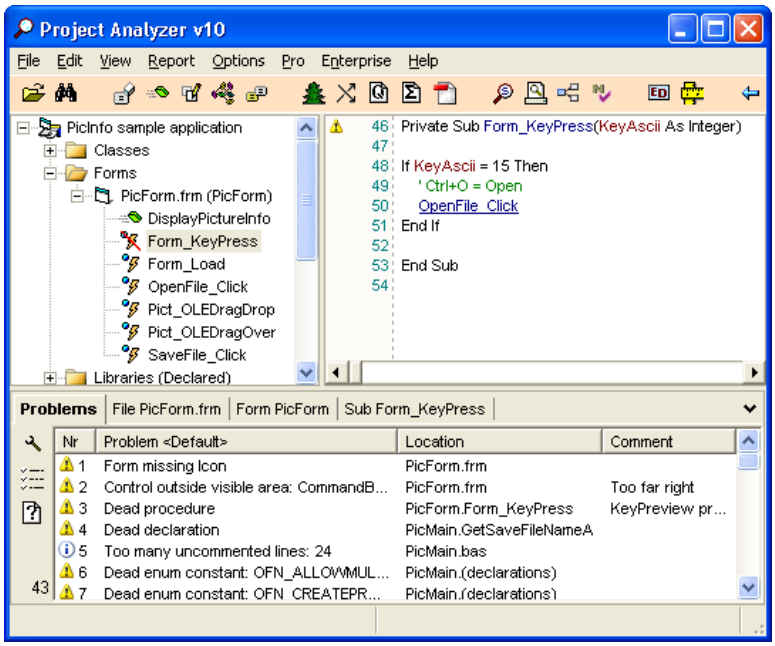
Tutorial

How to Use

Project Analyzer v10



Aivosto

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.

Suomenkielinen opas saatavissa
http://www.aivosto.com/project/tutorial-fi.pdf
Finnish translation available

www.aivosto.com
URN:NBN:fi-fe201010182609

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 v10 works with code written with Visual Basic versions 3.0, 4.0, 5.0, 6.0, VB.NET 2002, 2003, 2005, 2008, 2010, 2012 and 2013. 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!

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.

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 and viewing interactive graphs, a programmer can quickly understand how a certain function operates with other functions and variables. This helps evaluate the impact of code changes. It is also useful for understanding the migration effort from classic VB to VB.NET.

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 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, binary file analysis (.NET, COM and DLL) 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 *classic VB* to refer to Visual Basic versions from 3.0 to 6.0. With *VB.NET* we mean all the supported versions of Visual Basic .NET. In most cases, Office VBA is like Visual Basic 6.0.

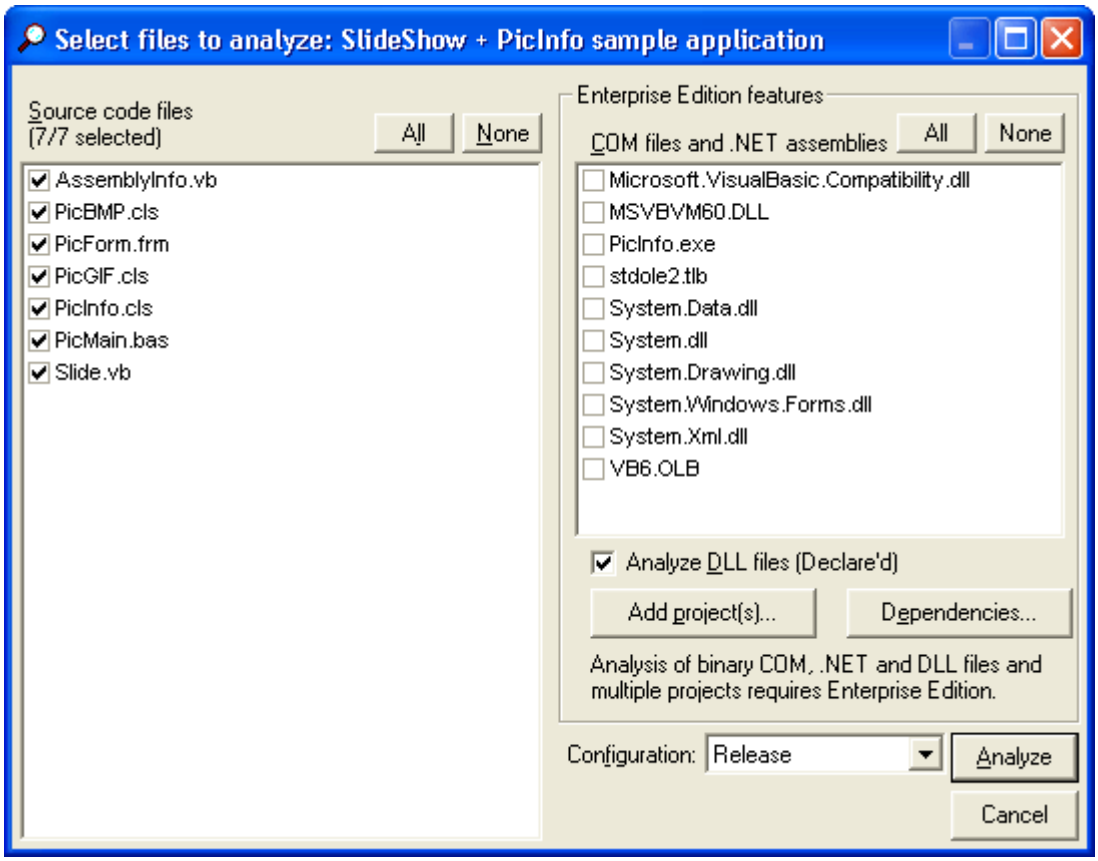
A *project* means a single program or library (.vbproj, .vbp or .mak project). Occasionally we also use it in a more general sense to cover the code currently being analyzed, be it from one or several projects. A solution and a project group mean a number of related projects (.sln and .vbg files).

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 free demo version, you can select a maximum of 10 source files at a time.

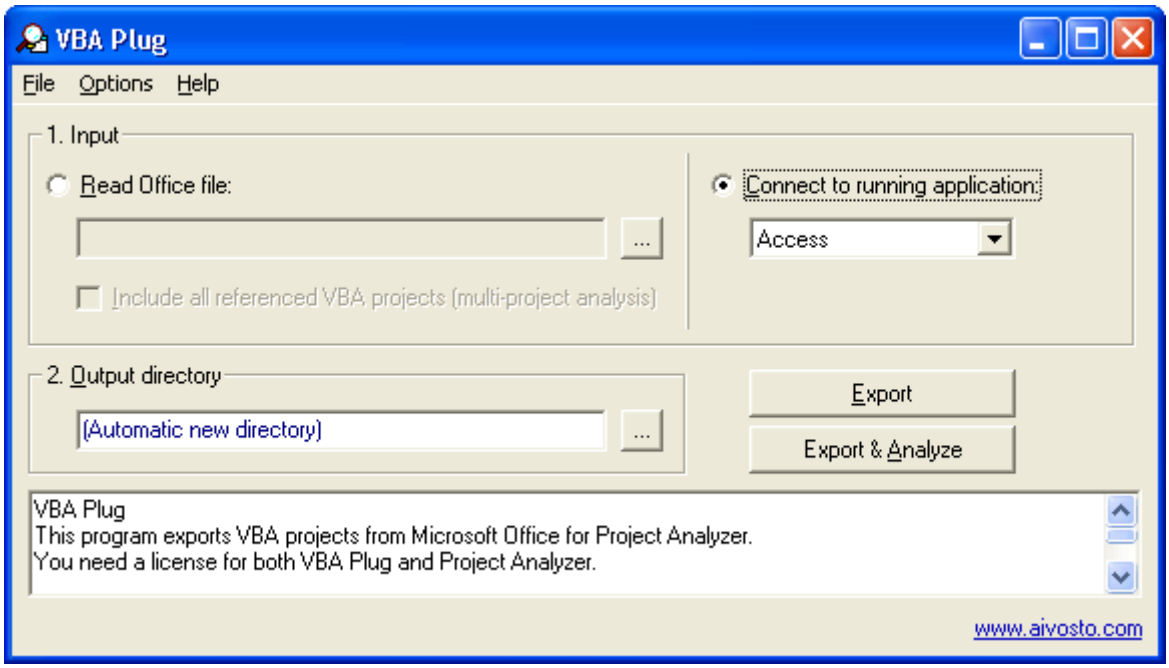
Notice that the analyses selected in the top right corner require the Enterprise Edition. They are also enabled in the demo and Standard/Pro Edition when you select a maximum of 10 source files to analyze.

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 Office VBA code, Project Analyzer requires an additional program called VBA Plug. This program is in your Project Analyzer directory (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*.



1. Input. There are two ways to select the VBA input source:

A. 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.

B. 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.

2. Output directory. After selecting the input source, define the output directory into 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 a directory for you. The automatically created directory will appear under the Windows 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*.

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

Troubleshooting VBA Plug

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.

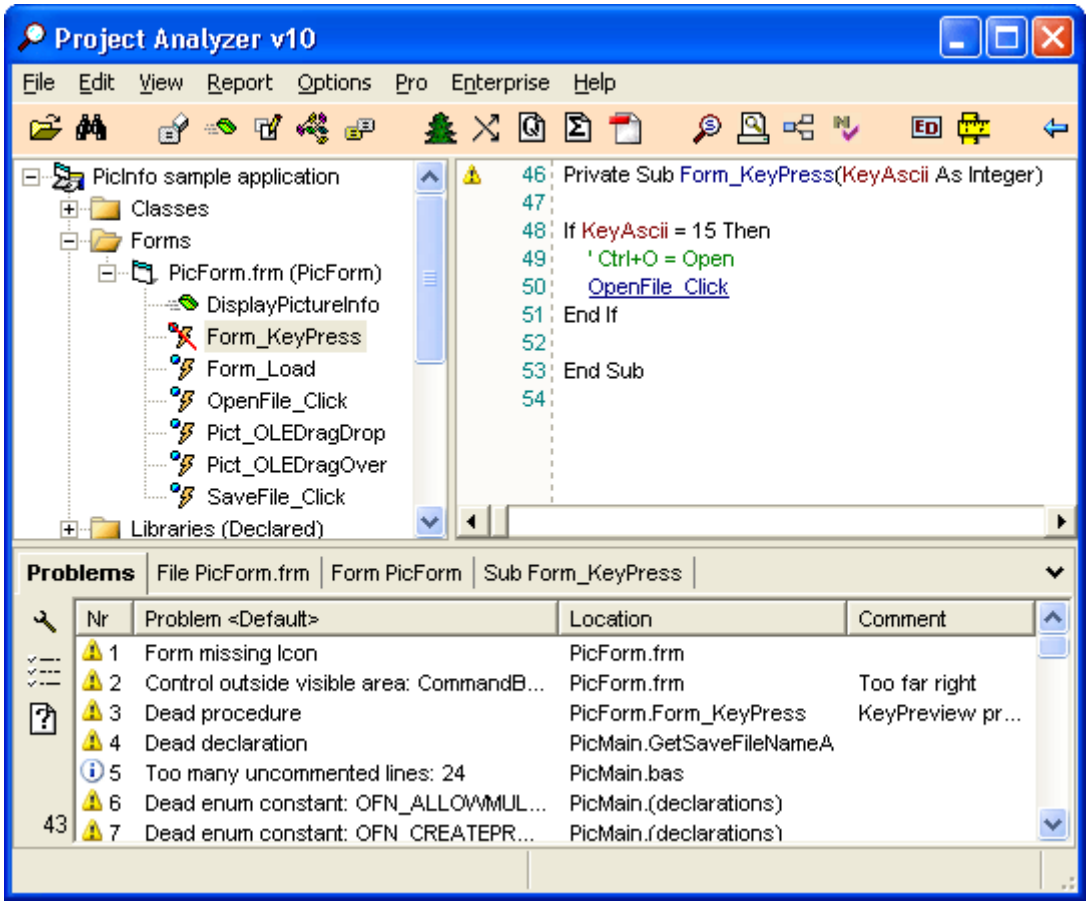
Tip to fix error *Programmatic access to Visual Basic Project is not trusted*. Your security options do not allow access to your VBA code. You can enable access by the following procedure in the appropriate Office application:

In Office XP or 2003, open the *Tools* menu and choose *Options* dialog. Choose the *Security* tab and press the *Macro Security* button. Select the *Trusted Sources* tab, and check *Trust access to Visual Basic Project*.

In Office 2007, press the round Office button and choose *Word Options* (*Excel Options* etc.). Choose *Trust Center*. Press the *Trust Center Settings* button and select the *Macro Settings* page. Here enable *Trust access to the VBA project object model*.

In Office 2010, 2013 or 2016, select the *File* tab and press *Options*. Choose *Trust Center*. Press the *Trust Center Settings* button and select the *Macro Settings* page. Here enable *Trust access to the VBA project object model*.

2.3 Main window of Project Analyzer



Hypertext view

The hypertext code panel appears in the top-right corner. Project Analyzer highlights and hyperlinks the code for you to surf in. 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 offering a way to view the declaration of the selected item, locate references to it, print, export and copy selected code, toggle the display of line numbers etc.

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.

Project tree

The tree view on the left shows the structure of your project. It includes each disk file with the module(s) and procedures in it. Left-click the items to move around in your project. Right-click the items for a variety of options and commands.

For image files and files containing several images (.frx files), there is a special command. Double-click the file to view its contents.

Details panel

The bottom of the window is reserved for code review results and detailed information.

On the Problems tab you will see a list of problems found by automated code review. You can configure the types of problems Project Analyzer finds by clicking the  *Problem options* button. We will go into the details of code review later.

On the other tabs you will see details on the selected disk file, module and procedure, respectively. These tabs also let you learn file dependencies and procedure call trees in an interactive way.

Legend

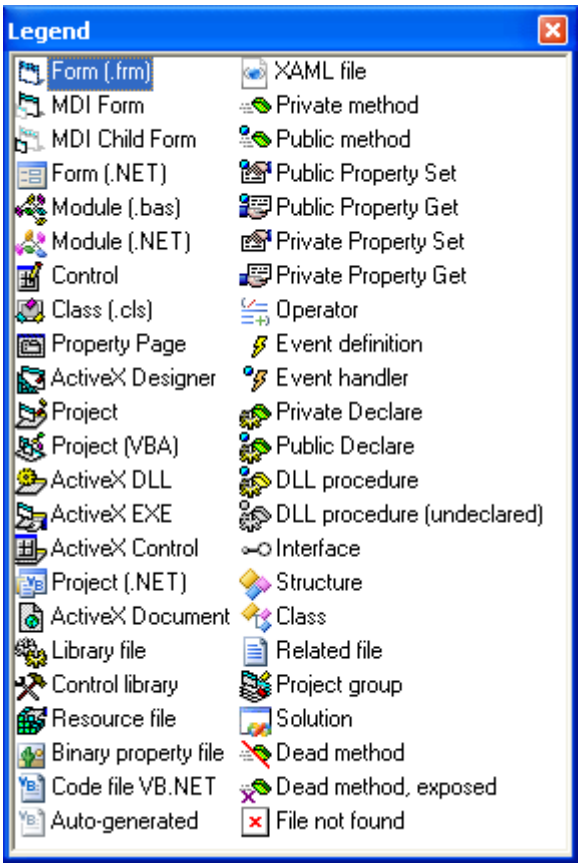
Project Analyzer uses various symbols and icons to illustrate your project (image on the right). In addition, there are some symbols related to problems found in the code.

~~Overstrike~~ indicates dead code.

A red line over an icon indicates unused, dead code.

A violet X indicates exposed code with deadness status *unknown*. No use was found for that code. The code is exposed, however. This means it is visible to other projects, which may use it. Because of this, Project Analyzer could not tell for sure if the code is really dead or not. It is up to you to decide. To be sure, run multi-project analysis to analyze all possible projects which could use the code. If the code still shows up as exposed, you may delete it since it is not used by any of your projects.

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



2.4 Producing reports

Project Analyzer produces a large number of reports. You can find many of them in the Report menu. There are also numerous Report buttons in various dialog boxes and other windows. These buttons usually report the window contents. This allows you to produce ad-hoc reports of a selected aspect of your system.

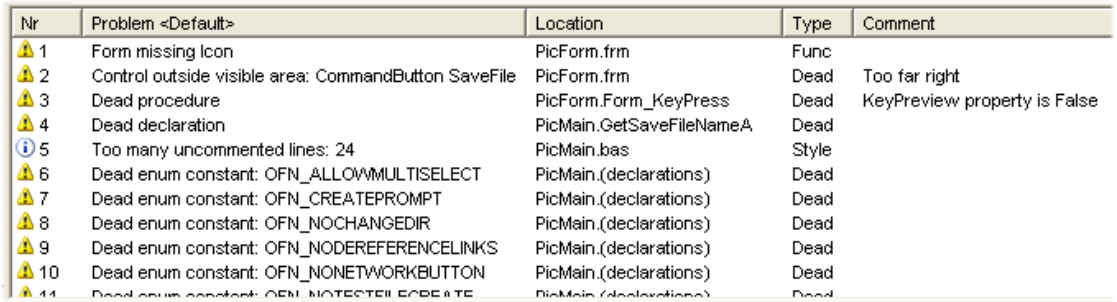
You can get the reports in a lot of formats for different uses. To view a report in another format, first select the report type in the toolbar of the main window, and then make a new report. The available report types are listed here.

- Display report (default option). The report shows up on a Project Analyzer window. Here you can print the report, save it to a file or copy to the clipboard.
- Print report. This option prints your report without showing any preview.
- HTML report. View the report in your web browser. Publish reports online.
- RTF report. Open the report in a word processor, such as Microsoft Word or WordPad.
- PDF report. View the report with Adobe Acrobat Reader. Publish reports online. Print reports.
- Text report. Produce plain-text reports for uses where the other formats are not viable.
- CSV report. Export column format reports to a spreadsheet application such as Microsoft Excel. Not suitable for reports that are not in a column format.
- MHT report. The report goes into a single-file web archive for viewing with Microsoft Internet Explorer. This format is suitable for saving a Project Printer generated project web site (see page 26) as a single file.

3 Automated code review

Project Analyzer reviews your code for errors, omissions and questionable coding style. Project Analyzer lets you detect flaws, enforce preferable programming practices, remove unused (dead) code, make your program behave in a more robust way and improve your code for better compatibility with newer Visual Basic versions.

Once Project Analyzer has completely analyzed your code, a list of coding problems appears at the bottom of the main window. If you don't see a list similar to the picture below, press *Ctrl+D* to bring it up.



Nr	Problem <Default>	Location	Type	Comment
1	Form missing Icon	PicForm.frm	Func	
2	Control outside visible area: CommandButton SaveFile	PicForm.frm	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.bas	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	
10	Dead enum constant: OFN_NONETWORKBUTTON	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 the following categories:

1. Optimization issues have a negative effect on the size and execution speed of your program. By fixing these problems you get a smaller program that runs faster – without losing any functionality at all.

2. Style suggestions deal with the way the source code is written. While the end users don't care about the style of your source code, consistent style is what makes a good program less prone to errors, easier to maintain and faster to develop further. Style is a matter of proper coding standards. Naming standards are a part of Style (see Project NameCheck on page 23).

3. Logic flaws are oddities in the control and data flows of your program: questionable coding, erratic behavior, hidden flaws and problems that are waiting to emerge later. These problems can cause your program to fail or to produce bad results, or they can represent optimization opportunities. Take time to carefully examine any found logic problems.

4. Functionality problems affect the actual behavior of your program at run-time. These are the problems that the end users will notice, sooner or later. Why not be ahead of them and fix the issues before anyone sends you a bug report?

5. VB.NET compatibility problems appear in classic VB projects. They represent incompatibility issues that appear when you try to migrate your code to Visual Basic .NET. *These problems are available only in the Enterprise Edition.*

6. Internal problems indicate problematic conditions in the analysis. A file may be missing, for example.

Problem filters

Project Analyzer supports hundreds of code review rules ranging from critical issues to less important suggestions. Fortunately, you are not required to follow Project Analyzer on everything. After all, we developers will never agree on what the best programming standards are. You can pick the subset of code review rules that suits your style best. You may be interested in just optimization issues, for example. Or you may think that an occasional GoTo is not such a problem. This is where you use problem filters.

A problem filter enables you to select and configure the subset of code audit rules that you are going to follow. You can create several filters for different purposes: quick check filters for the development phase and strict filters for a final polish. You can build a filter from scratch or base it on an existing filter.

Project Analyzer comes with a set of predefined filters. They are described in the following table.

<Default>	A default set of rules to start with. This default filter is applicable to most systems. You can use it as a starting point for configuring your own filters.
<Dead code>	Report dead and semi-dead code. *Ignore* dead but exposed code. This filter reports only certainly dead code.
<Dead code + exposed>	Report dead and semi-dead code. *Include* dead but exposed code. This filter reports all dead code, including *dead but exposed* code, which could be in use by external projects.
<Flaw finder>	Locate flaws and omissions and prevent potential errors.
<Functionality>	Report problems affecting how your application behaves and looks.
<Hide all problems>	Disable all problem checking.
<Logic>	Report logic flaws: oddities in the control and data flows of your program.
<Optimizations>	Report optimization issues and dead code.
<Project NameCheck>	Report naming standards related problems. *Requires Pro or Enterprise Edition.*
<Strict - Show all problems>	Report all problems. This filter may not be very practical since it is likely to show a very high number of problems.
<Style>	Report style suggestions, including clearing of variables and API handles, class design rules and Project NameCheck standards.
<VB.NET Compatibility>	Report all VB.NET compatibility 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  *Problem options* button to select another filter and configure your own. In fact, you *should* configure your own filter. We all have a different coding style, and not all code review rules are suitable for every developer.

Once you've configured your filter, you can share it with your colleagues. That's especially useful to enforce programming standards among teams. To save a filter to a file, use the Export button. Send the filter file to another Project Analyzer user. To load the filter in Project Analyzer, use the Import button.

We will now go into the basics of the various code review rules available in Project Analyzer. Since this is a tutorial, we will not list all the available rules here. See the help file for *Code review rules* for a comprehensive list with details.

3.1 Removing dead code

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 the program harder to read. This unused code never runs. It is called *dead code*. You can remove dead code without affecting the functionality of your program. Of course, sometimes you may decide to keep some parts for later use.

Dead code removal should be the first task you do to optimize your code. Only after that should you focus your efforts on the *live code* that still remains.

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 delete dead code in Visual Basic, remember to check every now and then that your program still compiles. Just in case.

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. Auto-fix makes a copy of your source and removes or comments out 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, superfluous code comes in several flavors:

Completely dead code is not required for the program to compile. Since it's not used for any purpose, it can be removed.

Semi-dead code is required at compile-time, but not when the program runs. As an example, semi-dead code may be required by dead parts only, making it effectively useless at run-time. Alternatively, code can be in partial use, such as a variable being read but not written to. Semi-dead code is often a sign of a flaw, or a missing or deleted implementation. Even when it's not an error, semi-dead code should be deleted to keep the code easier to maintain.

Exposed dead code appears unused, but it could be called from an external program. Before deleting exposed dead code, be sure to analyze all possible external programs together in a multi-project analysis.

Completely dead code

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.

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 to see 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. – An orphaned event handler is a procedure that used to be a regular event handler, but that is no more. Consider the Button1_Click event handler. If you remove Button1, the handler is orphaned. Project Analyzer reports an orphaned event handler as a dead procedure, like any other Sub.

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. Delete them to keep your code more manageable and to avoid introducing any errors due to dead variables or constants being brought back to life.

Dead Types and Enums are data definitions 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.

Dead Enum constants. An enumeration constant is not used. You may have forgotten to use it, or it is useless in this application. You should verify which case it is. If it is useless, you may remove it if you are sure you won't need it later. The removal doesn't affect the functionality of your program. You may consider keeping this constant available for the sake of completeness or later use, though. Note that removing a constant from an Enum may change the values of the constants that come after it, causing side effects. When removing an Enum constant it is safest to explicitly define a value for the next constant to prevent breaking existing code. The code may rely on the specific numeric values of the enum constants.

Dead compiler constants are useless #Const lines or constants defined in project settings. They are not used for conditional compilation. The constant may have been forgotten or it might have fallen out of use when the code was modified. A dead compiler constant does not cause any harm in itself, but you

may want to remove it. If it's a #Const, removal will have no effect on the program. If the constant is defined in project settings, there is a theoretical possibility that the constant might come back to life. This could happen if a file referencing the constant was later added to the program.

Dead line labels and numbers. Line labels and line numbers are an obsolete syntax feature of Visual Basic. They are used for error handling and as the target of a GoTo or GoSub jump, statements which are best avoided. A line label (or number) that is not being used by any of these statements is dead. Remove it to ensure it does not distract any developer in the future.

Dead classes are ones that are not used at all. You can remove them.

Dead interfaces. You can remove .NET Interface definition that is not used.

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

Dead modules can be removed in their entirety. Nothing the module contains is in use.

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. It's a good idea to move the file to a backup location in case you need to take it back to use later.

Deadness of procedure parameters

A procedure should read each of its parameters. In the case of an *out* parameter (ByRef), the procedure can alternatively write to it. If it doesn't do either, the parameter is superfluous and should be removed. Sometimes you can't rewrite the parameter list, though. This happens when writing event handlers or overriding procedures from a base class.

Dead parameter. A procedure parameter is not used by the procedure. Does this indicate missing implementation or someone just forgot to remove the parameter? Passing superfluous parameters adds unnecessary overhead. In addition, a developer may misinterpret what the procedure does by looking at its parameters, which may introduce new bugs in the code later. Remove extra parameters when you can. Removal should be done carefully, though, because it can affect existing callers. At times, removal is not possible when you need to keep the interface unchanged. Project Analyzer ignores obvious cases, such as event handlers and Overrides, which must conform to a pre-defined parameter list.

ByVal parameter not read. A procedure parameter is not read by the procedure. It is written to, however. Since it is a ByVal parameter, it is essentially a dead parameter being used as a local variable. This is bad programming style. If the procedure is expected to use the parameter value, there is a bug. On the other hand, someone may just have forgotten to remove the superfluous parameter. To fix your code, Dim a local variable with the same name. To optimize the program and make it more understandable, remove the extra parameter if you can. Removal should be done carefully, though, because it can affect existing callers. At times, removal is not possible when you need to keep the interface unchanged. Project Analyzer ignores obvious cases, such as event handlers and Overrides, which must conform to a pre-defined parameter list.

Optional parameter never passed a value. An Optional parameter is never passed as a call argument. The passed value is always the default value of the parameter. Determine if it is a superfluous parameter that could be removed, and the parameter be replaced by a local constant. Alternatively, it is possible that callers have omitted the value in error. Review each calling line to see if the parameter should actually be passed some value. You can do this by right-clicking the procedure name and selecting References.

Deadness of function return values

Functions, Operators and Property Get accessors should return a value. Moreover, the return value should be used.

Return value not set. A function does not set its return value, which can be an error. The function always returns zero, empty or Nothing, depending on the data type. Based on the documentation or purpose of the function, determine which value it should return. You can also visit the callers of the function to see which value(s) they expect to get. If there is no meaningful value to return, rewrite the function as a Sub.

Dead return value. The return value of a Function is not used by any of its callers. This indicates a possible bug in the callers or a useless return value. Review the function and the callers to determine whether the return value should be used or whether the function should be rewritten as a Sub. This

problem does not apply to VB3 where return values must be used at all times. See also *Return value discarded*.

Return value discarded. The return value is ignored at a call to a function. Although it is not necessary to actually use the return value of a Function, simply ignoring it could indicate a problem in the caller's logic. Review the call to determine if the return value can be ignored safely. This problem does not apply to VB3 where return values must be used at all times. See also the issue *Dead return value*.

Semi-dead variables

Semi-dead code is something that is in partial use. In most cases, it is not in proper use, really.

To be in full use, a variable should be both written and read. A read-only or written-only variable is in partial use only.

Variable written, not read. A variable is given a value but the value is never read by the program. The variable is either useless or there is a flaw in the program. Review the write locations. Check the code to determine if it's a flaw or if the variable is simply a leftover from an earlier version. Remove the variable and the write statements if you are sure the variable is useless. Before removal, determine if the write statements contain any procedure calls that must execute. Removing these calls could make the program behave erroneously.

Variable read, not written. A variable is never written to, even though its value is read by the program. The value is always zero, empty or Nothing, depending on the data type. Review the code to see if the missing write is a flaw in the program or if it is intentional. The write statement(s) may have been deleted by accident or the developer forgot to add them in the first place. In this case one or more writes should be added. On the other hand, the write(s) may have been removed on purpose, in which case the variable now works as a constant. Consider removing the variable altogether or declaring it as a constant. The use of a constant prevents unexpected changes later in the life-cycle of the program. You can also save a little memory as constants don't require run-time data storage. — This review rule is especially important for object variables, because the variable will always contain the value Nothing. This can cause an *Object variable not set* error at the location the object is referenced. A special case is denoted as *No real value given, cleared/allocated only*. For an object variable this means the variable is set to Nothing but it never contains a live object. Alternatively, an array is allocated but nothing is stored in it. The array is consuming memory for no purpose.

In addition for a variable being read and written, the read and write statements should actually execute too. If not, we may be facing faulty code.

Variable users all dead. A variable seems to be in use, but it is not. The user procedures never execute because they are all dead. As a result, the variable is completely unused and useless at run-time. If you remove all the user procedures, this variable will stay as a leftover unless you delete it. Use this rule to hunt for related pieces of dead code that you can remove at the same time. Right-click the variable name and select References to see the dead users. If the variable seems useful, some features may be missing from your program. Consider bringing the variable back to life by calling the dead users.

Variable readers all dead. A variable is being read and written. The reads never execute because the reader procedures are all dead. One or more writes do execute, but it is to no benefit. This is either a problem in your program or a good place for optimization. Check to see if not reading the variable indicates a logical fault. It could indicate missing or removed functionality. You may need to bring the dead readers back to life by calling them, or add new read statements in the live part of your code. If there does not seem to be a fault, consider dropping the writes. Your code possibly does some unnecessary work in getting the right value to the variable. Remove the unnecessary parts to optimize your code.

Variable writers all dead. A variable is being read and written. The writes never execute because the writer procedures are all dead. One or more reads do execute, though, indicating a flaw in the program. The value read is always zero, empty or Nothing, depending on the data type. Review the code to see if this is causes an error at run-time. Determine whether one of the dead writer procedures should be called to bring the variable back to full life, or whether you should add a new write statement to give the variable a real value. – This type of flawed dead code is really hard to catch without Project Analyzer. The problem is especially nasty for object variables, because the object will always be Nothing. This can cause an *Object variable not set* error when the variable is referenced.

Variable not read, writers all dead. A variable is practically unused and useless. It is never read. Write statements do exist, but they never execute because the writer procedures are all dead. The variable looks like a leftover from previous functionality. There is no point writing values to the variable

any more, since there are no reads from it. Consider deleting the dead writers along with the variable. Removing useless parts helps maintenance.

Variable not written, readers all dead. A variable is practically unused and useless. It is never assigned a value. Read statements do exist, but they never execute because the reader procedures are all dead. The variable looks like a leftover from previous functionality. The dead readers may contain a hidden bug (dead bug, so to say). The readers may perform in an unexpected way since the variable never gets any useful value other than zero, empty or Nothing. Taking the readers back to use can bring the bug to life. Consider deleting the readers along with the variable. Removing questionable, useless parts helps maintenance.

What has been said about variables above also applies to fields in user-defined types.

Other semi-dead code

Constant/Enum constant users all dead. The value of a Const or an Enum constant is not used at run-time. Even though the constant name is referenced in the code, the respective procedures are dead. The constant value is effectively dead too, because it is not used for anything. If the constant value *should* be used, the program contains a flaw. Alternatively, this is a leftover from removed or disabled functionality. In that case consider deleting the constant along with the user procedure(s). Note, however, that deleting a constant from an Enum might render the Enum incomplete or change other values in the same Enum. To avoid breaking existing code, you may want to keep useless constants in the Enum.

Class not inherited. A class is defined MustInherit but it is not inherited by any child class. Because it is an abstract class, you cannot instantiate it and it is thus of no use at run-time. However, the class is in compile-time use as a data type, so you cannot just remove it. Verify how it is being used to decide whether you should add a child class, remove the MustInherit keyword or remove the class together with the users.

Class not instantiated. A class is not instantiated even if it should be. Because no instances exist, you cannot access the instance procedures and variables. Thus, the class is not in use at run-time. However, the class is in compile-time use as a data type, or definitions such as Enums declared within the class are in use, so you cannot just remove it. Verify how it is being used to decide whether you should instantiate it via the New operator or remove it together with the users. This problem does not apply to abstract classes, classes with Shared members, classes that prevent instantiation or classic VB interface .cls files.

Implemented interface not used. An interface is implemented by one or more classes or structures. However, the interface is not used anywhere outside of the implementors. The interface is possibly unnecessary. This problem type is available for VB.NET Interfaces. It is also available for classic VB classes that are used as interface definitions.

Interface not implemented. An Interface is defined and used but it is not implemented by any class or structure. The Interface has no effect at run-time. You should either remove it or add one or more implementations. This problem type is available for VB.NET Interfaces. It is also available for classic VB classes that have no code in them except for a public interface definition. These classes are candidates for either implementation or removal. You can implement a class by either adding code in its procedures or by implementing it via the Implements statement.

Exposed dead code and multi-project analysis

Dead but exposed. Some project types expose an interface that other projects may call. As an example, an ActiveX project defines a public class with methods for other programs to call. Projects that can expose an interface include library projects and server projects. Starting with VB2005, even standard .exe projects can expose an interface. To determine calls to the exposed code, it's not enough to analyze just the one project. You need to analyze the calling projects too. In fact, you should analyze all of them together.

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. The default settings of Project Analyzer 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. The simplest way to run a multi-project

analysis is to open a .vbg or a .sln file. You can also select multiple project files to analyze. 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 invisible or disabled 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 were unnecessary controls in the program in the first place? To remove a control from a form, developers sometimes set the control to invisible. Alternatively, they drag it outside the visible form area. This is a quick way 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. Unfortunately, this kind of an incompletely removed control is easily left on the form, along with its event handlers. It continues to consume system resources and keep the program unnecessarily complex and hard to understand.

There are some rare cases where an invisible or disabled control can be required for the operation of the program. This means 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.

3.2 Further optimizations

Variable missing type declaration. Always give your variables a proper type. If you don't declare your variables as a specific type, you will get either Variant (classic VB) or Object (VB.NET). A Variant/Object can contain any kind of data: numbers, text, object references, tables... In some cases, this is great. 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 a bad idea? The first reason is that using specific data types greatly decreases the probability of run-time errors. You can never be sure about type of data a Variant/Object holds, but an Integer always holds a number. Forcing the use of an Integer, for example, makes sure you will never perform calculations on a string or any non-numeric object. In object-oriented programming, declaring proper data types (class name instead of Variant/Object) also enforces early binding. That is, the VB compiler will actually check the validity of method calls. A Variant/Object data type dictates late binding, which means that no compile-time checking is possible. Errors due to the use of an incompatible data type only trigger at run-time, meaning you move error detection from the compiler to the end users.

The other reason is that using strongly typed data makes your program run faster as VB can avoid unnecessary data type conversion operations. The third reason is because using Variant/Object consumes more memory than the other data types. You can save memory by declaring your variables, and most importantly, your *arrays* as Integer, Single, String, etc.

Proper typing is especially important for procedure parameters. Adding `As datatype` for each parameter is 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.

To get all these benefits, give your variables a specific data type 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
```

In VB.NET variable declarations work differently, though. Both x's will be Integer.

Starting with VB 2008, you can use type inference. This means the VB compiler deducts the data type from the initial value of the variable. You stop thinking and let the compiler do the thinking for you. Here is an example of how it works:

```
Option Infer On
Dim x = 3
Dim y = GetValue()
```

As you can see, the data type declaration is not explicit. x will be Integer since you initialize it with an Integer value. But can you tell the data type of y? No, you can't. It's the same as the return data type of Function GetValue. Type inference sounds like a great productivity feature, but it actually contains a hidden caveat. Developers stop thinking. As another caveat, a future change to the return data type GetValue() will propagate to y, possibly even further in the code. To ensure maximum quality, you should always declare the data type and not rely on Option Infer On.

Functions missing type declaration. Like variables, functions and properties require a type definition for their return value. 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 classic VB there is no way to require that, so you need to run Project Analyzer.

Object variable declared As New. In classic VB, 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 already holds 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*. VB.NET treats "As New" variables differently. VB.NET only instantiates the object once, which means there is no performance hit.

Consider short-circuited logic. In the expression `(x And y)` and a similar one, `(x Or y)`, both operands (x, y) are evaluated regardless of the value of x. 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 or involves a function call.

In classic VB, consider splitting an If ..And.. condition as two nested If statements like this:

```
  If x And y Then DoIt

→If x Then
     If y Then
        DoIt
     End If
  End If
```

Short-circuiting If ..Or.. yields more complex code. It may be useful where hard optimization is required.

```
  If x Or y Then DoIt

→If x Then
     flag = True
  ElseIf y Then
     flag = True
  Else
     flag = False
  End If
  If flag Then
    DoIt
  End If
```

In VB.NET, consider replacing And with AndAlso, and Or with OrElse. Read the VB.NET help for differences between And/AndAlso and Or/OrElse.

Short-circuiting involves a risk, because it changes the logic. If the second operand (y) calls a function, the call is no longer guaranteed to execute. Depending on what the function does, this may lead to your code not executing something important that should run at this point.

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 extensive articles on string optimization.

3.3 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 is able to report so many style issues that the list might look overwhelming at first. Fortunately, you're not supposed to follow all the rules. Pick those ones that are useful for you. Forget about those ones that are too exotic or complicated to enforce. Define your very own problem filters with the Problem Options command in the Options menu. Once you learn to polish your code with Project Analyzer, you can enable more review rules in your problem filter in order to make your program yet shinier.

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

Declaration style

Option Explicit Off. If you don't use Option Explicit, or you have set it Off, you've got to learn to set it On now. In classic VB, 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 to require less RAM. They may also 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 proper data type, the best type suitable for the job.

Here is yet another reason why you should use Option Explicit. It forces you think about the correct data type when you're writing the Dim statements, functions and properties. Increased thinking leads to increased quality, right?

Option Strict missing. A file does not define `Option Strict On`. In VB.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. Always give your procedures, variables etc. a scope. VB's default scope rules are somewhat complicated. They may lead to a scope that is either too wide or too limiting. 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()
Friend    Sub Calc()
Public    Sub Calc()
Protected Sub Calc()        ' VB.NET only
Protected Friend Sub Calc() ' VB.NET only
```

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 library projects, 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 a public interface to other libraries or programs. In standard executable programs, Public actually works as if it was Friend, because standard executables don't publish anything for other programs to call.

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 to see 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. Note that it's especially important to encapsulate *excess scope* class variables as properties. Variables defined in classes should be made inaccessible to other parts of the system. Read/write access should only be through properties (or methods). This protects your data from being modified the wrong way. Encapsulation also lets you keep certain data read-only. Just leave out the Property Set/Let part or make it Private. As an additional benefit of encapsulation, 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. The actual variable is not being passed. This makes your code safe. Whatever value you write to the parameter in the callee, the changes won't propagate back to the caller.

ByRef tells VB to pass a parameter by reference. This means that not only the value, but also a reference to the actual variable is being passed. 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) ' 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) ' Hello, fellow!
End Sub
```

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

Fortunately, VB.NET fixed this problem. In VB.NET the default is ByVal.

It should be noted here that there are good uses for ByRef. One case is an *out* parameter that returns a value back to the caller. This is usually best avoided, but sometimes it is 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. In addition, sometimes VB just demands ByRef. At those times VB will tell you about that. Project Analyzer knows about the obligatory ByRef cases and won't issue any unnecessary warnings about them.

The not-so-stylish statements

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

GoTo and **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. One should use real Subs and Functions instead of GoSub. Besides, GoSub is outdated and it isn't supported by VB.NET.

Exit statement found (Exit For|Do|While etc.). Use of the Exit statement indicates unstructured program flow, much the same way as the GoTo statement. An Exit statement causes an immediate jump out of a programming structure such as a loop. In purely structured programming style Exit statements should not be used. Well-placed Exit statements are not considered harmful by many programmers, though.

Exit Sub|Function|Property found. An Exit Sub|Function|Property statement causes an immediate jump out of a procedure. It potentially makes the procedure have several exit points. Procedures should have only one exit point at the end. — There is a case where VB requires a premature Exit. This is when you need to quit a procedure immediately before an On Error GoTo xxx type error hander. Project Analyzer allows this use.

Return statement found. A Return statement causes an immediate jump out of a procedure in VB.NET. It potentially makes the procedure have several exit points. Procedures should have only one exit point at the end. — There is a case where VB requires a premature Return. This is when you need to quit a procedure immediately before an On Error GoTo xxx type error hander. Another case where Return must be used is in an Operator (VB 2005 and later). Project Analyzer allows these uses. This rule is available for VB.NET.

Multiple Return statements found. A procedure contains multiple Return statements. There should be exactly one exit point in a procedure. Multiple exits make the procedure harder to understand. The requirement for multiple exits may indicate too complex a procedure, in which case it should be split. Rewrite the procedure to contain a maximum of one Return statement. This rule applies to VB.NET. Note that this rule does not count Exit Function statements. The rule *Exit Sub|Function|Property statement found* detects them.

While. The While..Wend or While..End 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. Starting with VB 2005, the Global keyword has a new meaning. It's best not to use Global to declare Public variables any more.

Enum style

Project Analyzer lets you enforce consistent standards for Enums. You can set one or more of the following requirements: all Enums should define a zero constant; all constants should be explicitly set as a `name=value` pair, not relying on VB to calculate the value; each constant should have a different value; Enum constants should be used in place of the numeric value; and each constant should actually be utilized. In addition, each Select Case block should have a Case for every constant (see the next section for this rule).

Enum has implicit member value(s). One or more members of an Enum do not have an explicitly declared value. The values might change if new members are added. This potentially affects code where the old values are used instead of the Enum member names.

Enum missing zero value. An Enum should define a constant with the value of zero. The default value of an uninitialized Enum variable is zero. Explicitly defining a zero Enum constant makes the uninitialized value valid. You can name the zero constant with a descriptive name such as None or Empty or Error.

Enum with duplicate value. Two or more constants in an Enum define the same value. Check to see if this is by accident. Declaring the same value several times may be intended. On the other hand, an error may have caused the duplicate definition. Check to see if you can join the multiple constants into one to prevent problems understanding the Enum later.

Enum constant expected. An enumeration constant is expected on this line. An Enum datatype is assigned a value not belonging to that Enum. One should only store Enum values, not integers or other datatypes. Otherwise you easily end up with a value outside to the Enum. An Enum should fully define and document its acceptable values. Using an undefined value is undocumented programming, really, or even an indication of erroneous logic. A programming style that uses "magic" integers interchangeably with Enum constants is prone to maintenance errors. If the Enum values are later changed, the respective "magic" integers are easily left unchanged, which leads to new errors.

Other style rules

Class member rules review the variables, properties and methods of classes. You get to know unorthodox classes, such as where a class consists only of data, or alternatively, stores no data. Specific rules review classes that are being used as an interface. This group of rules also checks out whether there are any orphaned event handlers and whether event handlers are being called directly. Check out the help file for the actual rules.

Class design rules (VB.NET only) form a group of style issues related to the class hierarchy and the definitions of classes and their members. This group of code review rules, part of the problem detection feature, detects potential problems with your VB.NET Classes, Structures and Interfaces. Rules include ones such as what kind of constructors and finalizers should exist, which keywords should be used on member declarations, correct scope declarations, improper use of inheritance or interfaces and the like.

Parameter style rules enforce a reasonable number of procedure parameters and check proper use of ByVal and ByRef parameters. These rules review "in" and "out" parameters and can suggest where to rewrite a Sub as a Function. There are also specific rules for Optional parameters.

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.

Style rules don't end here. There are even more of them. Refer to the help file for the complete list of all available style rules.

3.4 Logic flaws

Logic flaws are an important problem category. These are problems that potentially cause your program to fail or to produce incorrect results. If not plain erroneous, such problems often point out where the code should be optimized. Logic flaws can be hard to catch without a code analyzer.

This is a problem category that really requires thought. Spend time to examine each problem. Carefully determine if and how it should be corrected.

Path analysis

To detect logic problems Project Analyzer runs path analysis. It follows the execution flow through branches, loops and jumps to determine which lines will actually execute at run-time. It checks out when variables are read and when they are written to. As to loops, Project Analyzer evaluates whether a loop executes zero, one or more times and if it's an eternal loop.

Unreachable code found. A block of code is dead because it is not reachable. It will never execute. Unreachable code means impossible branches, loops that don't run, unused error handlers (On Error style) and labeled blocks that are not being jumped into. Code that immediately follows an unconditional jump (such as a Return or Exit statement) can also be unreachable. Unreachable code cannot run at all. It is a programming error: bad logic or the remains of old code that should have been deleted. Determine whether you should remove the unreachable block or if it should rather be brought back to life. Unreachable code should not exist. If it is necessary to leave it for later use, consider commenting it out or putting it in a #If..#End If block to exclude it from compilation and to make it explicit it may be

required later. — This rule is good for detecting On Error style error handlers that are not in use. It can also detect broken loops that don't loop (last statement never reached). It can even detect code that was removed by writing Exit Sub above it.

Variable read before written. A local variable is being read before a write instruction ever executes. At this point the variable will always contain its default value (zero or empty string). This is most probably a mistake. The preceding write instruction may have been deleted or the execution flow wrongly modified. It could also indicate a superfluous variable or the use of an incorrect variable. This rule ignores error handlers, as the run-time triggering of an error handler is difficult to determine by code analysis.

Variable read before written (along some path). A local variable is being read before a write instruction is guaranteed to have executed. Thus, when the variable is read, it may have a proper value, but it could also contain its default value – potentially due to a mistake. Depending on the case this may be problematic or not.

Note: The difference between *Variable read before written* and *Variable read before written (along some path)* is that the former is a stricter variant. The latter triggers when a read before write is even a remote possibility. The former triggers only when a read always happens before a write.

Object read before written. A local object variable is being accessed, but its value is always Nothing at this point. A run-time error is likely (null pointer reference, object variable not set). This warning appears where an object variable cannot be set by a preceding instruction.

Object read before written (along some path). A local object variable is being accessed, but its value may be Nothing. A run-time error may occur at this point (null pointer reference, object variable not set). This warning appears when some path may have left an object variable unset. To avoid a run-time error, verify that the variable will actually contain an object at this point.

Tip: To improve the accuracy of these "read before written" rules, enable analysis of library files (Enterprise Edition required).

Assigned value (or object) not used. A local variable is assigned a value that is not actually used after the assignment. It is either overwritten or not used at all. Review the code to see if it is a programming error or if you can optimize by deleting the write statement. Make sure you are not removing any required function calls at this point, calls that have useful side effects.

Parameter value not used. The value passed to a procedure parameter is not actually used. The value is overwritten or execution never flows into a read instruction. Review the code to see if it is an error. — This rule is related to the *Dead parameter* rule. A Dead parameter is one that is not used at all. *Parameter value not used* means the parameter name is indeed in use, but not the value passed to it.

Passed value not used. A parameter is being passed in a procedure call, but the receiving procedure does not actually use the value. Is the logic still all right? It may be possible to optimize by leaving out the parameter or by passing a dummy constant instead of a calculated value. This rule excludes "out" parameters, which are used to return a value.

Loop runs forever. An eternal loop was found. Once the loop is entered, there is no exit or all exits are unreachable. The program could jam here.

Loop runs only once. A loop starts but it doesn't execute another round. Is it a flaw? Should it run several times or can you just remove this useless loop? — Related rule: A loop that runs zero times is reported as *Unreachable code*.

Cyclic recursion found. Cyclic recursion happens when A calls B, which calls C, which eventually calls A again. Recursion through two or more procedures is difficult to understand and get to work correctly. There is also a risk of endless recursion. Cyclic recursion is also known as indirect recursion. The opposite is direct recursion, a procedure calling itself repeatedly. Direct recursion is not as problematic as cyclic recursion. Consider replacing cyclic recursion with either a non-recursive technique or direct recursion. Restricting recursion to a single procedure is easier to manage. To prevent endless recursion, you can add a safety counter that breaks the execution if it goes too deep. To fully understand a large recursive cycle, get the Recursion diagram in Enterprise Diagrams. Note: This rule triggers based on static analysis. Due to run-time conditions and checks, the actual execution path might not be recursive.

Select Case and conditionals

A Select Case statement should define a Case for each possible input value. For the code to stay understandable, each input value should match no more than one case condition. In addition, the last branch should be a Case Else that takes care of unhandled and unexpected input values. Naturally, useless dead case branches should not exist.

Case branch(es) missing for Enum. A Select Case statement branches off an Enum value, but it does not contain a Case for every Enum constant. This may indicate missing logic, possibly due to an Enum constant added later. Check to see whether this causes problems when an unhandled Enum value is passed. This rule requires every Enum constant name to be listed in a Case, even if the constants have the same value. Use of the underlying numeric values is not acceptable. Ranges such as Case X To Z are not acceptable either. Listing each individual constant helps prevent breaking existing Select Case blocks if the Enum is later changed or rearranged. To accept Case Else as a default branch, check *Ignore when Case Else exists*.

Case Else missing. A Select Case block is missing its Case Else branch. An unexpected input value can go unhandled. This rule lets you prevent bugs from causing havoc beforehand. You can require each Select Case to have a Case Else just in case an unexpected value comes up. As an exception, Case Else is not required by this rule if existing Case branches cover all theoretically possible input values. Tip: To detect unexpected values while debugging and run the final executable without interruption, add the Debug.Assert False statement in each Case Else that may execute due to an unexpected input value.

Case useless. A Case condition will not match any input values. The condition is already covered by an earlier condition or conditions (duplicated Case or totally overlapping range). This warning is also triggered for a never-matching Case x To y, where y is less than x.

Case overlap. A Case condition overlaps an earlier condition. Since previous conditions have priority, this condition will not match all the given values. As an example, Case 1 To 5 and Case 3 To 7 overlap in values 3 to 5. To make the Cases more understandable, rewrite this condition so that overlapping does not occur.

Condition always True or False. A conditional statement always evaluates the same way. This is an unconditional statement, really. It may be a flaw in the code. Conditional statements should be run-time decisions. Tips: To exclude useless code, use #If False rather than If False. This prevents compilation of dead lines into the executable. To loop eternally, use Do..Loop without either Until or While.

Conditional block excluded. A conditional compilation block (#If..#ElseIf..#Else..#End) is excluded due to a False condition. This is OK where conditional compilation is used to produce several versions of the program. In some cases, it may indicate remnants of code that should have been removed. This rule is there to help you detect all False branches, should you wish to get rid of them. — A special consideration affects VB6 code that is planned to be migrated to VB.NET. The VB.NET upgrade wizard doesn't migrate code inside a False branch. It copies the code "as is". If the False branch is large, it is recommended that you temporarily set the condition it to True before migration. Otherwise you will have to migrate it manually.

3.5 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 the problem detection feature of Project Analyzer. 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.

General functionality

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 VB Watch, a tool that adds advanced error handlers to your code. VB Watch works with Visual Basic 6.0. It adds error handling automatically. Once an error hits, the end user has an option to ignore the error and/or send you a detailed error report. VB Watch is available through *www.aivosto.com*.

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?

Hard-coded path found. A string that looks like a hard-coded path name exists in your program. File access through hard-coded paths will fail if the directory structure changes. The code seems to assume a certain directory structure at run-time. It is more flexible to operate with path names relative to the installation directory, for example. Where hard-coded paths are really desired, it is best to define the necessary paths with global constants or in a resource file to allow for easier maintenance.

Verify accuracy. A floating point constant may be imprecise. This rule searches the code for decimal numbers that look like common constants, such as pi or e. The rule verifies the accuracy of such constants. A flag is triggered where fewer than all available digits are used. The rule also flags values that are somewhat incorrect. This rule is good for verifying scientific programs. It supports hundreds of mathematical, physical and chemical constants. You can add your own constants as well. To customize the list of constants, see the file *constant.txt* in your Project Analyzer directory.

User interface issues (VB 3-6)

The following few rules work with the forms of Visual Basic 3.0 to 6.0.

Error event missing. Every Data control should have the Error event implemented. If not, 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.

Click event missing. A CommandButton or menu item does not do anything when clicked.

Timer event missing. A Timer does not fire an event at defined intervals. You could as well remove the timer.

Form missing Icon. A Form doesn't have an icon that is required. A default icon, generated by VB, will be shown.

Form missing HelpContextID. Your program uses context-sensitive help (F1), but a Form was found that has no HelpContextID set.

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.

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.

Hotkey conflict. Two or more controls or menu items share a hotkey. As an example, there are options &Save and &Search in the same menu.

Hotkey missing. A control or menu item is missing a hotkey. Checked controls: CommandButton, CheckBox, OptionButton, Frame, Menu.

Possibly twisted tab order. The tab order of controls looks questionable. Take a look at the TabIndex properties of the mentioned controls. The best way to do this is to open the form in VB, select the first control and press the Tab key repeatedly. Generally, the focus should move either right or down. Labels with an access key (& in the caption) should precede the control next to them. A container control should precede any child controls placed within it. A back jump is allowed to move the focus from the bottom of the form to the top-left corner. If the RightToLeft property of the Form is True, Project Analyzer uses a mirrored rule to require moves either left or down. The detection algorithm of this problem is not foolproof and the tab order might be all right in some cases.

3.6 More review rules and customization

Project Analyzer supports a wide range of problem detection rules. This tutorial only lists a part of them. See the help file for *Code review rules* for a comprehensive list. If you did not understand a rule, or if you disagree with the reasoning, you can right-click the problem icon in Project Analyzer for a detailed explanation.

You can customize the code review feature to ignore specific problems. There are some uses for questionable programming techniques after all. Quite often it's a specific location or two. What usually is a problem can really be desirable at times. You can mark specific locations so that Project Analyzer

will ignore problems in them. You do it by writing special comments in your code. These are called *comment directives*. A sample directive looks like this:

```
'$PROBHIDE DEAD
Sub MySub()
```

The directives 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.

3.7 Automatic code optimization with Auto-fix

Feature requires Enterprise Edition.

Project Analyzer is capable of fixing a subset of the coding problems it finds. 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. 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 overwriting any of your code.

There are two categories of coding problems: auto-fixable and those requiring a manual fix.

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 comprehensive list of auto-fixable problems can be found 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 add a comment next to the modified code. The comment will bear a prefix such as `'!` or `'HACK:` for you to notice it. This option handles dead code by commenting it out. You can define a special dead code comment, such as `'+` or anything you prefer to remind you the code has been commented out because it was dead.

Fix quietly. This option repairs all auto-fixable problems, but it adds no comment. Dead code is simply deleted without a trace.

Treat as manual fix. Selecting this option disables all automatic fixes. Auto-fixable problems will be treated the same way as manual fix problems. Use this option to comment your code with problem information so that you can open up Visual Basic, review the problems and fix your code manually.

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 the code as comments. You can specify the prefix to use for these comments. You can use a prefix such as `'!` or `'TODO:` or `'UNDONE:`. Look 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 the 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, there is a chance that your newly generated clean project might not compile or run. This might be due to live code being removed or an incorrect fix being applied.

Because of this, you should start by auto-fixing a limited number of problems at a time. Select a limited number of files to process in one run. 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. You can find the details in the help file.

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

Integrating with the Visual Studio .NET Task List

If you have Visual Studio .NET, `'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 (see the Options dialog in the Tools menu of Visual Studio).

3.8 Enforcing naming standards with Project NameCheck

Feature requires Pro or Enterprise Edition.

Project NameCheck is a naming standards review feature in Project Analyzer. It triggers a warning when the name of a programming object does not conform to your naming standard. You can configure your own standard based on what you think is the best way to name variables, constants, procedures, Enums, controls, modules and the like. You can define acceptable name prefixes and suffixes for each data type and demand constants to be in ALL_CAPS, for example.

> Note: Project NameCheck is most useful with the so called Hungarian notation, that is, names are prefixed (or suffixed) with a few characters. Microsoft has turned away from Hungarian notation in VB.NET. Project NameCheck is less useful with the VB.NET style naming conventions.

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. You can freely change the settings to whatever you think is the correct way to define names.

When you're done with the configuration, open the *Options* menu and choose *Problem Options*. Select the <Project NameCheck> filter. This filter is the best option when you're learning to use Project NameCheck and when you're configuring it. Later you can use any other filter that includes NameCheck problems, such as <Style> or <Strict - show all problems>.

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. You can configure the standard and see it applied right away to the currently analyzed project. There is no need to reanalyze the project.

Basics of a naming standard

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 more about the various configuration options read the help file topic *Project NameCheck standard configuration*.

3.9 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 files. 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 .vbg or .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 better to detect dead code than Super Project Analyzer.

When do you need 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 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.

4 Document your code

In many software 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 documentation features, such as reporting and diagramming. Although writing proper comments in 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. The following reports are available:

- File list: plain listing, with details, with file sizes and dates
- Files and projects: which file belongs to which project
- Module list: sorted listing of modules, modules classified into groups
- Namespaces list: VB.NET Namespace hierarchy
- Procedure list: plain listing, with details, sorted alphabetically
- Variable list, Constant list
- Types, Enums and Aliases report
- Dictionary report: all names sorted alphabetically plus statistics on names

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:
- AddEmployee
- UpdateEmployee

Calls:
- CheckDatabaseField

To fully utilize 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 a normal code line 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 (see page 26).

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 several ways:

- In the *Report|Lists* sub-menu, select *Variable list*. The Variable list shows the global and module-level variables in your program.

- The *Constant list* in the same menu puts out all constants, whether global or local, grouped by data type. You also get a listing of compiler constants in this report.
- 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, search for duplicated constants, for example.

4.4 Source code print-out and export

Project Analyzer produces syntax-colored print-outs of source code. In addition to print-outs you can save the code as a PDF file. Just browse to the code you wish to print and select either *Print code* or *Export code as PDF* in the File menu.

You can produce both color and monochrome output, with or without line numbers. You can put the code in several columns to save paper. By printing in 4 columns with a small font, you can fit 200–250 lines per page.

The available print/export ranges are: selected files, current file, current view (module or procedure) or just the lines you have currently selected.

In addition to PDF you can also export your code as syntax-formatted HTML or RTF. You can even copy code to the Clipboard to paste to a word processor. This way you can quickly produce great-looking documentation.

Tip: Use Project Printer (below) to produce complete syntax-formatted and hyperlinked source documentation.

4.5 Documenting all your code with Project Printer

Feature requires Pro or Enterprise Edition.

The most comprehensive and up-to-date software documentation is the source code. Project Printer is a feature in Project Analyzer that makes documenting the whole source code easy. Project Printer isn't a plain print-out utility. Even though the name suggests printing, you can generate versatile document files with it. Optionally, Project Printer formats your code for easier reading, adds hyperlinks, generates a table of contents and includes cross-reference information, problem descriptions and various reports.

Besides reporting all source code, Project Printer can also create a *Comment Manual*, which means a document built from the 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.

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*. 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* as the output format. This option makes a site identical to the HTML directory option, but all the data goes in a single file. You can open .mht files in Internet Explorer. Other web browsers may not support them, though. The great thing about the .mht format is that it's a single file instead of thousands of files.

Turn code into linked PDF

This option creates a PDF file that lets you surf your code. Hit F6 to run Project Printer. Select the files you want to go into the PDF. Click *Next* and select *Full source code*. Click *Next* again. Select *PDF*. Click *Next* and then generate the web site with the default options.

Print your code

Create a PDF file as described above. Print the PDF.

For conventional source-code print-outs without hyperlinks see *Source code print-out and export* (above).

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 special comment syntax*. Click *Next* again and choose the appropriate output type. PDF and HTML file are good choices. RTF is nice if you intend to edit the manual.

Comment manual is a manual-like report based on the 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*. View the help file to find out exactly which comments will be included in the document.

4.6 Documenting cross-references

A program is not just a collection of procedures; it's also 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 trees. A *file dependency tree*, or a *file use tree*, tells which other files a given file needs to compile and/or run. A *procedure call tree* tells which procedures are called by a given procedure.

Call tree window and reports

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. The + sign denotes a branch that could be expanded further.

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 in the Call tree window. Another version of call tree reports is available via the Report menu. This version produces more comprehensive (and longer) reports than the Call tree window.

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.

Call trees are also available in diagram format. Check out Section *5 Diagramming* to learn more.

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 program parts that a procedure requires to work. It lists all the 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.

Cross-reference reports

There are several reference based reports in the *Report|Cross-references* sub-menu.

The *Cross-reference report* is a master report that lists all *calls/called by* relationships between procedures. In addition, it lists where variables are being read from and written to, and where constants, Enums and Types are being references.

The *Procedure references* report lists procedures with their call sites. For each procedure it lists the lines that call it.

The *Variable references* and *Constant references* reports are similar to the *Procedure references* report. They list the lines accessing each variable and constant.

You can also view reference lists by selecting *References* in the View menu. In addition, you can right-click any programming object in the hypertext view of the main window. In the popup menu click *References* to view references to any particular programming object, such as a variable, constant, function and so on.

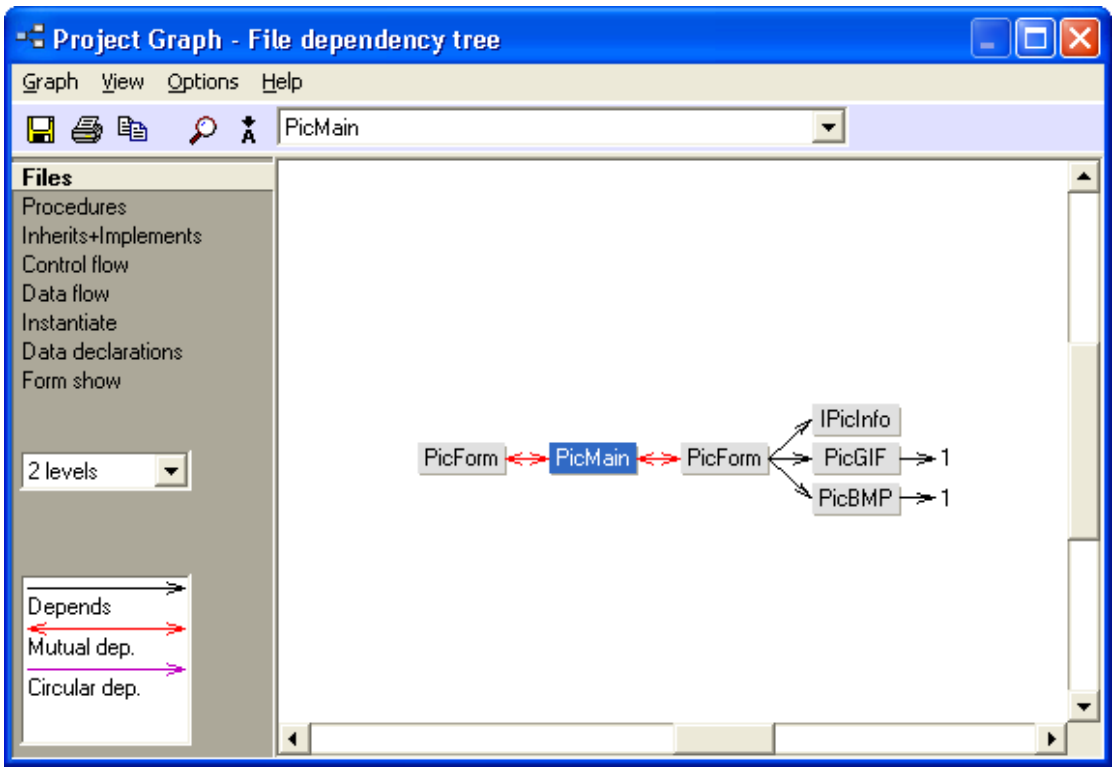
5 Diagramming

Project Analyzer provides two features to document the structure of a program in a graphical way. Project Graph is for interactive graphing and navigation. Enterprise Diagrams makes comprehensive documentation.

5.1 Project Graph

Feature requires Pro or Enterprise Edition.

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 the connections in a program. Instead, it offers a way to get different views in it. Start Project Graph pressing *F7* after analyzing a 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. The procedure-level detail has been left out and the modules remain. This is a higher-level view into the procedure call tree.

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 instantiation relationships as blue arrows. Thus, the Data declaration tree is a generalization of the Instantiation tree. You get to see kinds of all data type usage, including class instantiation.

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. The showing of 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. If the system is big and the dependencies are complex, the Enterprise Diagrams feature provides a better view into the dependencies.

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 are 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 graph as an image file.

5.2 Enterprise Diagrams

Feature requires Enterprise Edition.

Enterprise Diagrams makes complex diagrams for documentation purposes. Instead of interactivity, it offers the option to visualize all the links (calls, dependencies, variable uses etc.) in a single diagram. Start Enterprise Diagrams by pressing *Ctrl+F7* after analyzing a system.

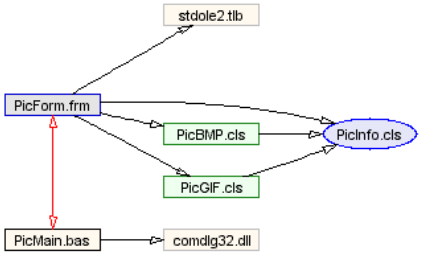
Diagram types

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.

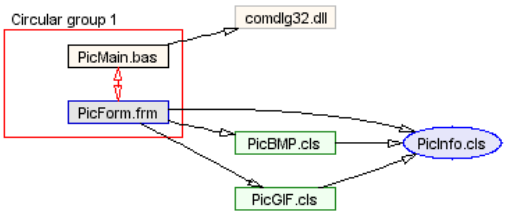
File dependencies. This diagram shows *file requires/is required by* information. A→B means file A requires file B to compile or run. A red double-headed arrow indicates mutually dependent files, which should be avoided if possible. File dependency diagrams also come in the following variants:

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.



File dependency diagram



File dependency diagram, circular groups boxed

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. The procedure-level detail has been left out and the modules remain. This is a higher-level view into a procedure call diagram. By leaving out cluttered details you get a high-level understanding of the call dependencies.

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.

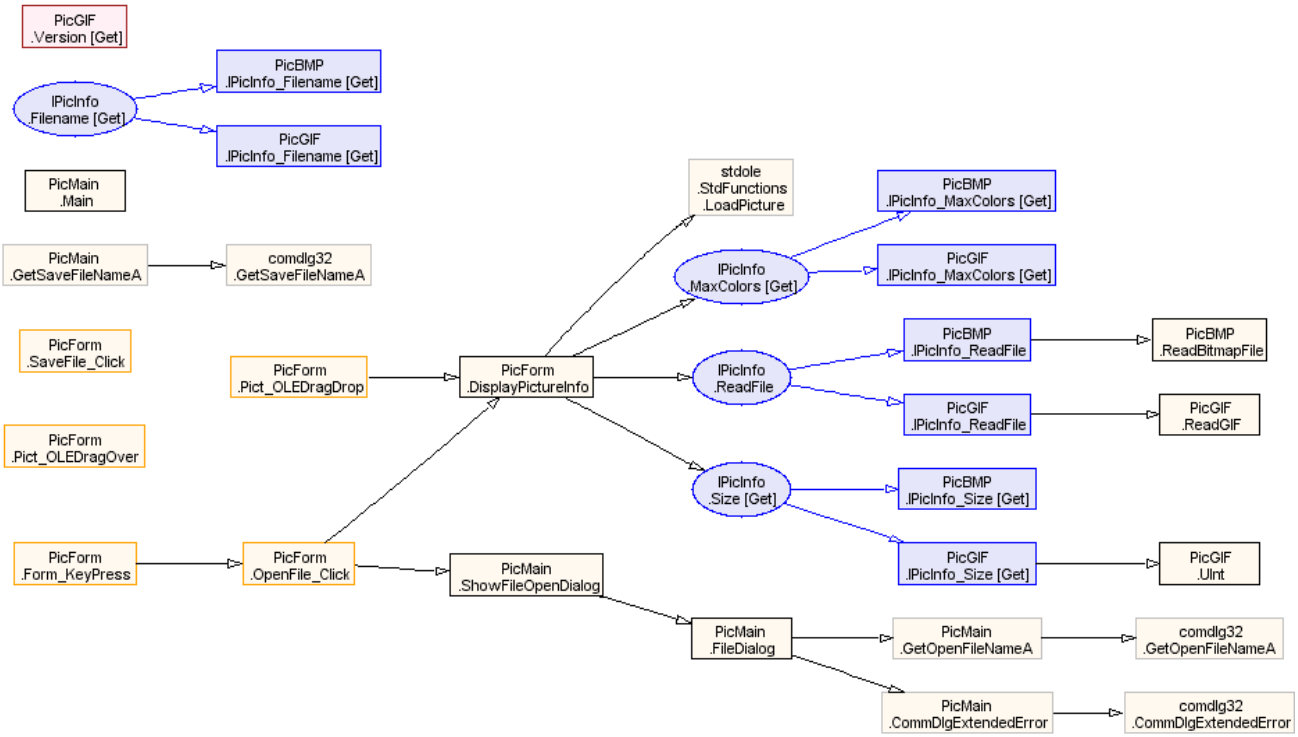
Variable data flow. This diagram shows data flows between modules *via variable read/write*. The difference to the Data flow diagram is that procedure calls and return values are not taken into account. Thus, this diagram considers only direct variable based data transfer.

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

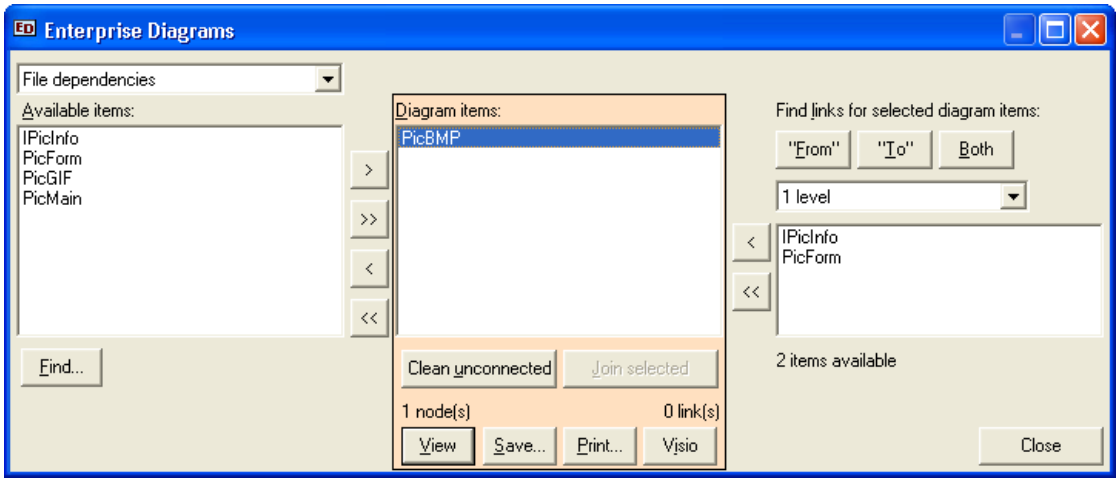
Project dependencies. This diagram shows dependencies between several projects. It is useful with a multi-project analysis.

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 helps you understand the procedure-to-procedure and procedure-to-variable relationships within a single module.

Recursion. The Recursion diagram helps find *indirect recursion*. Indirect recursion means that procedure A calls another procedure, which ultimately calls A again. While recursion is a perfectly valid program design, unintentional or ill-defined recursion can lead to an infinite loop, stack overflow or slow performance. Note: Recursion diagrams do *not* consider *direct recursion* where a procedure calls itself.

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 installed on the same computer as Project Analyzer. The out the help file for the supported Visio versions.

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 the *Clean unconnected* button. 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)
- detect 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 and .NET assembly analysis

Feature requires Enterprise Edition.

COM analysis reads information stored in a type library in a COM file: .dll, .exe or .ocx. A COM file includes a type library that describes the public interface that others can call. A type library stores information on the public classes, methods, properties, parameters, events, user-defined types, constants, Enums and aliases available in the COM file.

.NET assembly analysis reads the types available in a binary .NET assembly: .dll or .exe. As with COM, this information includes classes, methods, properties, constants and other things that external programs and components can access.

To include binary COM/.NET files in the analysis, simply select them 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.

When you include a binary COM or .NET assembly file in your analysis, its contents appear in Project Analyzer as regular classes, procedures, events, Enums etc. You get to see how the public interface of the binary is being accessed by your VB code.

It's important to understand that the information available for binary files is limited to their external interface. You can't see the source code of the binary file or the internals that actually implement what the binary does. Thus, if you have the VB source code for a COM/.NET binary available, it's better to analyze the source rather than the binary. Project Analyzer is not a disassembler!

6.3 DLL file analysis

Feature requires Enterprise Edition.

DLL analysis reveals the available procedures in a conventional library file. 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. It also displays declare syntax information when available.

How is DLL analysis different from COM/.NET assembly analysis? The difference is in the amount of detail. The information stored in a conventional non-COM, non-.NET DLL is limited to a list of callable procedures. The actual procedure parameters are unknown. Even the procedure name is optional. Several DLLs include callable anonymous procedures that are only known by their ordinal number. This is why you need to have the API declaration available in order call the DLL. Project Analyzer comes

with a built-in API database. This lets it show the declaration format of a large number of Windows API functions.

Reading the libraries

To enable DLL analysis. simply check the corresponding checkbox 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.

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.

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 its *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 code review rules

You can monitor a couple of 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 can find flaws with duplicated code analysis. Code is repeated where different code should actually have been used. Two different routines may inadvertently contain the same code.

How the duplicate code analysis works

The analysis attempts to locate identical lines of code. If it finds an exact match longer than a predefined minimum number of lines, it is reported as a duplicated block. Any difference, even if it's just one character, will end the duplicated 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 were not 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 *VB.NET Compatibility Check*, simply analyze a classic VB 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 a particular feature. 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 upgrading 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 Enterprise 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. See the VB.NET documentation for more help on migration issues.

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. Macros run unattended, so you can even run them from the command line or from a batch file.

Macros are described in detail in the help 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 how few files you analyze.

6.7 Project Metrics

Feature requires Enterprise Edition.

To monitor their programming efforts, software engineers often use some simple metrics such as 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

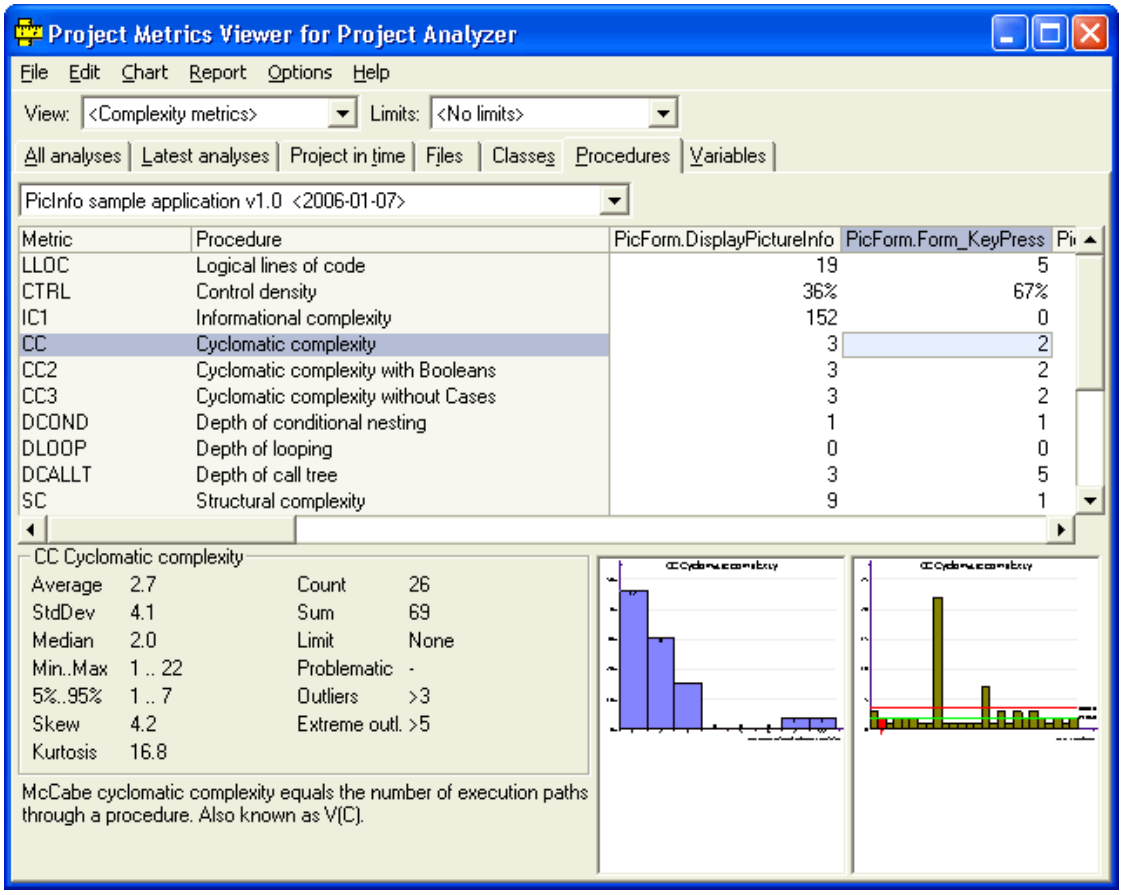
1. Analyze a project, project group or solution.

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.

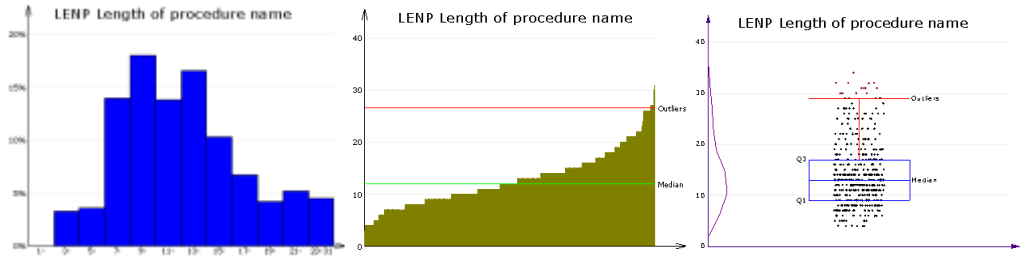
3. Run Project Metrics Viewer (*mtxview.exe*). You can find it in the Enterprise menu. The viewer allows you to view the stored metrics, compare projects and follow the development over time.

Project Metrics Viewer

Project Metrics Viewer is your user interface into the .mtx files you saved. Use it to understand the status of a project, track development between versions and compare programs.



Project Metrics Viewer displays metrics statistics and charts. This way you get deep knowledge of your system.



Sample metrics charts

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.

The following few pages list the metrics available. You can find a detailed description of each metric in the help file. There is also more discussion on the correct use of metrics.

List of supported source code 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: procedure headers, variable and constant declarations, all statements outside procedures.
STMTx	Executable statements	Number of executable statements. A statement is executable 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
SDENS	Statement density	Average number of statements per logical line. SDENS=STMT/LLOC
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 blocks.
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.
dUDTS	Number of dead user-defined types	Number of unused user-defined types, that is, Type and/or Structure blocks.
dENUMS	Number of dead Enums	Number of unused enumeration names.
dENUMCS	Number of dead Enum constants	Number of unused enumeration constants.
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 slightly 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 leaf 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 parameterized 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: procedure headers, variable and constant declarations, all statements outside procedures.
STMTx	Executable statements	Number of executable statements. A statement is executable 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: procedure headers, variable and constant declarations, all statements outside procedures.
STMTx	Executable statements	Number of executable statements. A statement is executable 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 counts 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: procedure headers, variable and constant declarations, all statements outside procedures.
STMTx	Executable statements	Number of executable statements. A statement is executable 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 is not all there is. Read on to find out more.

7.1 Introduction to other features

This chapter introduces you to a number of other features in Project Analyzer. See the help file for more information on these features.

Archive command in the File menu can zip up the files of a project in a .zip file.

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.

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 33) is enabled in the Enterprise Edition, the report includes more details on the DLL procedures. As a bonus, you get a listing of library base addresses. This helps you set the base address of your own library projects. Each library should have a different base address to keep library load times as short as possible.

Menu report lists all menus and menu commands in the project. Shortcuts, hotkeys and HelpContextIDs 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.

String literal report lists all string literals found in the code along with the context they are used in. It also provides an alphabetically sorted list of strings for spell checking or translation.

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 and status of the code being analyzed. The report is a very useful statistical information source about your program.

Variable use report looks for fully or partially dead 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. Classic VB 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 particularly useful with ASP.NET web projects that include graphics as a part of their web content.

7.2 More information

Now that you have learned to use Project Analyzer, you can find more information about Project Analyzer and advanced programming techniques behind the following links.

Online help	*http://www.aivosto.com/project/help*
Programming articles	*http://www.aivosto.com/resources.html*