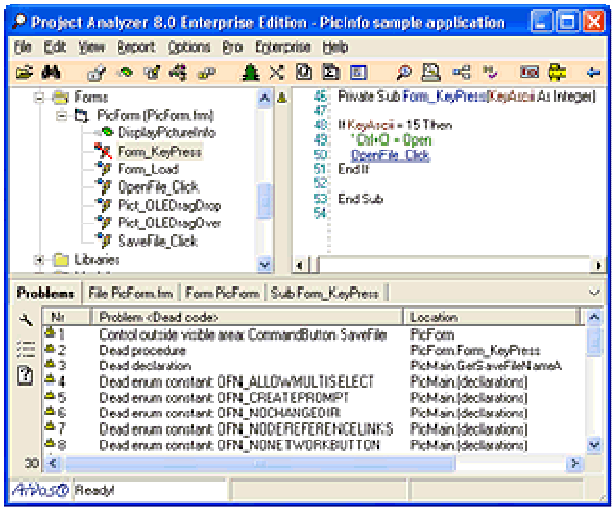
Tutorial

How to Use Project Analyzer v8.0



Aivosto

www.aivosto.com

URN:NBN:fi-fe20061421

Save the forests! If you print this document, print 2 pages per sheet, on both sides of the paper. Or just read on the screen.

1 Introduction

Aivosto Project Analyzer is a professional source code optimization and documentation software tool for Visual Basic developers. This tutorial leads you into the world of advanced code analysis. It assumes knowledge of the Visual Basic language, but no knowledge of code analysis.

Project Analyzer v8.0 works with code written with Visual Basic versions 3.0, 4.0, 5.0, 6.0, VB.NET 2002, 2003 and 2005. It also works with the VB.NET code in ASP.NET projects. Analysis of Office VBA projects is possible with VBA Plug.

Getting more help. This is a tutorial, not a comprehensive manual. Project Analyzer supports more features than what is described in this document. The help file *project.chm* is the comprehensive documentation and manual. The help is also available online at *http://www.aivosto.com/project/help* . Project Analyzer comes with free technical support, so feel free to ask!

For a Finnish tutorial, see *http://www.aivosto.com/project/tutorial-fi.pdf*

1.1 Executive summary

Why should we use Project Analyzer?

Writing high-quality, fully documented software is a goal that all developers share. The path to achieve that goal is paved with code reviews, tests, fine-tuning and document writing. Project Analyzer is a program that makes these tasks easier, saving effort and making it possible to achieve higher quality in less time.

What are the main benefits?

Project Analyzer makes a full code review. It suggests and performs numerous code improvements, leading to faster and smaller programs and enforcing adherence to programming standards. The improvements not only affect the internal technical qualities of code, but also add solidity and performance to the program itself.

Project Analyzer generates technical documentation by reading program source code. The available documents include graphical representations of program structure, commented source code listings and various reports such as file dependencies. Automatic document generation relieves the programmers from the burden of keeping technical documentation in sync with the existing code.

Project Analyzer helps programmers to understand existing code in less time. By browsing code in hypertext form, a programmer can quickly understand how a certain function operates with other functions and variables.

Who should use Project Analyzer?

Project Analyzer is most useful with middle to large sized projects and solutions. The users include programmers, testers and document writers. A basic understanding of the Visual Basic language is preferential.

What are the available editions?

Project Analyzer Standard Edition includes full source code analysis capabilities. It can analyze projects of any size, one project at a time.

Project Analyzer Pro Edition includes Standard Edition and all the following 4 professional features: Super Project Analyzer, Project Printer, Project Graph and Project NameCheck.

Project Analyzer Enterprise Edition adds automation to the analysis and correction of coding problems. It is meant for power users and development teams with large projects. It includes the Pro Edition and additional features. The Enterprise Edition supports multi-project analysis, that is, the analysis of several projects together. It also includes features that help with migrating existing code from earlier Visual Basic versions to Visual Basic .NET, Enterprise Diagrams for visualizing the structure of a program, advanced metrics to evaluate the quality of code, COM and DLL file analysis and search for duplicate code blocks.

VBA Plug

VBA Plug enables Project Analyzer to read Office VBA code. It retrieves VBA code from Office files and exports it in a format that Project Analyzer can read. VBA Plug is compatible with all editions of Project Analyzer. It requires a separate purchase.

2 Getting started

First a short note on the terminology. We use the term VB Classic to refer to VB versions 3.0 to 6.0. With .NET we mean all the supported VB.NET versions. In most cases, Office VBA is treated as if it were VB 6.0.

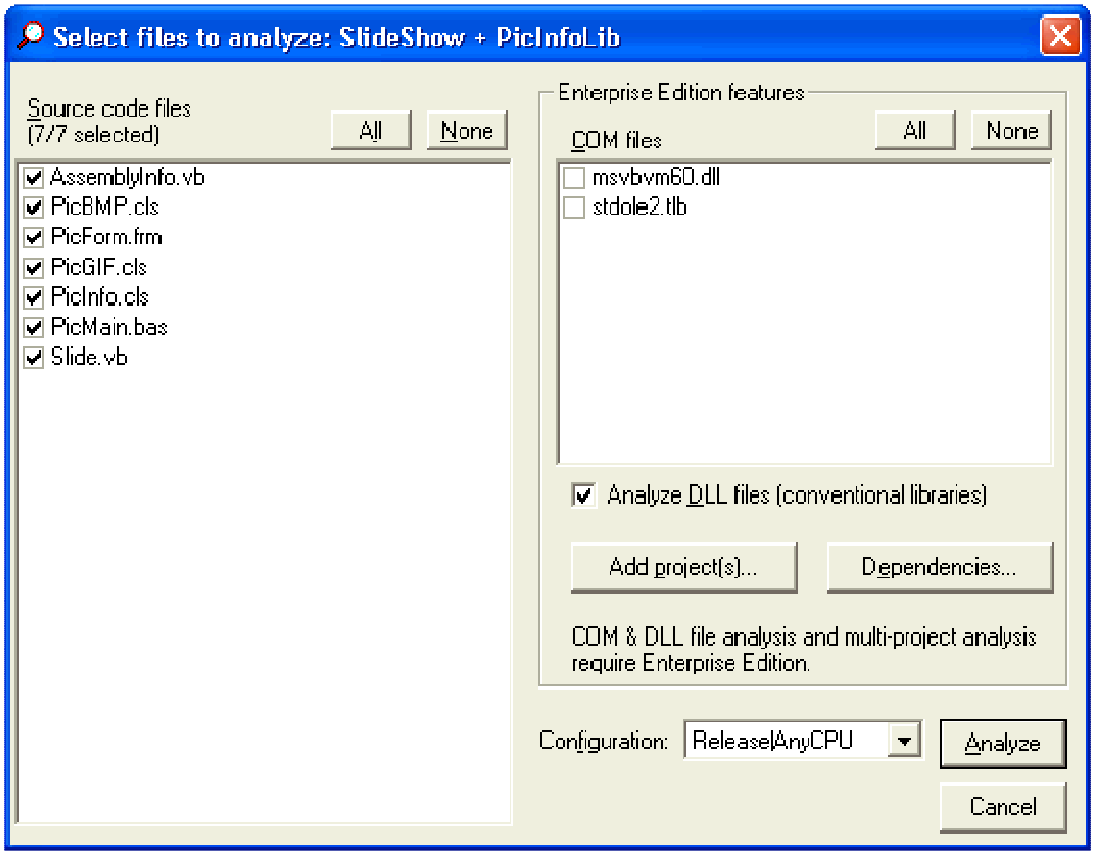
A project usually means a single .mak, .vbp or .vbproj project, although it may be also used in a more general way to cover all the analyzed code, whether it's from one or several projects.

2.1 Starting an analysis

Run Project Analyzer by double-clicking *project.exe*. Choose Analyze in the File menu. A dialog box opens up. Select a Visual Basic project for analysis. Any .mak, .vbp or .vbproj file will do. *If you are about to analyze an Office VBA project, please refer to the following chapter and then continue here.*

You can also analyze several projects simultaneously by selecting a .vbg (project group) or a .sln (solution) file. Notice that this feature, called multi-project analysis, requires the Enterprise Edition. The Standard and Pro Editions analyze only one project at a time.

This is the dialog that appears next:



In this dialog you can select the files to analyze. If you are running a demo, you can select a maximum of 10 source files at a time.

Notice that the features in the top right corner require the Enterprise Edition. They are also enabled in the demo and Standard/Pro Edition when you select a maximum of 10 source files to analyze.

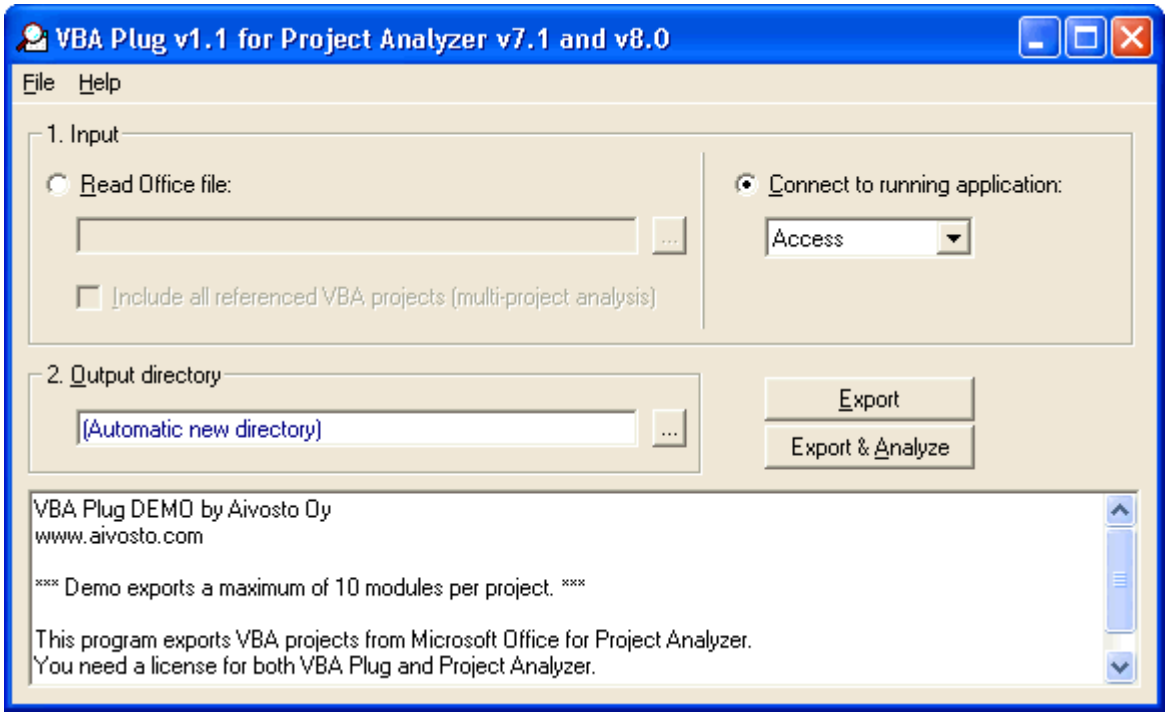
When you're done with the options, click Analyze to begin. The analysis consists of two phases. You can see the progress in the main window.

2.2 Analyzing Office VBA code

To analyze a Office VBA code, Project Analyzer requires an additional program called VBA Plug. This program is in your Project Analyzer directory in the file *vbaplug.exe*. VBA Plug requires a separate purchase. It supports the following Office applications: Access, Excel, PowerPoint, Visio and Word. The appropriate application needs to be

installed for VBA Plug to run. Please refer to the Project Analyzer help file for the system requirements and supported application versions.

Start VBA Plug by double-clicking *vbaplug.exe*.



There are two ways to select the VBA input source:

1. Read Office file. Click the button "…" to pick the Office file to analyze. By default, VBA Plug retrieves the VBA code in this file only. If you select "Include all referenced VBA projects", it will also retrieve any other VBA code referenced by the file. This is useful in a multi-document system where a VBA project may call other documents or templates.

2. Connect to running application. In this approach, you open the Office file in the appropriate Office application first. After doing this, return to VBA Plug and select the running application in the list.

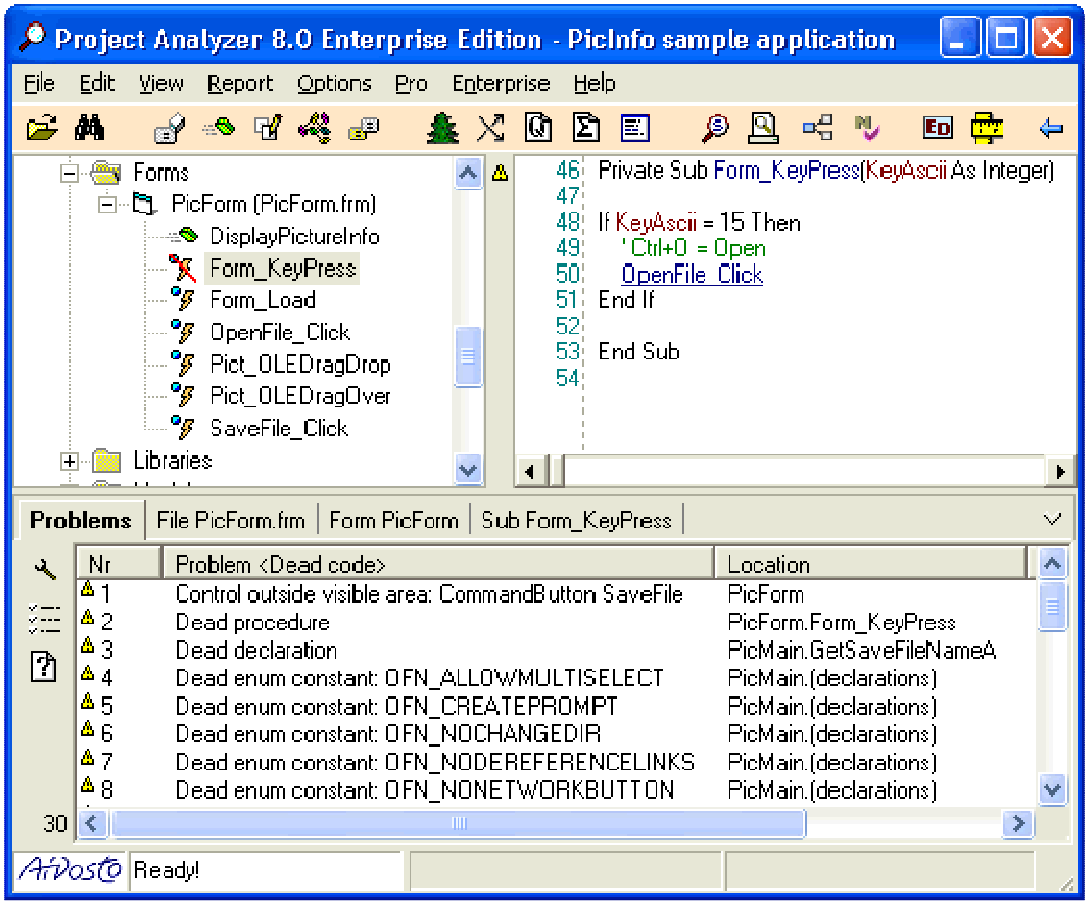
After selecting the input source, define the output directory in which VBA Plug will export the retrieved VBA code. The output directory must be an empty directory. You may leave this undefined, in which case VBA Plug will automatically create one for you. The automatically created directory will appear under the system's Temp directory by default. You may also define an alternative location via the File menu.

Now that you've defined the input and output settings, click Export & Analyze. VBA Plug will connect to Office, retrieve all VBA code and export it into files. The log text will tell you exactly what happens. If the export was successful, VBA Plug runs Project Analyzer to open the newly exported project(s). The process will continue as described in the previous chapter Starting an analysis.

If you see an error message in the log, read the Troubleshooting section in the help file. You can access this section via the Help menu. In many cases, trying the above mentioned "Connect" approach will help to avoid errors.

Press the Export button if you only want to export the code without running Project Analyzer.

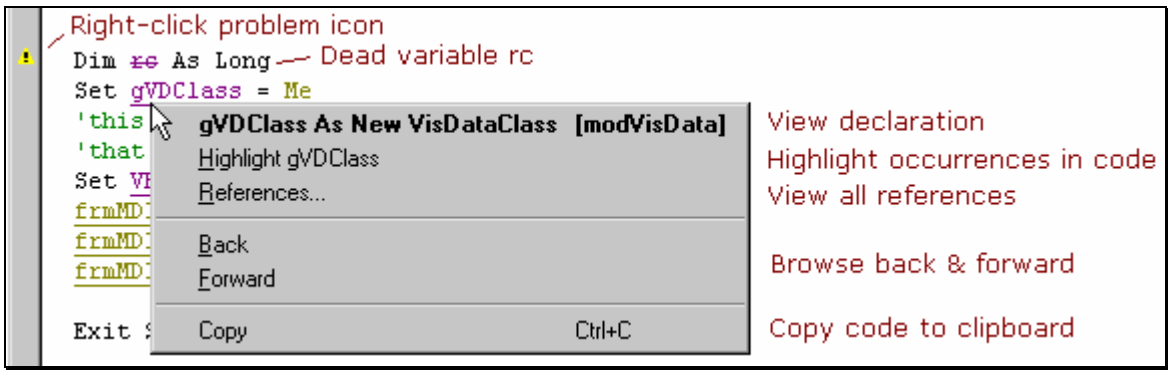
2.3 Main window of Project Analyzer



Hypertext view

Surf your code in the hypertext panel. Your code is automatically color-coded and hyperlinked by Project Analyzer. Use the mouse to left-click and right-click the highlighted words and icons:

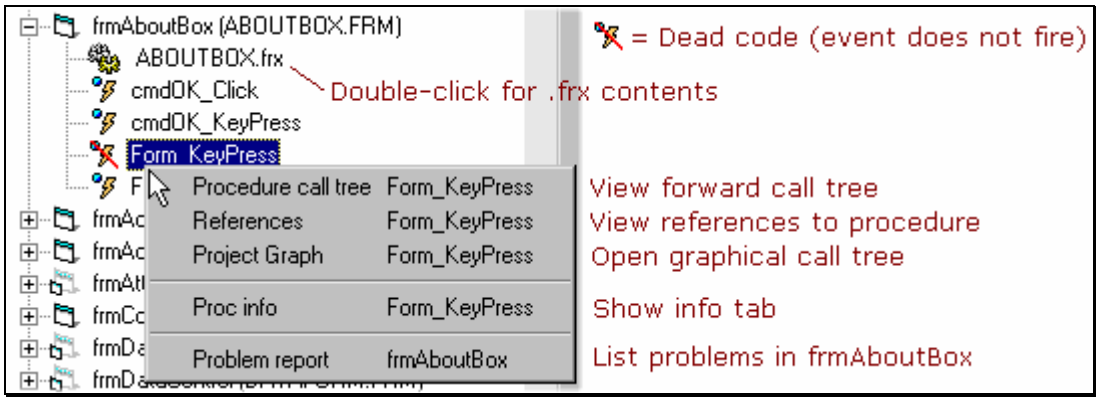
- Left-click a link. You are taken to the declaration of the clicked variable, function, enum, class etc.
- Right-click a link. A popup menu shows up. See the following picture.
- Left-click a problem icon (in the left margin). The problem view at the bottom shows the problem(s) on the line.
- Right-click a problem icon. A popup menu opens up listing the problem(s) on the line.



Right-clicking a link in the hypertext view

Project tree

Use the tree view on the left to move in your code. Use right-click to display a menu of options. You can also use double-click to view the contents of a picture or a .frx file.



Right-clicking an item in the project tree

Legend

~~Overstrike~~ indicates dead code.

A red line over an icon indicates dead code.

A violet X indicates exposed code with an unknown deadness status. The code was not found to be in use. However, the code is exposed to other projects and it may be in use by an unanalyzed project.

These problem icons indicate a problem found in code review. Left-click or right-click the icon for more information.

You can open a full legend of all the icons in the Help menu.

2.4 Using the reports

Project Analyzer produces a large number of reports. A large number of the are present in the Report menu, but there are also numerous Report buttons in the various windows to report the current window contents.

All the reports work the same way. First you select the report type, then you take the report. You select the report type in the toolbar in the main window. The available report types are listed here.

1. Display – the default. This also acts as a preview, from where you can print the report, save it to an rtf/pdf file or copy it to the clipboard.
2. Printer – to print your reports without previewing.
3. File – the available file types are:
   - Plain text file (txt)
   - Comma separated file (csv)
   - Rich text format file (rtf)
   - Portable document file (pdf)
   - Hypertext file (html)
   - Web archive (mht)

3 How to optimize your code

Project Analyzer makes your code shine – but it needs your help. With Project Analyzer you can enforce preferable programming practices, remove dead code, make the program behave in a more robust way and make changes to achieve better compatibility between Visual Basic versions.

It often happens that extra code is left in a project. The extra may be sub-programs and functions that are no longer called, events that don't fire, 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*. With Project Analyzer, you can find dead code and possibly remove it. This should be the first task you do to optimize your code. Only after that you should focus your efforts on what is left. That is the *live code*.

3.1 Code review (problem detection)

The key to optimizing VB programs with Project Analyzer is its code review feature. You can find the Problem view at the bottom of the main window.

Nr	Problem <Default>	Location	Type	Comment
1	Form missing Icon	PicForm	Funct	
2	Control outside visible area: CommandButton SaveFile	PicForm	Dead	Too far right
3	Dead procedure	PicForm.Form_KeyPress	Dead	KeyPreview property is False
4	Dead declaration	PicMain.GetSaveFileNameA	Dead	
5	Too many uncommented lines: 24	PicMain	Style	
6	Dead enum constant: OFN_ALLOWMULTISELECT	PicMain.(declarations)	Dead	
7	Dead enum constant: OFN_CREATEPROMPT	PicMain.(declarations)	Dead	
8	Dead enum constant: OFN_NOCHANGEDIR	PicMain.(declarations)	Dead	
9	Dead enum constant: OFN_NODEREFERENCELINKS	PicMain.(declarations)	Dead	

Tip: Right-click in the problem list to learn more about the problems and to produce reports.

Problem categories

Project Analyzer divides problems into 5 categories:

1. Optimization problems affect the speed and size of the resulting program negatively.

2. Style problems are related to the coding style. They don't necessarily lead to problems in the short run, but they often lead to worse understandability and maintenance errors in the future. – *Metrics* is a sub-category of Style, *available in the Enterprise Edition*. You can set target values for different metrics and monitor if some part of your program exceeds the limits.

3. Functionality problems affect the behavior of the program at run-time.

4. VB.NET problems represent migration issues. They affect you if you plan to upgrade your classic VB code to Visual Basic .NET. *These problems are available only in the Enterprise Edition.*

5. Internal problems indicate problematic conditions in the analysis. For example, a file may be missing.

Filter 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 described in the following table.

<Dead code>	Check deadness only.
<Default>	A default set of rules to start with.
<Hide all problems>	Disable all problem checking.
<Functionality>	Report problems affecting functionality.
<Optimizations>	Report problems related to optimization.
<Project NameCheck>	Report naming standards related problems. *Requires Pro or Enterprise Edition.*
<Strict - Show all problems>	Report all problems.
<Style>	Report problems related to style issues only. Includes Metrics and Project NameCheck.
<VB.NET Compatibility>	Report all VB.NET problems. *Requires Enterprise Edition.*
<VB.NET Fix before upgrade>	Report all issues that should be fixed before loading the project in VB.NET. *Requires Enterprise Edition.*

Press the button  to configure filters.

Sharing the filters

You can distribute the problem filters with Export/Import buttons. Export a filter to a file and import it on another computer. That's especially useful to enforce programming standards among teams.

3.2 Removing dead code

Dead means unused: unused files, subs, functions, properties, variables, constants, user-defined types, Enums, classes, modules and interfaces. Dead code is extra. It makes your files larger, your programs slower and it makes it harder for people to read and understand your code. You can remove the dead code without affecting the functionality of your program. Of course, sometimes you may decide to keep some parts for later use.

Project Analyzer Standard and Pro Editions detect dead code, but leave the removal to you. You will need to decide for each piece of dead code if you wish to keep or delete it. – Note: When you have made changes to your source code, remember to check that it still runs. Don't save yet. First, press Ctrl+F5 to make Visual Basic compile your code. If it runs, you're safe to save your modifications.

Project Analyzer Enterprise Edition includes an additional feature to automatically remove dead code. You can find this feature in the Enterprise menu, Auto-Fix command. It makes a copy of your source, and removes or comments all dead code. In addition, the Enterprise Edition supports multi-project analysis, allowing detection of code that's not called by any of several related projects.

Dead procedures. Dead subs, functions, properties and methods usually make up most of the extra code. Some Project Analyzer users have reported that 30 - 40% of their project was made of these things!

There are two kinds of dead procedures.

1. Procedures that are not called in the code. This is the primary case. You can safely remove any single dead procedure.

2. Procedures that are mentioned in the code but never executed. These procedures are marked as "called by dead only". The explanation is that there is a pair or even a larger group of procedures that call each other. However, no calls come from outside this group. So in effect, the whole group never gets executed! You need to remove the whole group at once, otherwise VB will complain about some missing procedures when you compile. The best way to do this is to select a dead procedure and follow its calls/called by relationships to find what the group consists of. The Project Graph and Enterprise Diagrams features can also be very helpful with this. Consider yourself lucky. This chunk of dead code might have been impossible to find without Project Analyzer.

Dead variables and constants are usually left behind because you alter some routines and don't remember to remove the extra declarations. They consume some extra memory (if it's an array, that might be a lot!) and make your program harder to understand.

Dead constants are simple: they are those ones that are not used at all. Dead variables come in two variations:

1. Dead variables that are not read nor written to.

2. Dead variables that are written to but never read. These variables are marked "assigned only". You should remove these variables. Why would you ever need to store a value if you don't read it? – There is one exception, but Project Analyzer handles it. The exception is object variables. Sometimes it makes sense to store an object in a variable without ever actually using the reference, so that the object stays alive doing whatever it is supposed to do. You don't need to worry about this; Project Analyzer automatically detects it and won't report a problem.

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, just modify this statement to

```
MyFunction y, z
```

Dead Types and Enums are simply those ones that are never used in the code. They're not likely to be a major problem source for you, but why keep them if you don't need them? You can safely remove a dead Type or Enum declaration if you're sure you won't need it at a later time.

Empty procedures are not exactly dead code, but they are still extra. There may be a reason for an empty procedure, but for most of the time, it's just taking up space and possibly slowing your program down if you happen to accidentally call it. Check if you really need the empty procedure. – Empty procedures might be required in an abstract base class or a class that implements an interface. Project Analyzer knows about this case.

Events. Event definitions (Event statements) and event handlers are a special case. An Event statement is dead if it's not raised nor handled. An Event statement may also be reported as Event not raised, if the event is handled but you don't call RaiseEvent to fire the event. In this case, you need to consider whether to raise the event or remove it. Yet another case is when an event fires but there are no handlers. This is not a problem case since event handling is optional. – Project Analyzer reports orphaned event handlers as dead procedures. An example of an orphaned event handler is Button_Click, where the original control named Button has been renamed or removed.

Return values. If a function returns a value, the callers are usually supposed to store or use it. Where a caller doesn't store or use the value, Project Analyzer reports the problem Return value discarded. If all callers discard the return value, Project Analyzer also reports it as Dead return value. You should consider using the return value or alternatively, rewriting the function as a sub.

Dead classes are ones that are not used at all. You can remove them. A class may also be semi-dead in that it is being used as a data type but it is not instantiated at run-time. Another semi-dead case is when an abstract MustInherit class is not inherited (.NET only).

Dead interfaces. You can remove .NET Interface definition that is not used at all. An interface may also be semi-dead in that it is being used as a data type but no class implements it. Another case of semi-deadness is when an interface is implemented but not actually used anywhere. In both cases, the interface is possibly unnecessary, but consider the intended use before you throw it away.

Dead structures. You can remove a .NET Structure that is not used at all.

Dead modules can be removed in their entirety.

Unused files. These are files that no other files refer to in the source code. They may be classes that are never instantiated, forms that are never shown, or standard modules whose procedures are no longer needed. Check to see if you really need these files and remove them from your project.

Exposed code and multi-project analysis

Dead but exposed. Some library and server project types expose an interface that other projects may use. For example, an ActiveX project defines a public class with methods for other programs to call. In this case, to determine calls to the exposed code, it's not enough to analyze just the one (ActiveX) project. The word "exposed" next to a dead code problem means that the code is not used by the project it's defined in, but it might be used by some other project(s).

In practice, it may be necessary to leave all exposed code undeleted, even if dead. That's because other programs may require the code and fail if you remove it. Project Analyzer's default settings hide all dead code problems for exposed code. You can enable dead code reporting for exposed code in problem options.

To reliably determine the deadness of exposed code, you need to do a multi-project analysis. This requires the Enterprise Edition. Multi-project analysis is able to detect dependencies between projects, and report dead code that's not used by any of the projects. You do a multi-project analysis by selecting more than one file to analyze, or alternatively, by opening a .vbg or a .sln file. Read the help file for in-depth instructions on how to do a multi-project analysis.

Multi-project analysis is recommended for the following project types: project groups (.vbg), solutions (.sln), ActiveX ocx/dll/exe projects, OLE server projects, and .NET class and component library projects.

Unnecessary controls to remove

Project Analyzer supports a set of *unnecessary control detection rules* for projects written with VB 3.0 to 6.0.

Most user interface controls are not useful if they are not visible and enabled at run-time. If you find them, you should consider removing them as they take up resources and increase the executable size. Events related to the controls are possibly not executed. Code that reads or sets the controls' properties and calls their methods is potentially unnecessary for the operation of the program.

Control not visible. A control's Visible property is set to False and the control is not made visible at run-time.

Control outside of visible area. A control is located outside of the visible area of the form it is on. At run-time, the control is not moved and the form is not resized to reveal the control.

Control not enabled. A control is disabled at design-time and not enabled at run-time.

Why are unnecessary controls left in the program in the first place? Developers sometimes set controls to invisible or disabled mode or drag them outside the visible form area. They do this to quickly remove controls from the user interface while still keeping some of the functionality of the controls, such as the use of control events or properties.

There are some cases where an invisible or disabled control can be useful, so you have to be careful about removing the control and related code. Note that the unnecessary control detection rules do not reveal all unnecessary controls; you should also verify each form visually for control arrays (which are not supported by these rules) and controls placed behind other controls.

3.3 Further optimizations

Option Explicit. If you don't use Option Explicit, you've got to learn it now. You need to write Option Explicit at the beginning of each file. In VB.NET you either write Option Explicit On or select the respective option in project options. This makes Visual Basic require variable declaration. Always declaring your variables is good programming practice and widely recommended by VB experts. Explicit variable declarations make your code more readable and require less RAM space. It might even make your program run faster, especially if you use a lot of objects in your code. How come? That's because you give all your variables a type. Read on why that's important.

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

Why is a Variant or an Object bad? The first reason is because it takes far more memory space than the other data types. You can save memory if you declare your variables, and most importantly, your arrays as Integer, Single, String, etc. The second reason is that using specific data types removes run-time errors. You never know what a Variant/Object contains, but an Integer always contains a number.

Proper typing is especially important for procedure parameters. That's the easiest way to ensure that you receive a number instead of a string, or that parameter lb is a ListBox and none of those other controls.

Here is yet another reason why you should use Option Explicit. It makes you think of the data types when you're writing the Dim statements. If don't declare them, they become Variants/Objects 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 data type whenever you can, like this:

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

There is a catch in classic VB. 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
```

VB.NET variable declarations work differently, the above x is Integer.

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

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

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

In VB.NET you can use Option Strict On to require a type definition for all variables and functions. In VB classic, there is no way to require that, so you need to run Project Analyzer.

Object variable declared As New. In VB Classic, declaring an object variable As New creates an auto-instantiating variable. Each time you read the contents of the variable, VB first checks if the variable contains an object, and creates one if not. This adds overhead, thus slowing your program down. To achieve better performance, remove the word New from the declaration, and instantiate your variable (Set x = New Class) before it is used. It makes sense to test with 'If x Is Nothing Then' before accessing the variable, to avoid the run-time error 'Object variable not set'.

Consider short-circuited logic. In the expressions (x And y), (x Or y), both operands (x, y) are evaluated. Short-circuiting means rewriting this so that when x=False in (x And y), y is not evaluated. The same goes for x=True in (x Or y). This saves CPU cycles, especially if y is a complex expression. In VB.NET, consider replacing And with AndAlso, and Or with OrElse. In VB Classic, consider splitting an If ..And.. condition as two nested Ifs. Short-circuiting If ..Or.. yields more complex code, usable case by case. Risks: Short-circuiting changes the logic. If the second operand calls a function, this call may not execute. Read VB help for differences between And/AndAlso and Or/OrElse.

String handling. Project Analyzer supports a set of string optimization rules. They are designed for string-intensive programs whose performance is limited by the performance VB's string functions. To learn more about this specific area, read the help file and visit our web site for an extensive article on string optimization.

3.4 Style suggestions

Now let's talk about coding style. We all have our own style. That's perfectly fine! There are many ways to write good and beautiful code. But be consistent. Develop your own coding standards and stick to them. It will make your life much easier later when you have to maintain your own code, or when another person tries to learn it.

Project Analyzer reports so many style issues that it might look just overwhelming at first. Fortunately, you're not supposed to follow all the rules. Pick those ones that are useful for you, and forget about ones that are too exotic or complicated to enforce. Define your very own problem filters with the Options|Problem Options menu command.

Some of the more important style issues are described here. You can find the rest in the help file.

Option Strict missing. A file does not define Option Strict On. In .NET, Option Strict On enforces type-safe code by allowing only widening data type conversions. It also disables late binding. You can set it as a project-level option or for each file. Disallowing late binding helps you avoid unnecessary run-time errors and add performance. Besides, analyzing your projects with Project Analyzer becomes more accurate when all calls are early bound.

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. Consider this syntax:

```
Sub Calc()
```

Can this be called from outside its own module? Can you immediately tell if Calc( ) is Public or Private? Many of us can't. So, why not write one of the following:

```
Private Sub Calc()
Public Sub Calc()
```

A Private thing cannot be called from other modules. 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. It's important to declare a proper scope to keep your code modular.

Friend, Protected or Protected Friend might also come into question in object-oriented programming. In Friend access, the declaration is available inside the project, but not exposed to any outside projects. The distinction between Public and Friend is important if you're writing a library project that exposes an public interface to other libraries or programs.

Protected scope, available in VB.NET, is like Private but it gets inherited to any descendant classes. Protected Friend is a combination of Protected and Friend, it gets inherited and it's also available in the declaring project.

Excess scope. A part of your program has a scope that's too wide. Even if you could, you don't actually use this part up to the scope. Check if you should declare the part with a more restricted scope. It is good programming practice to use as limited a scope as possible to prevent other parts of the program from calling and modifying parts that they shouldn't have anything to do with. It's especially important to encapsulate class variables, to make them inaccessible to other

parts of the system. This way you can protect your data from being modified, and also you can later change the way you store your data without affecting other parts of the system.

ByRef/ByVal missing. Incorrect or loose parameter declaration is a potential source of hard-to-find bugs. Always use ByRef or ByVal when declaring your procedure parameters, and your code gets more robust.

What exactly do these ByVal and ByRef things mean?

ByVal tells VB to pass a parameter by value. That is, the value of a variable is passed from the caller to the callee, but the variable itself isn't. This makes your code safe. Whatever value you write to the parameter in the callee, the changes won't affect the variable inside the caller.

ByRef tells VB to pass a parameter 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 caller too! This may lead to errors that are really hard to notice. Example:

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

Sub Main()
    Dim Text As String
    Text = "Hello, fellow!"

    DoThings(Text)
    MsgBox(Text) ' Shows Hello, world!
End Sub
```

```
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
```

The default in VB versions up to 6.0 is ByRef. That's the unsafe one! So if you don't choose between ByVal and ByRef, you always get ByRef. That's why it's important to write the words explicitly.

With VB.NET things get even more complicated. The VB.NET default is ByVal. However, when you migrate your existing code to VB.NET, the migration wizard will upgrade all missing declarations to ByRef. It is important that you go through all your code before you migrate it to VB.NET and add ByVal/ByRef where needed. It will be harder afterwards.

It should be noted here that there are good uses for ByRef. One case is when you wish to return values via the parameters. That is usually best avoided, but it's possible, and sometimes required. Another case is to obtain maximum speed. The copying of ByVal parameter values takes time, and if you call the procedure thousands of times, it might pay off to use ByRef. That depends on the parameter data types, and you should experiment with it if you need to squeeze out more speed. If you do a lot of string processing, the use of ByRef string parameters instead of ByVal may give your program a real performance boost. Also sometimes VB just demands ByRef. At those times it will tell you about that.

The not-so-stylish statements

The following is a list of statements that represent bad coding style.

Goto/Gosub are bad programming practice. They should be avoided when possible. Jumping around with Goto easily leads to unstructured control flow structure and spaghetti code. Gosub has low performance and isn't supported by VB.NET.

Exit/Return. Use of the Exit and Return statements may indicate unstructured program flow, much the same way as the Goto statement. Try to build your loops and procedures so that they only have one exit point, which is at the end of the loop or procedure. It is especially important to avoid jumping out of With..End With blocks.

There is one case where VB requires Exit/Return. This is when you need to quit a procedure immediately before an error hander. Project Analyzer allows this use.

Note: The Return statement is equally avoidable in all versions of Visual Basic, even though its functions are different. In VB 3-6, it is used to return from a GoSub jump. In VB.NET, it is used like Exit Function.

While. The While loop is outdated. Do...Loop provides a more structured and flexible way to perform looping. In VB 3-6, the syntax to avoid is While..Wend. In VB.NET, it is While..End While.

Call is not a necessary statement to call a procedure. Leave it out.

IIf / Switch / Choose. These functions are considered bad programming style because of functionality, optimization and readability issues. All conditions and branches of IIf / Switch / Choose are always executed resulting in longer execution

time and possible unwanted side-effects. Switch and Choose may return an unwanted Null. Use Select Case or If..End If instead.

Single-line If..Then statement. An If..Then(..Else) construct is on a single line. To make your program more readable, split the construct to multiple lines.

Multiple statements on one line. There is more than one statement on a single line, separated with a colon (:). To make your program more readable, write only one statement on one line.

On Error. The use of On Error for exception handling is outdated in VB.NET projects. It is better to use the Try..Catch block for error handling. On Error Resume Next also deteriorates the performance of .NET execution. Use Try..End Try without the Catch block to achieve a similar effect without the performance hit.

Global found, use Public. Early versions of VB didn't have the Public keyword. In VB 4.0, 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.

Explicit clearing of variables, arrays and resource handles. This is a group of rules, under the Style category, for developers who want to ensure that their code explicitly clears object variables, dynamic arrays and Win API resource handles after use. You can use these rules to avoid memory and resource leaks in object oriented code and API calls. Read the help file for a thorough description.

Functionality problems

Even if you couldn't care less about performance or coding style, you should still be concerned about how your program works for the users. . This is where it really pays off to scan your application with Project Analyzer's problem detection feature. When you fix these issues before finishing your project the users will find your program working better than what it would have been without the fixes. Most of the functionality problems are related to user interface issues, error trapping and event handling.

Error handler missing. A procedure has no error handler. Your program may crash if a run-time error occurs in this procedure.

Delayed error handler. A procedure uses delayed error handling. By delayed error handling we mean the use of On Error Resume Next or a Try block without Catch. Run-time errors are handled in the procedure(s) that call this one. This may be completely as you planned, or you may have forgotten to add an error handler (On Error Goto, or Catch in a Try..End Try block).

You can ignore the error handling issues in small procedures. For example, you may consider that procedures with just 1 or 2 lines are unlikely to cause crashes. While we don't recommend this approach, Project Analyzer allows you to set a minimum limit on the procedure size requiring proper error handling.

Consider using Aivosto VB Watch, a tool that adds advanced error handlers to your code.

Error event missing. Every Data control should have the Error event implemented. If you don't do it, Visual Basic only displays a simple error message. It is also important to use an error handler inside the Error event, because if a new error occurs, the Error event fires again. This problem is available for VB 3-6.

Click event missing. A CommandButton or menu item does not do anything when clicked. This problem is available for VB 3-6.

Timer event missing. A Timer does not fire an event at defined intervals. You could as well remove the timer. This problem is available for VB 3-6.

Events not handled. An object variable is declared WithEvents. However, none of the events are handled. Why did you declare it as WithEvents in the first place?

Form missing Icon. A Form doesn't have an icon that is required. A default icon, generated by VB, will be shown. This problem is available for VB 3-6.

Form missing HelpContextID. Your program uses context-sensitive help (F1), but a Form was found that has no HelpContextID set. This problem is available for VB 3-6.

Default/Cancel button missing. A dialog box has CommandButtons but none of them is marked Default/Cancel. When the user hits Enter/Esc, none of the buttons handle it as a Click event. This problem is available for VB 3-6.

Resizable Form missing Form_Resize. The Form_Resize event is missing from a Form that users can resize at run-time. Your application may look odd if you don't respond to Resize events. This problem is available for VB 3-6.

Hotkey conflict. Two or more controls or menu items share a hotkey. For example, there are options &Save and &Search in the same menu. This problem is available for VB 3-6.

Hotkey missing. A control or menu item is missing a hotkey. Checked controls: CommandButton, CheckBox, OptionButton, Frame, Menu. This problem is available for VB 3-6.

Possibly twisted tab order. The tab order of controls looks questionable. Take a look at the TabIndex properties of the mentioned controls. The detection algorithm of this problem is not foolproof, and the tab order might be all right in some cases. This problem is available for VB 3-6.

The above is not a comprehensive list of problem detection rules in Project Analyzer. See the help file for a complete list.

3.5 Customize code review with comment directives

Sometimes Project Analyzer reports a problem that you wish to ignore. Maybe it's a dead procedure that you'll need later, or maybe it's a syntax convention you like to use in some special case.

Project Analyzer has a special feature to ignore certain problems in a given range of code. You use it 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 how to behave. See topic *Comment directives* in the help file for the exact syntax.

3.6 Optimizing with Super Project Analyzer

Feature requires Pro or Enterprise Edition.

Super Project Analyzer is a feature in Project Analyzer that analyzes a 'super project'. That consists of projects that share some source modules. Super Project Analyzer combines the results of several analyses to report dead code.

> Super Project Analyzer requires that you have several projects that don't call each other, but that share some of their source code files. Super Project Analyzer does **not** handle project groups (.vbg files) or solutions (.sln files). It does **not** detect cross-project calls. See multi-project analysis (Enterprise Edition) for how to analyze that kind of code. In most cases, multi-project analysis is a better way to detect dead code than Super Project Analyzer.

Why do you need a separate super project analysis? When a file is included in several projects, it is impossible to tell if a procedure is dead or alive without looking at all of the projects. Super Project Analyzer takes the output of several analyses and forms a general picture of what can be removed and what can't.

To use Super Project Analyzer, you first analyze some projects the normal way. When an analysis is complete, open the Pro menu and click on *Save data for Super Project Analyzer*. After saving enough data, start Super Project Analyzer and load all these files with the Add project menu command.

Super Project Analyzer can also prove useful in some special cases.

Mutually exclusive compiler directives. If you use compiler directives (#If..Then..#Else..#End If) to produce several programs out of the same code, a normal single analysis may not be enough. That's because Project Analyzer ignores the false branches of the compiler directives, and the results remain incomplete. In this case, analyze the code under all the different compiler directive settings and save the results for Super Project Analyzer. – If your compiler directives allow a configuration where there are only True branches, or just a few False branches, you're better off. In this case, you might not need Super Project Analyzer. Instead, define the compiler constants so that most branches evaluate to True. This way you get the most complete picture.

Multi-project analysis, reference depends on project. This case applies to a multi-project analysis where a single source file belongs to several projects. If the calls from that file go to a different places depending on the project the file is in, Project Analyzer will find calls based on one project only. Example. You have 2 projects. Both of them define a different constant named APP_TITLE, with values "Standard Program" and "DeLuxe Program". Both of the projects also contain the same file library.bas, which uses the value of APP_TITLE. Now, APP_TITLE in library.bas means a reference to 2 constants, depending on the project. Multi-project analysis will detect just one reference, and the other APP_TITLE may look dead. In this case, analyze the projects separately and use Super Project Analyzer to combine the results.

3.7 Automatic code optimization with Problem Auto-Fix

Feature requires Enterprise Edition.

Project Analyzer is capable of fixing a subset of the found coding problems. This includes automatic handling of dead code by deleting it or commenting it out. Problem Auto-fix is a feature of Project Analyzer Enterprise Edition. In the Standard and Pro Editions, you're limited to manually fixing the problems. Auto-fix is at its best in processing a large amount of code where manual work would take too long.

Auto-fix is available for projects created in Visual Basic 4.0, 5.0, 6.0 and .NET. Visual Basic 3.0 is not supported, neither is VBA.

Running Auto-fix

Analyze a project and start Auto-fix in the Enterprise menu. A dialog box appears. Auto-fix will generate a copy of your project and work on it, without touching the original files in any way. So you need to select a new, empty directory for the cleaned project. For your safety, the cleaned files will be put into this directory, regardless of which sub-directory they were originally in. This makes sure that you never need to fear of losing any of your code.

Auto-fixable problems

This group of problems is handled automatically. Auto-fixable problems include dead code, simple optimizations, procedure, variable and constant declarations, and missing events. A list of auto-fixable problems is in the help file in topic "Auto-fix problem list". Your options are:

Fix & comment. This option makes Auto-fix repair all auto-fixable problems and leave a comment next to the modified code. The comments will bear a prefix such as '! or 'HACK: for you to notice them. This option handles dead code by commenting it out. You can define a special dead code comment such as the '+ prefix.

Fix quietly. This option repairs all auto-fixable problems, but adds no comment. It handles dead code by deleting it.

Treat as manual fix. Selecting this option causes all auto-fixable problems to be treated the same way as manual fix problems. Use this option if you wish to review each change manually before accepting it.

Manual fix problems

This group of problems allows no auto-fix because of complexity or need of programmer skills. Instead, you have the option to mark the problems in your code so that you can easily handle them by yourself. Your options are:

Comment. Write problem descriptions in code as a comment. Each comment gets the prefix you choose, such as '! or 'TODO: or 'UNDONE:. You for these comments in the code and fix the problems little by little. Use this option if you have lots of problems to handle and expect that you can't fix them all in a short time.

Alert. Write problem descriptions in code and prefix them with the dollar sign ($). VB will not compile the code before all these problems are handled. This allows you to do fix & try-to-compile cycles in VB.

Ignore. Ignore all problems that require a manual fix. Don't add a comment or alert. Use this option in combination with the Fix & comment or Fix quietly options. This allows you to handle the auto-fixable problems automatically while ignoring the rest. Run a new analysis later to take care of them. This option is also good for cleaning the project of dead code before compilation.

Auto-fix tips

After running Auto-fix, your newly generated clean project might not compile or run. This might be due to code that was removed but that shouldn't have been. Sometimes source code analysis is not capable of detecting all dead code correctly, due to the fact that certain language features in VB make it able to call a piece of code at run-time so that the call is not explicitly visible in the code.

Because of this, you should start by auto-fixing a limited number of problems, and perhaps selecting a limited number of files at a time. Define a problem filter that selects just dead procedures, for example. Use the Fix & comment option to comment out the dead code. Try to compile. If it won't compile, you can always restore the required procedure by removing the comments.

You can overcome the occasional limitations of the analysis by writing special comments in the code. For example, the comment '$ PROBHIDE DEAD will hide the dead code problem for the selected piece of code. This feature is called Comment directives and is described in the help file.

Auto-fix is not recommended for partially analyzed projects. Correct detection of certain problems, such as dead code, requires that the full project has been analyzed.

Integrating with the Visual Studio .NET Task List

If you have Visual Studio .NET, the 'TODO: style comments will show up in the Task List. You can use markup such as 'TODO:, 'HACK: and 'UNDONE: for things you need to take a look at. You can even define your own markup such as 'FIXED: or 'REVIEW: by adding these strings in both Project Analyzer and in Visual Studio .NET (see the Options dialog in the VS.NET Tools menu).

3.8 Enforcing naming standards with Project NameCheck

Feature requires Pro or Enterprise Edition.

Every programming project includes a large set of names. Functions, variables, user-defined types, constants, Enums, controls, module names, … It is important for names to be descriptive and clear. Following a naming standard makes your code consistent. It is easy to read and understand, especially when working in a team. When you see a properly formatted name, you can immediately tell what the name is for, what type it is, and what scope it is.

What is a naming standard? To make a complex issue simple, a naming standard is about accepted prefixes, suffixes and case conventions. For example, gintUserID might stand for a global integer containing an ID number for the current user, and gstrUserName might be the user's name. When you see this name, you immediately know how you are supposed to use it, and you can tell that modifying the value might have effects in other parts of the program. Your standards might also state that constants must be in ALL_CAPS. This way you are not likely to try to assign a new value to PRODUCT_WEIGHT. Moreover, when you see a Variant called datBirth, you're not going to assign the variable a string of a city name, but a birth date. Thus, naming standards can prevent both compilation and run-time errors.

It’s not that you need to follow someone else’s standards. You should define your own standards and use them. There is no single, universally accepted standard. Project NameCheck includes one example. You’re free to fine-tune it or define a completely new one. Just remember, it makes more sense to use a loose standard consistently than define a great standard and then forget about it.

Getting started with Project NameCheck

Hit F8 to run Project NameCheck. First, you need to specify the standard you wish to use. Project NameCheck comes with default settings for VB 3-6, VB.NET and VBA, but you will most likely want to configure them to suit your needs. Read on to find configuration tips.

When you’re done, go to *Options|Problem Options*. Select the <Project NameCheck> filter. Of course, you can use any other filter that includes NameCheck problems, like <Style> or <Strict>. These include many other problems too, so it will be easier to start with <Project NameCheck>.

Now, analyze a project. NameCheck problems will appear in the problem view at the bottom of the main window. Project Analyzer also lists suggestions of accepted names. If the suggestion doesn’t seem right to you, you need to fine-tune the standard.

How do I configure a standard?

Usually you use just one naming standard. If you follow complex naming conventions, you may even define several standards and switch from one to the other when required. But let’s assume now that we’re using just one standard. Start Project NameCheck now.

Standards terminology

Names in your project consist of a prefix, a base name, and a suffix. Not every name has all of these. Some name might have a prefix+base, another might have base+suffix, and a special name might have no prefix nor suffix at all. Prefixes and suffixes tell you, the programmer, what scope the name has, and what type it is.

The length of a prefix or suffix is usually 1 to 3 characters. A prefix is often a lowercase abbreviation like “lng”, but it could as well be an uppercase letter like “T”. A suffix is often one of the VB type characters ($%&@#!), but you could as well use a suffix in the form of “_int”, “_enum” etc.

There are prefixes and suffixes for each scope and each type of a name. There are also special cases, but let’s now restrict ourselves to the scopes and types.

Scope The scope where the name is available. It can be global, procedure-level, etc. A typical scope prefix is "g" or "glb" for global, and "m" for module-level.

Type The type of a variable, function or control. Typical type prefixes include "cmd" for CommandButton, "int" for Integer and "E" for Enums.

When you combine Scope and Type with <base>, you get 10 possible combinations. They are here shown for the variable containing an age.

Combination	**Example**	**Declaration**
Scope+Type+ <base>	giAge	Public giAge as Integer
Type+Scope+ <base>	igAge	Public igAge as Integer
<base> +Scope+Type	Age_glb%	Public Age_glb%
<base> +Type+Scope	Age_int_glb	Public Age_int_glb as Integer
Scope+ <base> +Type	gAge%	Public Age%
Type+ <base> +Scope	iAge_glb	Public iAge_glb as Integer
Scope+ <base>	gAge	Public gAge As clsPersonAge
<base> +Scope	Age_glb	Public Age_glb As clsPersonAge
Type+ <base>	iAge	Public iAge As Integer
<base> +Type	Age%	Public Age%

Some of the combination may not actually look very useful, at least not for a global integer. Your job is to decide which combination(s) you wish to use. To learn how to set all the various standard configuration options, read the help file topic "Configuring a standard".

Sharing the standards

Project NameCheck has Export/Import functionality. You can save the settings to a .std file and distribute to other computers. That's especially useful to enforce standards among teams.

4 How to document your code

In many projects, documentation is the last and the most boring job. 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 learning effort 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.

4.1 Simple listings of files, procedures, variables...

The sub-menu Report|Lists is the home of a large selection of listing type reports. You can get a report of all the available declarations in the code.

- Files (sorted by name, size and date)
- Files and projects (which file belongs to which project)
- Modules
- Namespaces (.NET)
- Procedures (by location or sorted alphabetically)
- Variables and constants
- Types, Enums and COM Aliases

4.2 Listing procedures with comments

A commented listing of procedures is a simple but useful form of code documentation. Project Analyzer makes this listing by the menu command Report|Lists|Procedure list with details. For each procedure, this report lists 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 feature. It is described later in this tutorial.

4.3 Listing variables and constants

Procedures are not everything a project contains. Variables and constants are equally important for documentation. You can list variables and constants in two ways:

- In the Report|Lists sub-menu, select the Variables and constants report. This report lists all variables, constants and parameters defined in your program.
- In the View menu, open the Variables, constants and parameters window. This window lets you list, filter and sort the variables, constants and parameters and then get a report of them.
- In the View menu, open the Constants and Enums window. This window lets you list and report all available constants and enumeration constants. You can also perform analyses on them, such as search for duplicated constants.

4.4 Documenting all your code with Project Printer

Feature requires Pro or Enterprise Edition.

The most thorough documentation of a project is its source code. Project Printer is a feature in Project Analyzer that makes documenting the whole source code easy. It isn't a plain text file print-out utility. Even though the name suggests 'printer', you can also generate document files with it. Optionally, Project Printer formats code for easier reading, generates a table of contents, and includes cross-reference information, problem descriptions 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.

A third use for Project Printer is turning your project into a *code web site*. This way you can browse your code and learn how it works. You may even put this site available online for your colleagues to learn from it.

In the following you learn a few easy ways to use Project Printer. You're not restricted to these choices. Project Printer is rather complex because of the various options. Some option combinations are more optimal than others. Experiment to get the most out of this feature.

Turn code to a web site

Hit F6 to run Project Printer. Select the files you want to include in your web site. Usually it's best to select all the files, but a partial web site may also be enough. Click Next and select Full source code. Click Next again. Now, click on *HTML directory – project web site...* and select an empty directory. You can create a new directory or clean an existing one. Click Next and then generate the web site with the default options.

You now have a web site consisting of a large number of .html files. The start page is index.html, as you could expect. Calls from procedure to procedure are hyperlinked. You also have a set of reports generated as a bonus.

Alternative. Instead of a lot of .html files, you can put the entire web site in a single .mht file. Just select *MHT – single-file project web site* as the output format. This option makes an identical site to that of the HTML directory option, but all the data is in a single file. Internet Explorer is able to open .mht files, but some other web browsers may not support them. The great thing about the .mht format is that it's a single file instead of thousands of files.

Print your code

Hit F6 to run Project Printer. Select the files for which you wish to get a document. Click Next, choose Full source code. Click Next again, and choose File. Select HTML as the output format. This option will generate a long web page for you to print. It's especially nice if you have a color printer, because the code is syntax highlighted. If you have a monochrome printer, you can select the Printer option.

Alternative. Instead of printing it all, you can keep the document in the .html file.

Generate a Comment Manual

Hit F6 to run Project Printer. Select the files for which you wish to get a document. Click Next and choose Comment manual using a special comment syntax. Click Next again and choose the appropriate file type or Printer.

Comment manual is a manual-like report based on comments in your source code. It doesn't contain your code, just the procedure declarations with comments. This function is an enhanced version of listing procedures with comments (as introduced earlier in this document). What is different is that you can use special comment syntax to get a nice-looking, formatted manual.

Read the help file for "Comment manual" for instructions on how to prepare your comments for this feature.

If you don't have the time to format your comments according to the supported syntax, you can use the option Comment manual with unformatted comments. Yet you should read the help file to find out exactly which comments will be included in the document.

4.5 Documenting 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, for files and for procedures. A *file dependency tree*, or *file use tree*, tells which other files a given file needs to compile and/or run. A *procedure call tree* is more exact: it tells which procedures are called by a given procedure.

Ordinary 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. It's in the View menu. Use the Report button to get a report of what you see on the screen.

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

Need 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. The 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.

5 Diagramming

Project Analyzer provides two features to document the structure of a program in a graphical way.

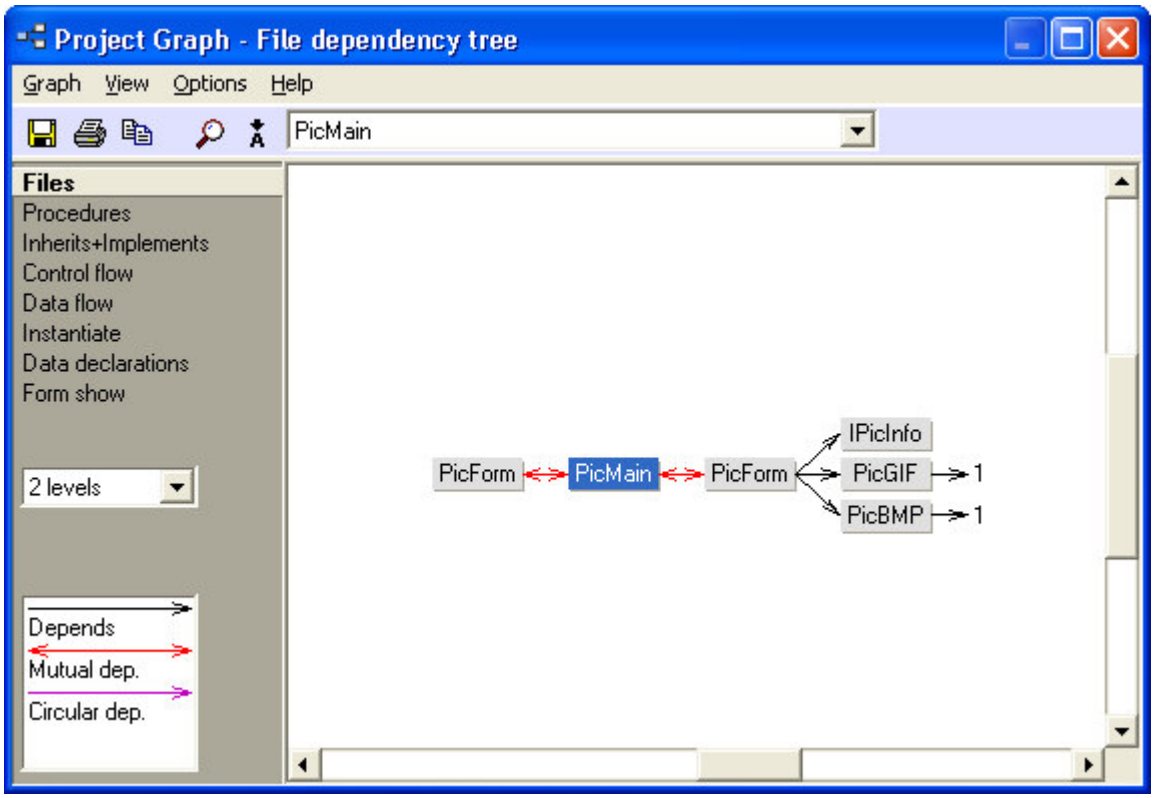
- Project Graph is an interactive tree-format graphical navigation system. You can use it to navigate your project and also to document it. Project Graph does not try to make a comprehensive visualization of all connections in a program. Instead, it offers a way to get different views inside it.
- Enterprise Diagrams makes more complex diagrams for documentation purposes. Instead of interactivity, it offers the option to visualize all links (calls, dependencies, variable uses etc.) in a single diagram.

Thus, you can use Project Graph for interactive graphing and Enterprise Diagrams for comprehensive documentation.

5.1 Project Graph

Feature requires Pro or Enterprise Edition.

The most intuitive call trees are graphical. Project Graph is a feature in Project Analyzer that can produce graphical call trees. It also makes other types of interactive graphs of your system.



The available graph types are:

File dependency tree. This diagram shows *file requires/is required by* information. A → B means file A requires file B to compile or run. A red double-headed arrow indicates mutually dependent files, which should be avoided if possible.

Procedure call tree, both forwards and backwards. Arrowheads show the direction of control flow. A double-headed arrow indicates procedures that call each other.

Inherits and Implements tree displays inheritance by statements Implements and Inherits (Inherits applies to VB.NET only). Not all projects use inheritance. Therefore, the graph may be empty.

Control flow tree. This graph shows how procedure execution traverses between modules. It is the equivalent of a procedure call tree taken to the module level.

Data flow tree. This diagram shows all data flows between modules. Data flows via *variable read/write*, *procedure parameters* and *function return values*. It is typical that data flows in both directions between 2 modules as the modules pass data via parameters and return values.

Instantiation tree shows where classes are being instantiated.

Data declaration tree shows where classes, structures, interfaces and forms are being used as data types. This graph also shows instantiate relationships (as blue arrows).

Form show tree. This graph type describes the order in which forms show other forms by calling Form.Show (or Form.ShowDialog in VB.NET). Form1 shows Form2 if it either calls Form2.Show, or the procedure call trees starting at Form1 contain a call to Form2.Show. Form1 may also show Form2 if Form1 contains a UserControl that shows Form2. This graph does not include self-showing forms. A self-showing form is one that displays itself in its constructor or event such as Form_Load. Showing forms by setting their Visible property to True is not included in this graph.

Project dependencies. This graph will appear in a multi-project analysis to show dependencies between several projects.

Because the graphs easily get huge, the size of a graph is automatically limited to max 6 levels forwards and max 6 levels backwards. You can get more levels by right-clicking a node and selecting Expand. On complex systems you may find out that even 2 levels is too much. In this case, you need to take more than one picture. Open the Options menu for some settings to keep the graph size smaller.

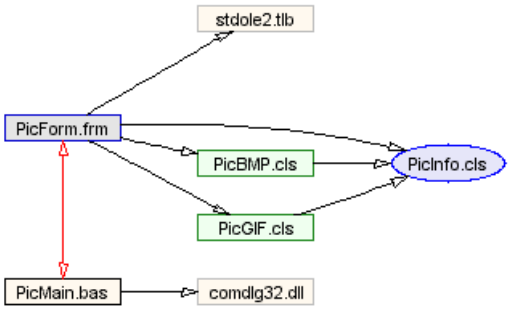
To print the graph call tree, use the Print button. This prints a page with the graph that is currently shown on the screen. To include the picture in your documentation, press Ctrl+C to copy to the clipboard and paste the resulting picture in your word processor. You can also save the picture into a file.

5.2 Enterprise Diagrams

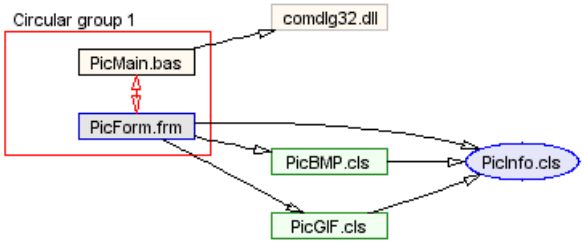
Feature requires Enterprise Edition.

Diagram types

The diagram types in Enterprise Diagrams are roughly similar to those in Project Graph, but Enterprise Diagrams also provides some additional diagrams not present in Project Graph.



File dependency diagram



File dependency diagram, circular groups boxed

File dependencies. This diagram shows *file requires/is required by* information. A → B means file A requires file B to compile or run. A red double-headed arrow indicates mutually dependent files, which should be avoided if possible. File dependency diagrams also come in the following variants:

- **File dependencies, circular groups boxed**. This file dependency diagram emphasizes circular file dependency groups. Consider removing the circular dependencies to reach better reusability.
- **File dependencies, circular groups collapsed**. Save space by collapsing circular dependency groups. The groups are frequently very complex inside, making them hard to visualize. This option hides the circular dependencies in effort to make the diagram easier to read. To see the hidden links inside the groups, get the *circular groups boxed* diagram for each group.

Inherits and Implements. This option builds a full class hierarchy diagram with interfaces. Not all projects use inheritance or interfaces. Therefore, the diagram may consist of separated classes only. For classic VB projects the diagram displays only Implements since there is no Inherits available.

Control flow. This diagram shows how procedure execution traverses between modules. It is the equivalent of a procedure call diagram taken to the module level.

Data flow. This diagram shows all data flows between modules. Data flows via *variable read/write*, *procedure parameters* and *function return values*. It is typical that data flows in both directions between 2 modules as the modules pass data via parameters and return values.

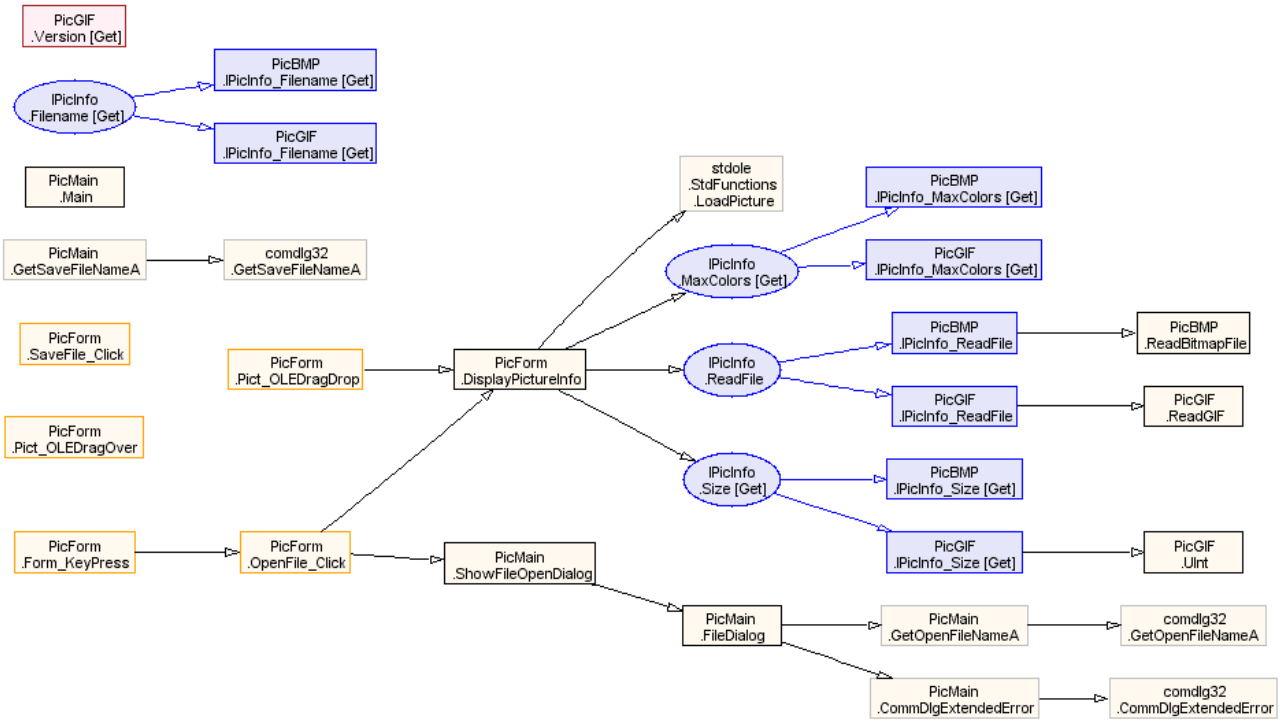
Variable data flow. This diagram shows data flows between modules *via variable read/write*. The difference to the Data flow diagram is that procedure calls and return values are not taken into account. Thus, this diagram considers only direct variable based data transfer.

Variable access. This diagram shows direct *variable access* between modules. The only difference to the Variable data flow diagram is the direction of the read access. This diagram is useful for reviewing access to global variables and data structures. Direct variable access is often considered bad. Instead of direct variable access, one should access data via properties or functions. This diagram reveals the direct variable access that could be rewritten.

Instantiate. This diagram shows where classes are being instantiated.

Data declarations. This diagram shows where classes, structures, interfaces and forms are being used as data types.

Form.Show order. This diagram type describes the order in which forms show other forms by calling Form.Show (or Form.ShowDialog in VB.NET). Form1 shows Form2 if it either calls Form2.Show, or the procedure call trees starting at Form1 contain a call to Form2.Show. Form1 may also show Form2 if Form1 contains a UserControl that shows Form2. This graph does not include self-showing forms. A self-showing form is one that displays itself in its constructor or event such as Form_Load. Showing forms by setting their Visible property to True is not included in this graph.



Procedure call diagram (Enterprise Diagrams)

Procedure calls. The procedure call diagrams display all calls between the selected procedures. The diagram comes in three variants:

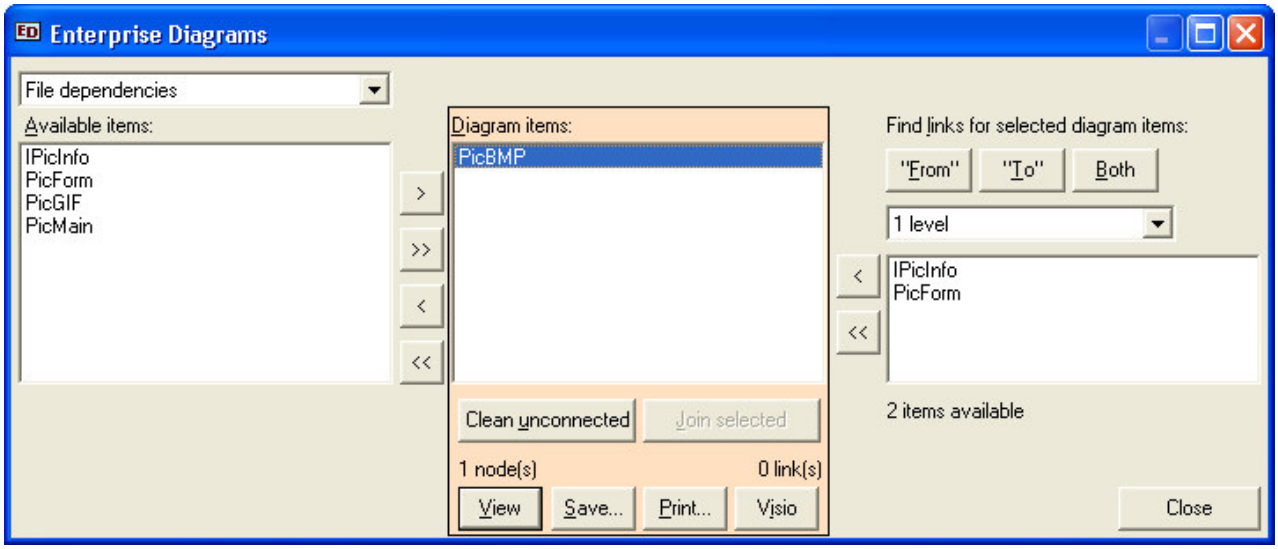
- Procedure calls – shows all calls
- Procedure calls, grouped by module – groups procedures by their module and shows all calls
- Procedure calls, cross-module only – groups procedures by their module and shows calls between the modules

Project dependencies. This diagram shows dependencies between several projects. It is useful with a multi-project analysis.

File belongs to project. This option lets you view which files belong to which project in a multi-project analysis. Remember to select the project file(s) into the diagram, otherwise there are no links.

Cohesion. The Cohesion diagram displays the procedures of a module and also the variables they use in that same module. The diagram lets you understanding the procedure-to-procedure and procedure-to-variable relationships within a single module.

Creating charts with Enterprise Diagrams



You create Enterprise Diagrams by adding items in the *Diagram items* list, in the middle of the dialog. When you've done with your selection, press the View button and a diagram shows up. You can also Save to a file, Print the diagram or export to Microsoft Visio. For Visio export to work you need Visio 2002 or 2003 installed on the same computer as Project Analyzer.

It is often impractical to visualize a large program in a single diagram. If the diagram is too complex, it may be impossible to create, read or print. Therefore, don't just drag all available items into the diagram and expect it to work like magic. Add some key items first, such as the main form or a central class. Select these items in the *Diagram items* list. Now you can find connected items via the "Find links" feature on the right. Select the number of link levels you wish to find and press "From" to find incoming links, "To" to find outgoing links and Both to find both "From" and "To" links. Then drag the items found to the middle.

To remove unnecessary content from the diagram, press Clean unconnected. This will remove unlinked nodes from the graph, but keep all linked ones. Unlinked items frequently resemble something uninteresting, such as a function that is not in use by the code in question.

You can also use the Join selected button. This will make a complex chart simpler by joining the selected items and hiding any links between them. You can use Join selected to group things together. As an example, you can group related forms to visualize connections between form groups instead of individual forms.

6 Advanced analyses in the Enterprise Edition

The Enterprise Edition supports a number of advanced analysis techniques for source and binary files. The features described in this chapter require an Enterprise Edition license. In the demo and Standard/Pro Editions, they are enabled for small analysis with a maximum of 10 source files.

6.1 Multi-project analysis

Feature requires Enterprise Edition.

Larger systems may consist of several projects, for example, a standard executable project, a .dll library project, and several .ocx component projects. When you need to find out the dependencies between several projects, you need multi-project analysis.

Project Analyzer Enterprise Edition supports analysis of several projects at a time. The Enterprise Edition is required if you want to

- analyze several projects (.vbp, .vbproj) in one run
- analyze .vbg or .sln files
- discover dependencies between several projects (.vbp, .vbproj)
- show up dead code in source code files shared by several projects (in the Pro Edition, this analysis can be done with Super Project Analyzer)

Multi-project analysis is essential for many web systems, mixed language systems and any other systems consisting of several projects. It is capable of analyzing hundreds of projects in one run. Without multi-project analysis you are limited to analyzing one project at a time.

You start a multi-project analysis simply by choosing a .vbg or .sln file to analyze. You can also add new projects to the analysis by pressing the Add project(s) button in the dialog that pops up when you have selected the project to analyze.

6.2 COM file analysis

Feature requires Enterprise Edition.

COM analysis reads information stored in a type library in a COM file. A type library describes the public interface that is available for others to call. It stores information on public classes, methods, properties, parameters, events, user-defined types, constants, Enums and aliases. Unions, which cannot be used in VB, are not shown.

When you include a COM file in your analysis, its contents appear in Project Analyzer as regular classes, procedures, events, Enums etc. For example, the summary report will include COM procedures in its procedures count. Because of this, you might not always want to include COM files in your analyses. COM analysis is most useful for files that you or your colleagues wrote.

The COM information you get is limited to the public interface. COM analysis doesn't reveal anything about the internals, such as private procedures that actually implement the interface. Thus, when you have the source code for a COM file available, it's better to analyze the source rather than the COM file. The information available in a COM file is limited to the interface it exposes to other projects. However, by analyzing the source code you can also understand the internals.

Select the desired COM files when starting the analysis. Note that you need the Enterprise Edition for this feature. You can also get a demo for analyses with max 10 source files.

6.3 DLL file analysis

Feature requires Enterprise Edition.

DLL analysis works with conventional library files. In Visual Basic you use DLLs via the Declare statement and also the <DLLImport> attribute in VB.NET. DLL analysis reads the contents of library files and also displays declare syntax information.

DLL analysis reveals all available procedures in a conventional library file. A built-in API database shows the declaration format and operating system compatibility of a large number of Windows API functions.

Note that you need the Enterprise Edition for this feature. You can also get a demo for analyses with max 10 source files.

Reading the libraries

DLL analysis reads the function export table of a PE (Portable Executable) file. The export table lists the procedures available in the library. You need to enable this feature when you open the file(s) to analyze.

Any 32-bit PE file that has a function export table can be analyzed. It's required that this file is referenced in the source via a Declare statement or via a <DLLImport> attribute. The analysis is not limited to files with the .dll extension, it also works with .exe and .ocx files, provided that they export functions. A typical case for DLL analysis is when you call the Windows API routines.

Enable analysis of DLL contents by checking the appropriate option when opening a project for analysis.

DLL declare format

For a large number of Win32 API functions, Project Analyzer displays the Declare statement syntax as given by Microsoft. This feature is useful if you want to compare an existing Declare statement with the canonical declaration or if you want to learn how to declare and use an undeclared function.

Note that it's perfectly all right for your own declarations to differ from the canonical declaration. You can declare the same procedure in a number of ways, varying the parameter data types, parameter passing convention, parameter names and the procedure name.

To view the available declarations, click on a procedure in a library file. You will see the declarations listed. The canonical declaration, if available, is given first. If there are 2 or more declarations, your declarations differ from the canonical declaration and/or from each other. In this case, check the procedure call tree at the bottom left of the main window to find your own declarations.

DLL analysis related problem detection

You can monitor DLL related issues with the problem detection feature.

Procedure not found in library. A procedure is declared, the library exists on your system, but there is no such a procedure in the file. You have declared a non-existing procedure or the library file you have is not the required version. If you execute this procedure call on this machine, you will most probably get a run-time error. However, the code may still work on another computer with the correct library version available.

Return value discarded. A DLL procedure is declared as a Sub instead of a Function. The function returns a value but it is ignored by the declaration. Although it is not necessary to actually use a function's return value, simply ignoring it could indicate a problem in the caller's logic. Review the documentation of the library to determine what the type of the return value is and if it can be ignored safely.

6.4 Find duplicate code

Feature requires Enterprise Edition.

Duplicate code analysis searches for repeated code blocks, including identical procedures and shorter code snippets. You can find this analysis in the Enterprise menu. It requires the Enterprise Edition.

Why be concerned about duplicate code?

Copy & paste coding can result in the same code being repeated at different locations with few or no modifications. Duplicate code analysis finds blocks of code that exist in two or more files across the project or solution. A duplicate block can consist of a few lines, complete procedures or even entire modules. You may also find repeated declarations for constants, API procedures or user-defined types.

You can use the duplicate code analysis feature to eliminate repeated code and detect possible logical errors. The elimination of unnecessary duplicates yields the following benefits.

- Less code to test, maintain and document.
- Maintenance efforts can focus on a single location instead of making the same changes at multiple locations.
- Fewer errors. If the same code is repeated at several locations, changes and fixes at one location don't get propagated to the other locations.
- Smaller executable size.

You can eliminate repeated code snippets by turning them into procedures. If entire procedures are repeated, you can usually delete the duplicate(s) and keep just one of them.

Sometimes you may find repeated code at a place where different code should actually have been used. Two routines that should perform a different thing may inadvertently contain the same code.

How the duplicate code analysis works

The analysis compares the lines of two files with each other. If it finds an exact match longer than a predefined minimum number of lines, it is reported as a duplicate block. Any difference, even if it's just one character, will end the duplicate block.

White space (empty lines and indentation) is ignored. Two blocks that only differ by the use of indentation or empty lines are considered similar. You can optionally ignore comments, line numbers and line labels, which are not really executable code.

String literal analysis

There is a related analysis in the Report menu that handles string literals. It's called String literal analysis and it finds duplicate string definitions within the code. Use it to eliminate duplicate definitions of the same text. That feature is available in all editions.

6.5 VB.NET Compatibility Check

Feature requires Enterprise Edition.

Visual Basic .NET is very different from earlier versions of VB. Migrating existing VB code to VB.NET is not just a matter of loading it to the new version. VB.NET does include an upgrade wizard. It does a good job converting your syntax, but that's just half of the work. Your project will not compile right away, you'll have a lot of issues ahead. If you weren't prepared, you might decide you didn't want to migrate after all.

Fortunately, you can prepare your code in advance. Actually, it is essential to work on your existing code before you load it in VB.NET. This is where the Enterprise Edition comes to help. VB.NET Compatibility Check consists of a set of problem detection rules that help you change your existing code before the migration, and get notified of possible major issues that affect your decision whether or not to migrate. The check works with classic VB projects.

To use the Compatibility Check, simply analyze a VB classic project and enable the checks via Problem Options in the Options menu. You can use any of the 2 predefined Compatibility Check filters or define one of your own.

Compatibility issues

A typical compatibility issue cannot be resolved by a computer. Human attention is required. Therefore, you need to get prepared for manual work. The available types listed in the below table.

Feature not upgradable	VB.NET does not support this feature. You should either consider leaving the code in your current Visual Basic version, or prepare for major upgrade work. Example: web programming.
Fix required before upgrade	VB.NET does not support this feature. You will have to rewrite your code before you upgrade to VB.NET. Example: data binding.
Fix recommended before upgrade	You will save work if you adjust your code or forms before you upgrade. Example: correct declaration of variables and class members.
Can be fixed before or after upgrade	The code needs to be changed. You may do it now or after the upgrade. Example: explicit declaration of procedure parameters.
Work required after upgrade	The code needs to be reviewed or changed, but you can't do it before the upgrade. Example: syntax changes, changes in control object models, unsupported functions requiring complete redesign where used.

You can find a description of each individual problem by right-clicking it in Project Analyzer. The descriptions are also listed in the help file.

You probably get a long list of compatibility issues. It is advisable to first use a problem filter that selects the most critical issues. Pay special attention to those problems that are classified as Feature not upgradable, Fix required before upgrade or Fix recommended before upgrade. These classes are indicated with either a red or a yellow problem icon.

You can get a summary of the upgrade work by generating a VB.NET Compatibility Report. You can find this report in the Pro menu. This report tells you if the project is upgradable to VB.NET. It also estimates the size of the upgrade work. Selection of the problem filter (in Problem Options) does not affect this report.

Project Analyzer Enterprise Edition does not check for all incompatibilities between VB6 and VB.NET. It provides a limited set of compatibility checks. When you upgrade to VB.NET, you are likely to find more incompatibilities. The VB.NET help file describes migration issues in detail.

6.6 Macros

Feature requires Enterprise Edition.

Have Project Analyzer work for you while you sleep or drink coffee! If you need to do the same analyses and again and again, try the macro feature. The macros are simple plaintext .pam files that instruct Project Analyzer to open projects, analyze them and perform some operations such as reporting and auto-fix.

The macro syntax is described in the help file. You can run the macros via the Enterprise menu, the command line or a batch file. Notice that the use of macros always requires the Enterprise Edition – they are not available in the demo or the Standard/Pro Editions regardless of analysis size.

6.7 Project Metrics

Feature requires Enterprise Edition.

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. They aren't very sophisticated, but they're easy. Project Analyzer knows more. It can tell you about the *understandability*, *complexity* and *reusability* of your code.

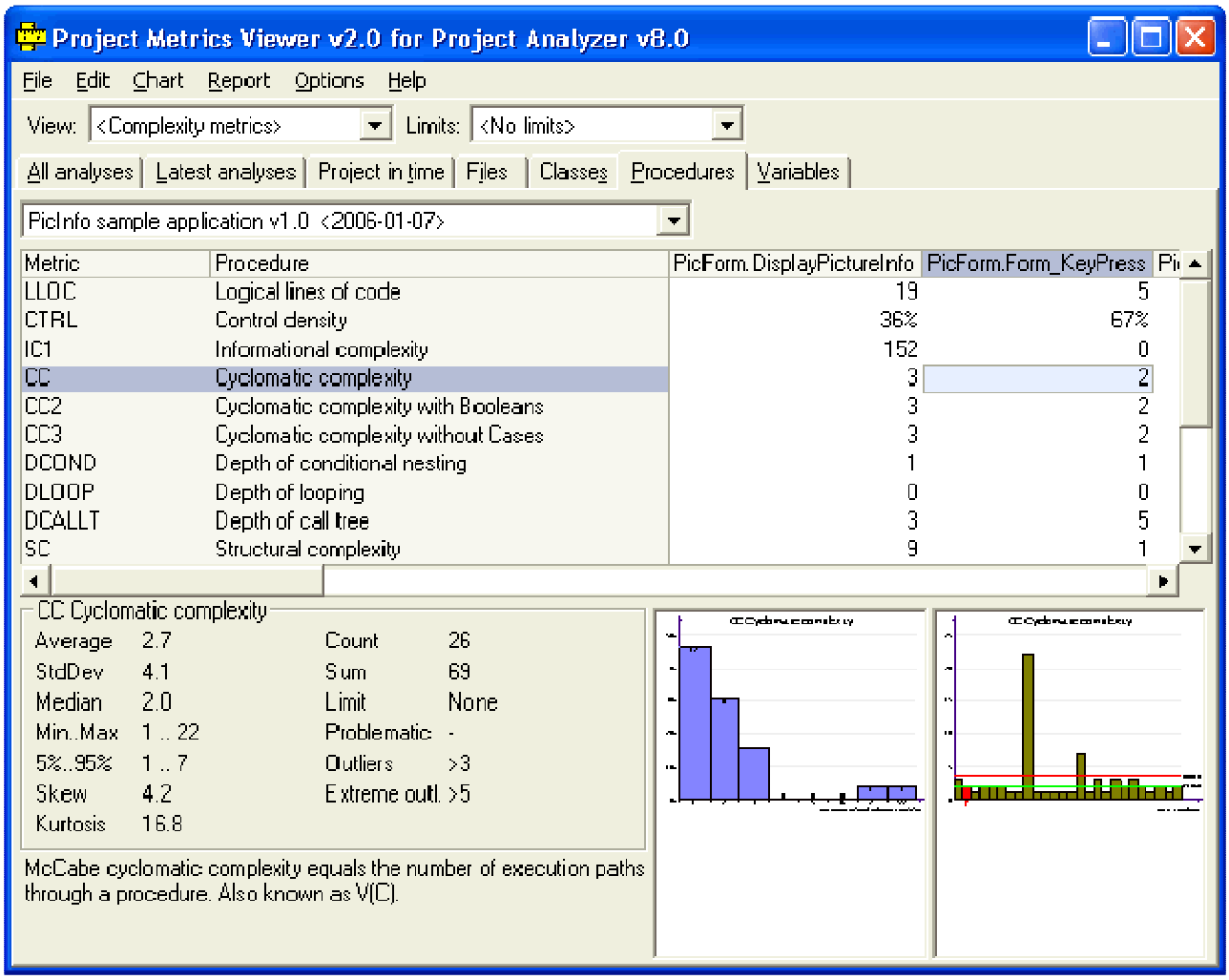
To get the most out of metrics, you need the Enterprise Edition and the Project Metrics feature. Project Metrics provides more than a hundred different metrics. You can find comprehensive instructions to metrics and their use in the help file. Some of the available metrics are:

- Size metrics, such as lines of code and number of methods.
- Complexity metrics, such as McCabe cyclomatic complexity, cyclomatic density, depth of conditional nesting, structural fan-in/fan-out, informational complexity, class hierarchy metrics.
- Understandability metrics, such as length of names and amount of comments.

Getting metrics for your program

1. Analyze a project, project group or solution.
2. Save metrics by choosing the appropriate command in the Enterprise menu. This will create a new .mtx database file that stores the values for later use.
3. Run Project Metrics Viewer (mtxview.exe). You can find it in the Enterprise menu. The viewer allows you to view the stored metrics, compare projects and follow the development over time.

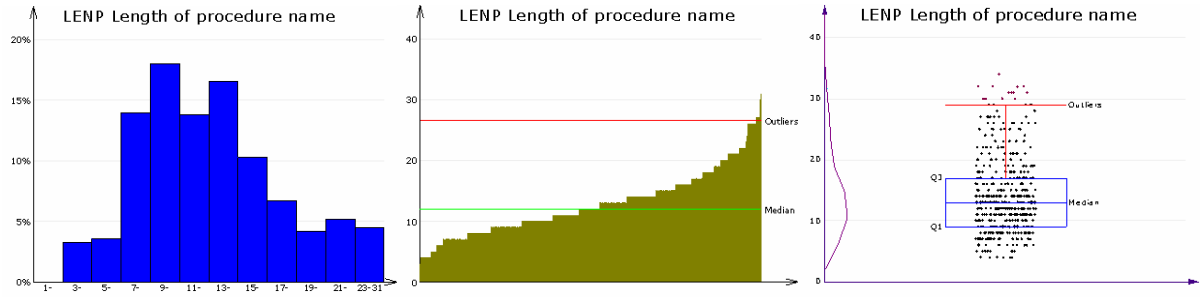
Project Metrics Viewer



The viewer is your user interface to the .mtx files on your computer. Use it to understand the status of a project, track development between versions and compare programs.

What are the target values of the metrics? This is the question that every metrics user faces. Some metrics have recommended ranges and anything below or above the range indicates a problem. Other metrics have no such values. Then you can watch for exceptionally low or high values to detect problematic code.

We recommend that you pick a suite of metrics that suits your use. Analyze your projects to save metrics regularly, and you can follow how your projects develop.



Sample metrics charts

List of supported metrics

Project metrics

DATE	Project date	Date of newest file in project.
DAYS	Days passed	Days passed between versions.

LINES	Physical lines	Physical source lines, including code, comments, empty comments and empty lines. This metric is what you would see with a simple text editor or line count program.
LLINES	Logical lines	Logical source lines, including code, comments, empty comments and empty lines. One logical line may be split on several physical lines by a line continuation character. LLINES=LLOC + LLOC' + LLOW
LLOC	Logical lines of code	Code lines count. One logical line may be split on several physical lines by a line continuation character.
LLOC'	Logical lines of comment	Comment lines count. All full-line comments count, even if empty. End-of-codeline comments not included. One logical line may be split on several physical lines by a line continuation character.
LLOW	Logical lines of whitespace	Logical lines of whitespace. This is mostly equal to physical lines of whitespace, that is lines that have no other content than spaces, tabs and the line continuation character.
LLOC%	Code percentage	Percentage of code lines. Counted from logical lines. LLOC%=LLOC/LLINES
LLOC'%	Comment percentage	Percentage of comment lines. Counted as full-line comments per logical lines. LLOC'%=LLOC/LLINES
LLOW%	Whitespace percentage	Percentage of whitespace lines. Counted from logical lines. LLOW%=LLOW/LLINES
MCOMM	Meaningful comments	Full-line and end-of-line comments that have meaningful content.
MCOMM%	Comment density	Meaningful comments divided by number of logical lines of code. MCOMM%=MCOMM/LLOC
kB	Project size	Project size in kilobytes. Includes all source files included in the analysis, excluding referenced files.
DATEF	Average file date	Average file date.
STMT	Number of statements	Total number of all statements.
STMTd	Declarative statements	Number of declarative statements, which are are: procedure headers, variable and constant declarations, all statements outside procedures.
STMTx	Executable statements	Number of executable statements. A statement is executabe if it is not declarative. Executable statements can only exist within procedures. STMTx=STMT-STMTd
STMTc	Control statements	Number of control statements. A control statement is an executable statement that can alter the order in which statements are executed.
STMTnc	Non-control statements	Number of non-control statements, which are executable statements that are neither control nor declarative statements. STMTnc=STMTx-STMTc
XQT	Executability	Executability measures the amount of executable statements. It equals the number of executable statements divided by the number of all statements. XQT=STMTx/STMT
CTRL	Control density	Control density measures the amount of control statements. It equals the number of control statements divided by the number of all executable statements. CTRL=STMTc/STMTx
FILES	Number of files	Number of files in project.
PROCS	Number of procedures	Number of procedures, including subs, functions, property blocks, API declarations and events.
VARS	Number of variables	Number of variables, including arrays, parameters and local variables.
CONSTS	Number of consts	Number of constants, excluding enum constants.
UDTS	Number of user-defined types	Number of user-defined types, that is, Type and/or Structure statements.
ENUMS	Number of Enums	Number of enumeration names.
ENUMCS	Number of Enum constants	Number of enumeration constant names.
VARSgm	Global and module-level variables	Total number of global and module-level variables and arrays.
VARSloc	Local variables	Total number of procedure local variables and arrays, excluding parameters.
FORMS	Number of forms	Number of real forms excluding any usercontrols.
MDLS	Number of standard modules	Number of standard modules: .bas files and Module blocks.
dPROCS	Number of dead procedures	Number of unused procedures.
dVARS	Number of dead variables	Number of unused variables.
dCONSTS	Number of dead consts	Number of unused constants.
dLINES	Dead lines	Physical lines in dead procedures.
DEAD	Deadness index	Evaluates the average percentage of dead code. DEAD=(dPROCS + dVARS + dCONSTS) / (PROCS + VARS + CONSTS)
LEN	Length of names	Average length of names defined in VB files, including everything in LENV, LENC and LENP, plus Types, Enums, Type fields, Enum constants, controls, classes, structures and modules.
LENV	Length of variable and parameter names	Average length of names of variables, arrays and parameters defined in VB files, excluding parameters in event handlers and implementing procedures.
LENC	Length of constant names	Average length of all constant names defined in VB files.
LENP	Length of procedure names	Average length of procedure names defined in VB files, excluding event handlers and implementing procedures. Each property is counted only once. This metric may differ from the other LENP, which is defined for a slighly different set of procedures.
UNIQ	Name uniqueness ratio	Number of unique names divided by the total number of names. All the names in LEN are counted.
ENUMSZ	Average Enum size	Average number of constants in an Enum block. ENUMSZ=ENUMCS/ENUMS

ENUMR	Enum ratio	The percentage of Enum constants among all constants. ENUMR=ENUMCS/(ENUMCS+CONSTS)
DECDENS	Decision density	Indicates the density of decision statements in the code. Calculated as sum of procedural cyclomatic complexity divided by logical lines of code. DECDENS=Sum(CC)/LLOC
TCC	Total cyclomatic complexity	Total cyclomatic complexity equals the total number of decisions + 1. TCC=Sum(CC)-Count(CC)+1
SYSC	System complexity	Sum of SYSC over all procedures. Measures the total complexity of a project. SYSC=Sum(SYSC) over all procedures
RSYSC	Relative system complexity	Average system complexity among procedures. RSYSC=avg(SYSC)
CALLS	Number of procedure calls	Number of procedure call statements, including calls to subs, functions and declares, accesses to properties and the raising of events. Implicit calls (such as Form_Load) are not counted.
CALLDENS	Call density	Average number of calls on a code line. Measures the modularity or structuredness. CALLDENS=CALLS/LLOC
maxDCALLT	Maximum depth of call tree	The depth of the largest call tree in the system: number of levels in the tree. maxDCALLT=Max(DCALLT)
maxSCALLT	Maximum size of call tree	The size of the largest call tree in the system: number of distinct procedures in the tree. maxSCALLT=Max(SCALLT)
RB	Reuse benefit	Reuse benefit RB is the extent to which you reuse your procedures. A procedure that is being called at several locations is said to be reused. A procedure that is called just once or not at all is not reused. RB measures the overall amount of reuse in the entire system.
Rc	Reuse of constants	The average number of times you reuse your constants and enum constants. Rc=uses/count - 1
CLS	Number of classes	Number of classes defined in project.
ROOTS	Number of root classes	Number of root classes defined in project.
LEAFS	Number of root classes	Number of leaf classes defined in project. A leaf class has no descendants.
INTERFS	Number of Interface definitions	Number of Interfaces defined in project.
maxDIT	Maximum depth of inheritance tree	maxDIT is the depth of the deepest inheritance tree. maxDIT=max(DIT)
CLSa	Number of abstract classes	Number of abstract classes defined in project. In VB.NET, an abstract class is declared MustOverride. In VB Classic, it's a skeleton class that defines an interface class for Implements.
CLSc	Number of concrete classes	Number of concrete classes defined in project. A concrete class is one that is not abstract (see CLSa). CLSc=CLS-CLSa
U	Reuse ratio	Reuse ratio for classes. A class is reused if it has descendants. U=(CLS - LEAFS) / CLS
S	Specialization ratio	Specialization ratio for classes. A class is specialized if it inherits from a parent class. In a project without superclasses, S is undefined. S=subclasses/superclasses
MIF	Method inheritance factor	The sum of inherited methods divided by the total number of methods in a project.
AIF	Attribute inheritance factor	The sum of inherited variables divided by the total number of variables in a project.
MHF	Method hiding factor	Measures how class methods are hidden from other classes.
AHF	Attribute hiding factor	Measures how class attributes (variables) are hidden from other classes.
PF	Polymorphism factor	Percentage of actual polymorphic definitions of all possible polymorphic definitions. Also known as POF.
CF	Coupling factor	Measures the actual couplings among classes in relation to the maximum number of possible couplings. Also known as COF.
OHEF	Operation hiding effectiveness factor	Classes that do access operations / Classes that can access operations.
AHEF	Attribute hiding effectiveness factor	Classes that do access attributes / Classes that can access attributes.
IIF	Internal inheritance factor	The relative amount of internal inheritance. Internal inheritance happens when a class inherits another class in the same system (not an external class).
PPF	Parametric polymorphism factor	Percentage of parametrized classes (generic classes).
TREADS	Total variable reads	Number of read instructions from global and module-level variables. TREADS=Sum(READS)
TWRITES	Total variable writes	Number of write instructions to global and module-level variables. TWRITES=Sum(WRITES)
TRW	Total variable reads+writes	Number of reads from and writes to global and module-level variables. TRW=TREADS+TWRITES
DATADENS	Data access density	Average number of variable access instructions per logical line of code. DATADENS=TRW/LLOC
IOg%	Global I/O ratio	Amount of data flow via global and module-level variables versus procedure parameters and function return values. IOg%=Sum(IOg) / Sum(IOg+IOp)
		File metrics
LINES	Physical lines	Physical source lines, including code, comments, empty comments and empty lines. This metric is what you would see with a simple text editor or line count program.
LLINES	Logical lines	Logical source lines, including code, comments, empty comments and empty lines. One logical line may be split on several physical lines by a line continuation character. LLINES=LLOC + LLOC' + LLOW

LLOC	Logical lines of code	Code lines count. One logical line may be split on several physical lines by a line continuation character.
LLOC'	Logical lines of comment	Comment lines count. All full-line comments count, even if empty. End-of-codeline comments not included. One logical line may be split on several physical lines by a line continuation character.
LLOW	Logical lines of whitespace	Logical lines of whitespace. This is mostly equal to physical lines of whitespace, that is lines that have no other content than spaces, tabs and the line continuation character.
LLOC%	Code percentage	Percentage of code lines. Counted from logical lines. LLOC%=LLOC/LLINES
LLOC'%	Comment percentage	Percentage of comment lines. Counted as full-line comments per logical lines. LLOC'%=LLOC/LLINES
LLOW%	Whitespace percentage	Percentage of whitespace lines. Counted from logical lines. LLOW%=LLOW/LLINES
MCOMM	Meaningful comments	Full-line and end-of-line comments that have meaningful content.
MCOMM%	Comment density	Meaningful comments divided by number of logical lines of code. MCOMM%=MCOMM/LLOC
kB	File size	File size in kilobytes.
DATEF	File date	File date.
PROCS	Number of procedures	Number of procedures, including subs, functions, property blocks, API declarations and events.
VARS	Number of variables	Number of variables, including arrays, parameters and local variables.
CONSTS	Number of consts	Number of constants, excluding enum constants.
SFIN	Structural fan-in	Number of files that use a file.
SFOUT	Structural fan-out	Number of files that a file uses.
STMT	Number of statements	Total number of all statements.
STMTd	Declarative statements	Number of declarative statements, which are are: procedure headers, variable and constant declarations, all statements outside procedures.
STMTx	Executable statements	Number of executable statements. A statement is executabe if it is not declarative. Executable statements can only exist within procedures. STMTx=STMT-STMTd
STMTc	Control statements	Number of control statements. A control statement is an executable statement that can alter the order in which statements are executed.
STMTnc	Non-control statements	Number of non-control statements, which are executable statements that are neither control nor declarative statements. STMTnc=STMTx-STMTc
XQT	Executability	Executability measures the amount of executable statements. It equals the number of executable statements divided by the number of all statements. XQT=STMTx/STMT
CTRL	Control density	Control density measures the amount of control statements. It equals the number of control statements divided by the number of all executable statements. CTRL=STMTc/STMTx
		Class metrics
WMC	Weighted Methods Per Class	Number of subs, functions and property procedures in class.
DIT	Depth of Inheritance Tree	Number of ancestor classes.
NOC	Number of Children	Number of immediate sub-classes that inherit from a class.
CBO	Coupling between Object Classes	Number of classes to which a class is coupled. Coupling is defined as method call or variable access.
RFC	Response for a Class First step	Number of methods that can potentially be executed in response to a message received a class. Counts only the first level of the call tree.
RFC'	Response for a Class	Number of methods that can potentially be executed in response to a message received a class. Counts the full call tree.
LCOM1	Lack of Cohesion of Methods (1)	A zero value indicates a cohesive class. A positive value indicates a class that should be split. Also known as LCOM. Defined by Chidamber & Kemerer.
LCOM2	Lack of Cohesion of Methods (2)	The percentage of methods that do not access a specific attribute averaged over all attributes in the class.
LCOM3	Lack of Cohesion of Methods (3)	Also known as LCOM*. Defined by Henderson-Sellers. Values of 1 and greater are considered extreme lack of cohesion.
LCOM4	Lack of Cohesion of Methods (4)	Defined by Hitz & Montazeri. Value 1 indicates a good, cohesive class. Values 2 and greater are considered bad (lack of cohesion). Such a class should be split. LCOM4 is more suitable for VB than the other LCOMx variants.
TCCi	Tight Class Cohesion	TCCi is 'TCC with inheritance'. TCC tells the connection density of the methods in a class. TCC varies from 0 (totally non-cohesive) to 1 (maximally cohesive).
LCCi	Loose Class Cohesion	LCCi is 'LCC with inheritance'. LCC describes the connectedness of the methods in a class. LCC<1 indicates a non-cohesive class. LCC=1 indicates a cohesive class."
TCCl	Tight Class Cohesion (local)	TCCl is 'TCC without inheritance'. TCC tells the connection density of the methods in a class. TCC varies from 0 (totally non-cohesive) to 1 (maximally cohesive).
LCCl	Loose Class Cohesion (local)	LCCl is 'LCC without inheritance'. LCC describes the connectedness of the methods in a class. LCC<1 indicates a non-cohesive class. LCC=1 indicates a cohesive class."
MPC	Message-Passing Coupling	Number of procedure calls going outside of a class. Each call is counted once, whether it's early bound, late bound or polymorphic.
LINES	Physical lines	Physical source lines, including code, comments, empty comments and empty lines. This metric is what you would see with a simple text editor or line count program.

LLINES	Logical lines	Logical source lines, including code, comments, empty comments and empty lines. One logical line may be split on several physical lines by a line continuation character. LLINES=LLOC + LLOC' + LLOW
LLOC	Logical lines of code	Code lines count. One logical line may be split on several physical lines by a line continuation character.
LLOC'	Logical lines of comment	Comment lines count. All full-line comments count, even if empty. End-of-codeline comments not included. One logical line may be split on several physical lines by a line continuation character.
LLOW	Logical lines of whitespace	Logical lines of whitespace. This is mostly equal to physical lines of whitespace, that is lines that have no other content than spaces, tabs and the line continuation character.
STMT	Number of statements	Total number of all statements.
STMTd	Declarative statements	Number of declarative statements, which are are: procedure headers, variable and constant declarations, all statements outside procedures.
STMTx	Executable statements	Number of executable statements. A statement is executabe if it is not declarative. Executable statements can only exist within procedures. STMTx=STMT-STMTd
STMTc	Control statements	Number of control statements. A control statement is an executable statement that can alter the order in which statements are executed.
STMTnc	Non-control statements	Number of non-control statements, which are executable statements that are neither control nor declarative statements. STMTnc=STMTx-STMTc
XQT	Executability	Executability measures the amount of executable statements. It equals the number of executable statements divided by the number of all statements. XQT=STMTx/STMT
CTRL	Control density	Control density measures the amount of control statements. It equals the number of control statements divided by the number of all executable statements. CTRL=STMTc/STMTx
IMPL	Implemented interfaces	Number of interfaces implemented by class.
WMCnp	Non-private methods defined by class	WMC excluding private methods.
WMCi	Methods defined and inherited by class	WMC including inherited methods.
VARS	Variables defined by class	Number of variables defined by class. Does not include inherited variables.
VARSnp	Non-private variables	Number of non-private variables defined by class. VARS excluding private variables.
VARSi	Variables defined+inherited	Number of variables defined and inherited by class.
EVENTS	Events	Events defined by class. This metric coutns the event definitions, not event handlers.
CTORS	Constructors	Constructors defined by class. VB.NET Sub New is a constructor, whereas VB Classic Class_Initialize is an event.
CSZ	Class size	Size of class measured by number of methods and variables. CSZ=WMC + VARS
CIS	Class interface size	Size of class interface measured by number of non-private methods and variables. CIS=WMCnp + VARSnp
TCC	Total cyclomatic complexity	Total cyclomatic complexity equals the total number of decisions + 1. TCC=Sum(CC)-Count(CC)+1
		Procedure metrics
LINES	Physical lines	Physical source lines, including code, comments, empty comments and empty lines. This metric is what you would see with a simple text editor or line count program.
LLINES	Logical lines	Logical source lines, including code, comments, empty comments and empty lines. One logical line may be split on several physical lines by a line continuation character. LLINES=LLOC + LLOC' + LLOW
LLOC	Logical lines of code	Code lines count. One logical line may be split on several physical lines by a line continuation character.
LLOC'	Logical lines of comment	Comment lines count. All full-line comments count, even if empty. End-of-codeline comments not included. One logical line may be split on several physical lines by a line continuation character.
LLOW	Logical lines of whitespace	Logical lines of whitespace. This is mostly equal to physical lines of whitespace, that is lines that have no other content than spaces, tabs and the line continuation character.
MCOMM	Meaningful comments	Full-line and end-of-line comments that have meaningful content.
LLOCt	Lines in call tree	Logical lines of code in call tree. The number of lines that may potentially execute in a call to this procedure. Includes all procedures that may execute.
PARAMS	Procedure parameters	Number of formal parameters defined in procedure header.
VARSloc	Local variables	Number of procedure local variables and arrays, excluding parameters.
STMT	Number of statements	Total number of all statements.
STMTd	Declarative statements	Number of declarative statements, which are are: procedure headers, variable and constant declarations, all statements outside procedures.
STMTx	Executable statements	Number of executable statements. A statement is executabe if it is not declarative. Executable statements can only exist within procedures. STMTx=STMT-STMTd
STMTc	Control statements	Number of control statements. A control statement is an executable statement that can alter the order in which statements are executed.
STMTnc	Non-control statements	Number of non-control statements, which are executable statements that are neither control nor declarative statements. STMTnc=STMTx-STMTc
XQT	Executability	Executability measures the amount of executable statements. It equals the number of executable statements divided by the number of all statements. XQT=STMTx/STMT

CTRL	Control density	Control density measures the amount of control statements. It equals the number of control statements divided by the number of all executable statements. CTRL=STMTc/STMTx
IOg	Global I/O	Number of global and module-level variables accessed by a procedure.
IOp	Parameter I/O	Number of parameters used or returned by a procedure. The function return value counts as one parameter (output parameter).
IOvars	Input and output variables	Total number of input and output variables for a procedure, including parameters and function return value. IOvars=IOg+IOp
IFIN	Informational fan-in	Amount of data read.
IFOUT	Informational fan-out	Amount of data written.
IFIO	Informational fan-in x fan-out	Fan-in multiplied by fan-out. IFIO=IFIN*IFOUT
IC1	Informational complexity	Fan-in multiplied by fan-out multiplied by procedure length (logical lines of code). IC1=IFIO*LLOC
CC	Cyclomatic complexity	McCabe cyclomatic complexity equals the number of execution paths through a procedure. Also known as V(C).
CC2	Cyclomatic complexity with Booleans	CC2 equals the regular CC metric but each Boolean operator within a branching statement causes complexity to increase by one. Also called Extended cyclomatic complexity (ECC).
CC3	Cyclomatic complexity without Cases	CC3 equals the regular CC metric, but each Select Case block is counted as one branch, not as multiple branches.
DCOND	Depth of conditional nesting	Maximum number of nested conditional statements in a procedure.
DLOOP	Depth of looping	Maximum number of nested loop statements in a procedure.
DCALLT	Depth of call tree	Maximum number of nested procedure calls from a procedure. Recursive calls are not counted.
SCALLT	Size of call tree	Number of distinct procedures in the call tree of a procedure, not counting the procedure itself.
SC	Structural complexity	Measures the external complexity of a procedure. Equals the number of other procedures called squared. Defined by Card & Agresti, also known as S(i). Used to calculate SYSC. SC=SFOUT^2
DC	Data complexity	Measures the internal complexity of a procedure. Calculated by dividing the number of input/output variables by SFOUT+1. Defined by Card & Agresti, also known as D(i). Used to calculate SYSC. DC=IOVariables/(SFOUT+1)
SYSC	System complexity	Composite measure of complexity inside procedures and between them. Defined by Card & Agresti, also known as C(i), or design complexity. SYSC=SC+DC
LENP	Length of procedure name	Length of procedure name in characters.
SFIN	Structural fan-in	Number of procedures that call a procedure.
SFOUT	Structural fan-out	Number of procedures that a procedure calls.
		Variable metrics
READS	Reads from variable	Number of read instructions from variable.
WRITES	Writes to variable	Number of write instructions to variable.
RW	Reads and writes	Number of reads and writes. A single instruction may count both as a read and as a write. RW=READS+WRITES
FLOWS	Data flows	Number of data flows into and out of a variable. FLOWS=READS*WRITES
VARUSR	Variable users	Number of modules that use a variable.
LENVgm	Length of variable name	Length of variable name in characters.

7 Other features

The basic features of Project Analyzer are related to analysis, optimization, documentation and measurement as described in the preceding chapters. This chapter describes other features shortly. See the help file for more information on these features.

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 WinZip, PKZip and Arj. You can also get a text file listing all files in a project using this feature.

Call tree reports in the Report menu are an alternative way to document call trees. These reports can get unexpectedly large – especially the All procedures call tree report. You might prefer using the Call tree window, Project Graph or Enterprise Diagrams.

Control report is a structured listing of controls in the project. *This report is available for VB 3-6.*

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

Design quality report gives an overall picture of the quality of a project, measured by several metrics.

Dictionary report lists all names in the project in alphabetical order.

Executed by report goes back in the call tree to the top-level procedures that will, when called, eventually execute a selected procedure.

File dependency analysis finds circular file dependencies, which have a negative effect on reusability.

File dependency levels report helps you understand the dependency structure of your source files.

Hotkey conflicts report is an alternative way to view hotkey conflicts among controls and menus. Hotkey problems are also listed in the problem view. *This report is available for VB 3-6.*

Interface report lists the available interface definitions in a program and the classes that implement these interfaces.

Library report lists all DLL, TLB, OCX and VBX files and other libraries referenced by your project. This list does not include all files required by your program (when distributed), only those that are explicitly referenced either in the source code or other project options. It also shows declared API procedures, including ones are defined but not used. When DLL analysis (see page 26) is enabled in the Enterprise Edition, the report includes more details on the DLL procedures.

Menu report lists all menus and menu commands in the project. Shortcuts, hotkeys and HelpContextID's are included. Invisible and disabled menus or menu items are displayed in grey. *This report is available for VB 3-6.*

Module interface report lists the public members of a module.

Module members report lists the member procedures and variables of classes and modules. Where inheritance is used, it also includes the inherited members. There are several sub-types of this report that allow you to document your classes and modules the way you like best.

Non-cohesive classes report. This specialized report detects classes that could be split into smaller classes. The report lists class of low cohesion. These classes consist of 2 or more unrelated parts (sub-components). The parts don't call each other nor do they access common class-level variables. As they have no common functionality, they could be in separate classes.

Subsystem report. When you wish to find potentially reusable components in a large system, get the Subsystem report. It locates independent subsystems, that is, groups of files that depend on each other but no other parts of the full system. Since a subsystem is independent of other parts, in can potentially be reused in another program.

Summary report summarizes the size of a project. You can use it as a statistical information source for your efforts.

Variable use report is designed for locating hidden problems with variables. The report counts the number of reads and writes to each variable. The reads (and writes) are further classified into live, dead and dead but exposed reads (and writes). When there are no live reads from a variable, it is possibly useless. Alternatively, there is a problem that prevents the read instructions from executing. When there are no live writes to a variable, it is possibly always zero or empty. This may indicate a problem where something prevents the write instructions from executing.

View FRX files. VB Classic stores form and control properties in files with the .frx extension. There are also some other extensions serving the same purpose, such as .ctx, .pgx and .dox, but here we refer to them collectively as the

FRX files. You can view the contents of a FRX file by double-clicking it within Project Analyzer. You can even save graphics to individual files if you want to reuse a picture that you don't have available as a disk file.

View graphics files. You can double-click most graphics files, including .bmp, .gif and .jpg, to view the picture. This is especially useful with .NET web projects that include graphics as part of their web content.

Table of contents