Structural fan-in = **the number of procedures that use this procedure**

Structural fan-out = **the number of procedures this procedure calls**

For modules:

Structural fan-in = **the number of modules that use variables, constants, or procedures in this module**

Structural fan-out = **the number of modules whose variables, constants, or procedures this module needs**

A high structural fan-in denotes reusable code.

The higher the structural fan-out, the more dependent that procedure is on other procedures, and more complex too.

4.2.6Informational fan-in/fan-out and informational complexity (Henry & Kafura)

Lines of code, cyclomatic complexity, or structural fan-out are not perfect in predicting the "real" complexity of a procedure. For example, a procedure may access a number of global variables and be very complex without having to call many other procedures.

Informational fan-in = **Procedures called + parameters referenced + global variables referenced**

Informational fan-in estimates the information a procedure reads

Informational fan-out = **Procedures that call this procedure + [ByRef] parameters assigned to + global variables assigned to**

Informational fan-out estimates the information a procedure returns

Combined, these give a new metric: informational fan-in x fan-out. This is reportedly good in predicting the effort needed for implementing a procedure, but not so good in predicting complexity. To predict complexity, we need a new metric: informational complexity. It is calculated as follows:

Informational complexity = **lines of code x (informational fan-in x informational fan-out)**

4.3 What are the target values?

There are no widely accepted target values for most of the metrics. You will have to make up your own goals. Take the Design quality report for some of your projects and decide the good and bad values yourself.

4.4 Literature

See your library for software engineering related books. They usually have a chapter or two about metrics too. Some books you could check:

Tom Manns and Michael Coleman: Software Quality Assurance.
Carlo Ghezzi, Mehdi Jaszayeri, Dino Mandrioli: Fundamentals of Software Engineering
Roger S. Pressman: Software Engineering

If you are really interested in structural complexity measures, there is a book that makes a thorough mathematical examination of 98 proposed measures for structural intra-modular complexity. This is for the advanced reader only.

Horst Zuse (1991) Software Complexity. Measures and Methods. Walter de Gruyter. Berlin - New York.

4.5 Systematic use of metrics

To follow how your projects develop, take a Summary report and a Design quality report often. The Summary report tells how much has been done, and the Design quality report tells how well it was done. If the comment to code ratio, for example, goes below a threshold level you've set, you know your project isn't well documented and that more comments have to be written for the newly created code.

When you have come up with some good target values for some metrics, you can set them as a limit for the Problem view. Go to Options|Problem options and define a filter with your own set of values. This way, you get notified when a procedure or module exceeds a target. You can also check metrics values by taking reports from the View|Metrics window.

You should keep the number of overly complex procedures to a minimum. Otherwise these procedures may get out of hand. It is easier to control these procedures when they are created. If you try to control them afterwards, it gets harder and harder when you forget what they contained.

5Other features

The basic features of Project Analyzer are related to analysis, optimization, and documentation as described in the preceding chapters. This chapter describes other features shortly. You are asked to see project.hlp for more information on these features.

VB Browser is in the main window of Project Analyzer. You can browse your code and jump from procedure to procedure using hyperlinks.

Archive project files command, in the File menu, can be used to put all files contained in a project to an archive file. It is designed to work with pkzip or arj. You can also get a text file listing all files in a project using this feature.

File and procedure details reports are similar to the File list and Procedure list reports, but only contain one file or procedure.

Call tree reports in the Report menu are a substitution to using the Call tree window in the View menu. Use of the Call tree window is recommended instead of the reports, because the reports can get unexpectly large – especially the All procedures call tree report.

Cross-reference report lists all calls to and from procedures, references and assignments to variables, constant references, and Type and Enum references.

Call depth report reports number of nested procedure calls. In other words, this is the same as the depth of a forward call tree, procedure by procedure. It also reports dead code.

Name shadowing report finds objects that have exactly the same name and a name clash thus occurs. This happens when the same name exists in both two scopes – for example, when *AskName* is both a form name and a procedure-level variable. Name shadowing may lead to programming errors that are hard to detect, but is not dangerous otherwise.

Library report lists all DLLs, OCXs and other libraries referenced by your project. This list does not include all files required by your program (when distributed), but only those that are referenced in the code and in the .vbp file.

Save analysis is a feature that saves the analysis results in a file that you can load the next time you use Project Analyzer. Loading from a saved analysis is faster than re-analyzing the whole project again.

Table of contents

1. Introduction
1.1 Executive summary. Why should we use Project Analyzer?..
1.1.1 What are the main features of Project Analyzer?..
1.1.2 What are some special professional characteristics of the software?................................
1.2 How do we get more help?...
1.3 Before we start: How to use the reports...

2. How to optimize your code
2.1 Problem view..
2.1.1 Problem categories...
2.1.2 Filter configuration..
2.1.3 Removing dead code..
2.1.4 Further optimization with Problem report...
2.1.5 Style issues...
2.1.6 Other problems...
2.2 Optimizing project groups with Super Project Analyzer..
2.3 Comment directives: Bypassing certain optimizations...

3. How to document your code
3.1 Listing procedures with comments...
3.2 Listing variables and constants...
3.3 Project Printer...
3.3.1 Documenting all source code with Project Printer...
3.3.2 Creating a comment manual with Project Printer..
3.4 Documenting cross-references with call trees...
3.4.1 Ordinary call trees..
3.4.2 Need report (closure of a call tree)...
3.4.3 Graphical call trees (Project Graph)...

4. How to measure your code
4.1 The different groups of metrics...
4.1.1 Understandability..
4.1.2 Complexity...
4.1.3 Reusability..
4.2 What do the metrics actually mean?..
4.2.1 Lines of code...
4.2.2 Cyclomatic complexity (McCabe)..
4.2.3 Nested conditionals...
4.2.4 Nested loops..
4.2.5 Structural fan-in/fan-out (Constantine & Yourdon)..
4.2.6 Informational fan-in/fan-out and informational complexity (Henry & Kafura)...............
4.3 What are the target values?..
4.4 Literature...
4.5 Systematic use of metrics..

5. Other features