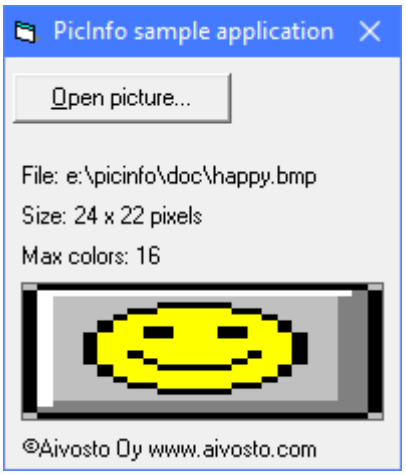
PicInfo sample project for Project Analyzer

Aivosto



PicInfo is a Visual Basic 6.0 sample project that you can analyze with the free Project Analyzer demo. Analyzing PicInfo helps you learn to use Project Analyzer. This document is a "hands on" tutorial into using Project Analyzer on VB6 code. It helps you even if you're analyzing VB.NET or VBA, as the same concepts apply.

Introduction to PicInfo

The PicInfo program is a simple picture information retriever. The user can open picture files (.gif and .bmp) to view the following image information: picture size (x × y pixels) and the maximum number of colors in the image.

PicInfo is a stand-alone program. In addition to the stand-alone executable, one can use PicInfo as an ActiveX server (ActiveX exe). There is a COM interface to query picture size and colors programmatically.

To get started, analyze PicInfo.vbp to view it along with this document. Run Project Analyzer and select PicInfo.vbp to analyze. Choose the default settings (select all source files, no COM or DLL analysis) and press *Analyze*.

Lesson 1. Understanding PicInfo with Project Analyzer

When you need to understand how a program works, run it through Project Analyzer to examine the files, file dependencies, the class hierarchy and how the various modules call each other.

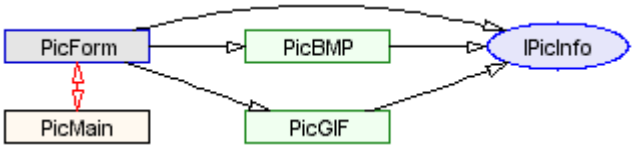
Let's start with a list of the source code files:

PicInfo files	
PicInfo.vbp	VB6 project file
PicForm.frm	Main form
PicMain.bas	Main module with auxiliary routines
PicInfo.cls	IPicInfo interface to query picture information
PicBMP.cls	Implements IPicInfo for .bmp files.
PicGIF.cls	Implements IPicInfo for .gif files.
comdlg32.dll	Windows library for common dialogs

This list above is not something Project Analyzer produces. It's something a developer has to write. As it happens, it's all the existing documentation we have on this program. The developer didn't feel like writing any more docs as he believed the code is self-explanatory. Luckily enough, we can use Project Analyzer to create the missing documentation.

Diagramming

Diagrams are a great way to learn how the VB files work together. You can produce these diagrams via the Enterprise Diagrams feature. Press Ctrl+F7 to run it. The following file dependency diagram shows how the files require on each other:

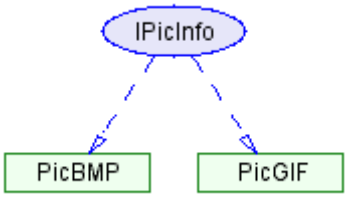


File dependency diagram

As you can see, PicForm.frm requires PicBMP.cls, which in turn requires PicInfo.cls. PicInfo.cls is a *leaf* file that does not depend on any other files. There is a *mutual dependency* between PicForm.frm and PicMain.bas. This dependency is shown with the red arrow. The files require each other. You could not use one without the other.

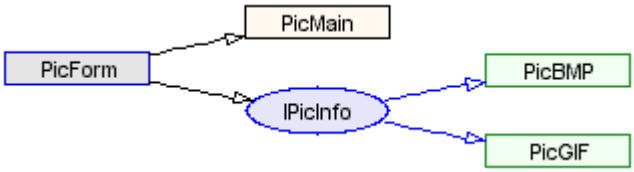
IPicInfo (in PicInfo.cls) is the important interface in this program. This interface allows access to the picture information (image size and colors). The following classes implement IPicInfo: PicBMP for .bmp files and PicGIF for .gif files. One calls IPicInfo.ReadFile to open a file for reading. If this call succeeds,

one can read the properties of IPicInfo to determine information about the picture. You can see the class hierarchy in the following diagram:



Implements diagram

To see how the modules call each other, get the control flow diagram. It shows that PicForm, the main form, calls the IPicInfo interface. The calls are delegated to the implementation classes PicBMP and PicGIF (which will retrieve the actual picture information). PicForm also calls PicMain, which in turn calls comdlg32 (to display common dialog boxes).

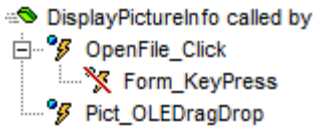


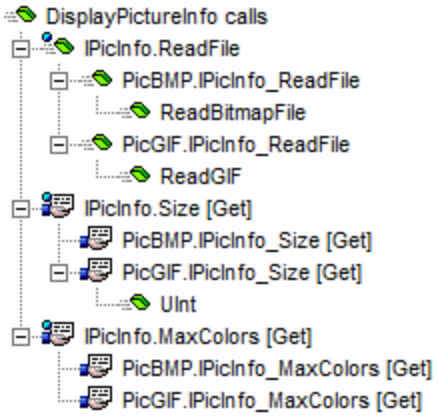
Control flow diagram (procedure calls)

Getting to know the procedures

There are just 29 procedures in this sample program, so it's not very big. We can learn how the procedures work together by getting some call trees. The easiest way is to press Ctrl+T to bring up the *Call tree* window. There is also another alternative, which we will use here.

Look up and select to PicForm.DisplayPictureInfo in the main window. This is the Sub that displays picture information for a given disk file. Bring up its call trees by right-clicking DisplayPictureInfo in the project tree on the left and selecting *Call trees* in the popup menu.





Backward call tree *Forward call tree*

In the call trees you can see calls into DisplayPictureInfo (backwards) and out of it (forwards). Press the "×" key on the numeric keypad to expand the entire trees into view. You can right-click the items in the call trees to move or produce reports.

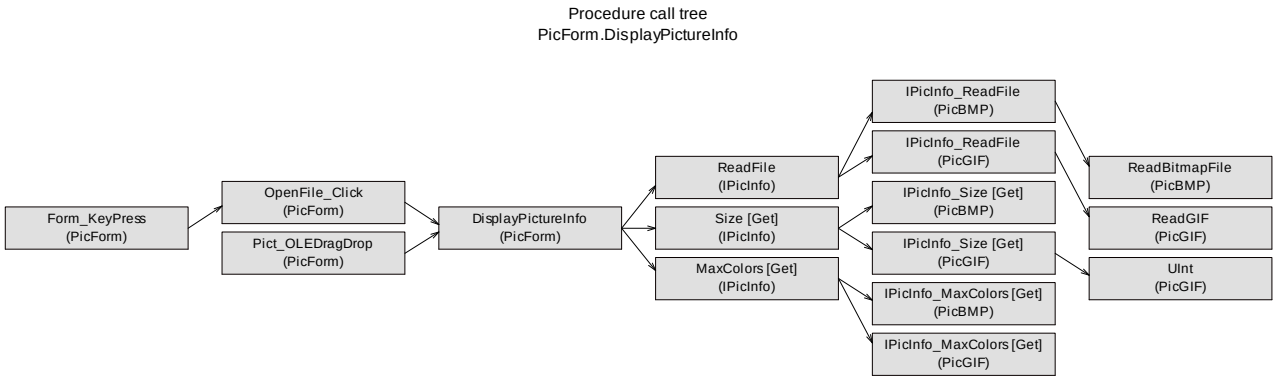
The backward call tree shows that DisplayPictureInfo is called by events OpenFile_Click and Pict_OLEDragDrop.

In the forward call tree, you see how DisplayPictureInfo calls IPicInfo.ReadFile, which in turn calls the respective implementation functions in either PicBMP or PicGIF.

Call graphs

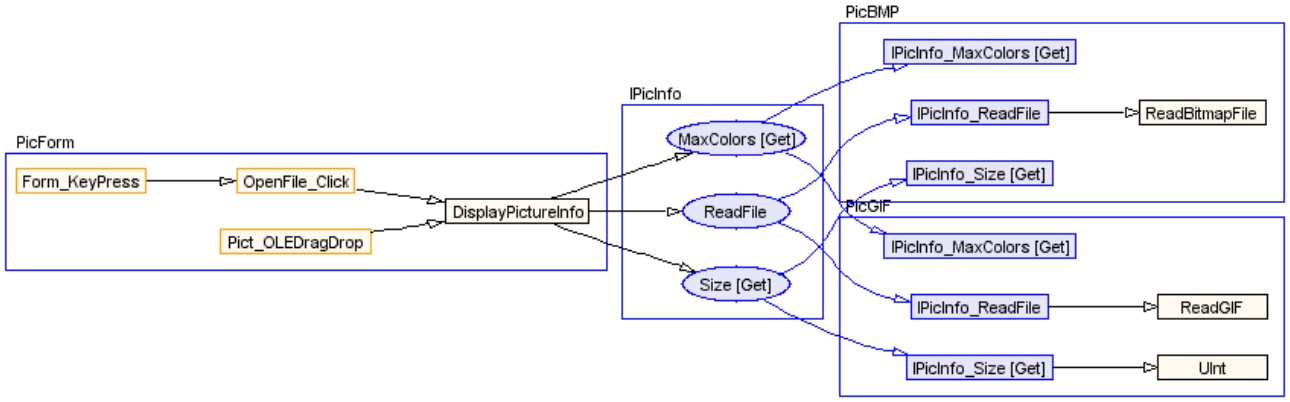
You can also get the procedure call information in two graphical ways. Press F7 to run Project Graph or Ctrl+F7 to run Enterprise Diagrams.

This is the call tree that Project Graph displays for DisplayPictureInfo.



Project Graph: Call tree for Sub DisplayPictureInfo

You can move in the tree interactively. Enterprise Diagrams, on the other hand, creates more complex diagrams. Instead of interactive use, they are suitable for documentation. Here is what Enterprise Diagrams shows for the same procedure:



Enterprise Diagrams: Call diagram for Sub DisplayPictureInfo

Interface and implementation

Notice how Project Analyzer detected the use of interface inheritance (`Implements` statement). It logs calls to the abstract IPicInfo members and "puts the calls forward" to the concrete implementation classes PicBMP and PicGIF. Sub DisplayPictureInfo calls PicBMP and PicGIF via the IPicInfo interface, but it makes no direct calls. If one were to amend this program with PNG or TIFF file support, one could add new classes PicPNG and PicTIFF quite easily without making many changes to DisplayPictureInfo.

Lesson 2. Code review

Now let's see what the automated code review feature of Project Analyzer can dig up. First select the <Default> problem filter. You do this via the *Options* menu, *Problem Options* command. The review findings will be displayed in list at the bottom of the main window. If you don't see the list, press Ctrl+D to bring it back to view.

To make it easier to follow the following text, sort the problems by their type. You do this by clicking on the Problem column of the Problem list.

Dead code detection

As Project Analyzer reviewed the code, it logged what code was used on which lines. It also found several cases of unused, dead code. Even though the sample program is small, there's a lot of dead code in it. This is typical for programming projects; they often have a considerable amount of redundant code in them.

Dead procedures

Dead procedure PicGIF.Version [*Get*]. This property returns the GIF file version. The property is not accessed by the sample program. Project Analyzer flags it as a dead procedure. One could well remove this property. Alternatively, one could add a feature that utilizes the data (by showing the GIF version to the user, for example).

You can notice dead code in several ways:

- When you browse to Version in Project Analyzer, you can see a yellow problem icon to the left of the procedure header line. Try left- and right-clicking the icon to learn more about the problem.
- You can also see that the icon ( ) next to Version in the *project tree* on the left has a red line over it. The red line means dead code. When you select the *Property Get Version* tab at the bottom of the window, then click the *Proc Info* tab, you can verify the dead code status in the Deadness field. In this case it simply says "Dead". Here you can often find more detailed information on the deadness of an item.
- You could also get the *Dead procedures report* in the *Report* menu.

Now let's move on to the other dead procedures.

Dead procedure PicForm.Form_KeyPress. This is an event that never occurs. That's because the Form's KeyPreview property is False. You could enable the event by setting KeyPreview=True.

Dead declaration PicMain.GetSaveFileNameA. This API declaration is not used. One should remove it since the declaration consumes a little space in the executable.

Dead code in exposed interface

The public interface IPicInfo may be accessed by external programs if the project is compiled as an ActiveX exe. External programs may call the Public members of the interface to retrieve picture information.

Property Filename in this interface is not required by the project itself. However, external programs might access it. For this reason, Project Analyzer marks the property as *Dead but exposed*. You can also see a violet "×" in the icon of the property ( ).

Dead but exposed is not listed as a problem with the <Default> problem filter. To keep on the safe side, Project Analyzer assumes that an external program actually utilizes this code. If you remove the code, external programs may fail to run. If you wish to treat exposed dead code as truly dead, you need to select another problem filter (or configure your own).

Either way, you need to open *Problem Options* in the *Options* menu. The quickest way is to use the problem filter <Dead code + exposed>. This filter shows dead code related problems, including those problematic *dead but exposed* cases. To configure your own problem filter, press New to create a new filter and remember to deselect the *Ignore deadness of exposed code* checkbox on the *Dead code* tab in the problem configuration dialog.

Dead and semi-dead variables

Now that you know how to find dead procedures, we can move on to something more advanced: dead variables. A dead variable is one that is not written nor read. As it happens, there are no such variables in our sample project. Great! This is good code.

But wait: How about variables that are read but never written, or that are written but never read? These are cases of semi-dead code. While they may simply be remnants of earlier stages of the project, they can also indicate logical flaws with the code.

Variable written, not read: IsRLE. This variable in PicBMP is *written twice but never read*. Right-click IsRLE in the hypertext code window to bring up References. This way you can review the write locations. IsRLE appears to be a leftover from copy & paste coding. The variable was indeed required earlier when it was used as a flag to tell RLE compressed bitmaps from uncompressed ones. Our sample project doesn't care about the actual value of IsRLE, because we're only interested in the size and colors of the picture. We might as well remove IsRLE.

So, should we remove IsRLE? Having IsRLE in the code doesn't seem to cause any harm. We could have use for it later if we decided to retrieve the compression status of pictures. This kind of thinking is typical to programmers: I don't use this but I'll keep it because I might use it later. The problem with this thinking is that you're leaving code that might become obsolete or stop functioning correctly. Are you sure that the value of IsRLE is correct at all times? Has someone tested it? You could as well comment it out. You can always uncomment it back in later.

Variable read, not written: StoredFilename. This variable in PicBMP is *read but not written*. Now we have found a real bug: we forgot to store any filename in StoredFilename! Right-click StoredFilename and select References to view where it's being read. IPicInfo_Filename is the procedure that reads StoredFilename. Now that we never store anything in the variable, IPicInfo_Filename always retrieves an empty string instead of the real filename. This might cause big trouble if we released this code. To fix the problem, we should add the missing write. We can do this by adding the following line at the beginning of Function ReadBitmapFile:

```
StoredFilename = Filename
```

As the filename is never actually required by the sample program (IPicInfo.Filename never executes in the sample), this bug didn't show up in the test phase. The bug had successfully hidden itself in the dead parts of the program. Because of this possibility, it often pays off to remove dead code to eliminate the lurking bugs from emerging later.

Dead user-defined type fields

Type field written, not read. There are several semi-dead type fields in the program. These fields are write-only: they are written but the data is never read. You can see many *written, not read* fields marked with the ⚠ icon in the declarations section of PicBMP. Let's take BITMAPFILEHEADER as an example.

```
Private Type BITMAPFILEHEADER
      bfType As Integer      ' Specifies the file type, must be BM.
⚠     bfSize As Long         ' Specifies the size, in bytes, of the bitmap file.
      bfReserved1 As Integer ' Reserved; must be zero.
      bfReserved2 As Integer ' Reserved; must be zero.
⚠     bfOffBits As Long      ' Specifies the offset, in bytes, from the beginning of
                             ' the BITMAPFILEHEADER structure to the bitmap bits.
End Type
```

This is the header block that exists at the start of every .bmp file. The first 2 bytes are "BM" in ASCII. The bytes are followed by the file size, 2 reserved fields and an offset field. Fields bfSize and bfOffBits appear to be *written but not read*, while the other fields are in proper use.

Our code reads the .bmp file header in Function ReadBitmapFile (which you can easily find out by right-clicking BITMAPFILEHEADER and selecting References). We read the bfType field to verify it indeed is "BM". It also verifies that the reserved fields are zero. If any of these checks fails, the file is not a bitmap. However, we never read or verify the fields bfSize or bfOffBits, as Project Analyzer already showed us. We simply make the optimistic assumption that these fields contain correct data. Now, if the .bmp file was truncated (in file transfer, for example), our program fails to notice that. We could easily detect that by testing the file size and bfSize for equality. Since this is a picture info application, it would

make sense to detect damaged files and report them to the user. We have thus found out a missing feature in the sample project!

A related problem is *Type field read, not written*. This problem is not found in the sample project. Great coding!

Now we can take a quick look into one of the many subtle ways how Project Analyzer protects you from deleting unused-looking but still good code. See *Type OpenFilename* in file PicMain.bas. This is a user-defined type that seems to be in full use at the first glance. Project Analyzer doesn't report any of these fields as dead code. If you right-click some of the fields (say lpTemplateName) and select References, you can see that many of the fields aren't actually used by the program. Why doesn't Project Analyzer suggest removing these useless fields? That's because the Type is used in an API call (see the `Declare` statement above). If you were to remove a field, the API call would probably fail or crash. This is an API data type; you shouldn't touch it unless you're absolutely sure about what you're doing. Project Analyzer noticed this and didn't report the unused fields as dead code.

Dead constants and Enum values

Dead Enum constant. In PicMain.bas there is a large Enum called EFileDlgFlags, which is used for displaying file dialogs. Some of these enum constants are in use while some are dead. When you take a closer look at the constants you can see that some of them are related to "File Save" dialogs and some are for "File Open" dialogs. Since this program has no "File Save" feature, one should remove the saving related constants. This is recommended because the superfluous values increase the likelihood of errors as the code is developed further.

Dead constant. There are 2 dead constants in PicBMP.cls: BI_RGB and BI_bitfields. These are related to BI_RLE4 and BI_RLE8. In this case they don't indicate any type of a problem. They could be removed, though, in case someone should later wonder what they are used for.

Invisible controls

There's an invisible control on the main form. Can you see that? No, because it's invisible! It's hidden beyond the form's right border. It's a button called SaveFile. Someone has forgotten it there. Maybe the program was designed to save files too, but the feature is not implemented. Project Analyzer reports this as the *Control outside visible area* problem.

The SaveFile button isn't very useful in our sample application. If the user can manage to get it clicked (by pressing Alt+S for "&Save picture..." or by Tabbing onto it), the event SaveFile_Click executes and the program stops running due to the `Stop` statement in SaveFile_Click. One should remove both the button and the event to prevent nasty surprises to the unexpecting user.

Assigned value not used

View the statement `FileNr = 0` at the end of Function ReadGIF in PicGIF.cls. This line is reported as *Assigned value not used*. The local variable FileNr is given a value, but the value is not actually required.

In this case, `FileNr = 0` is simply an extra statement that can be removed. The statement does no real harm, but if it were executed in a loop, it could slow the program down a bit. Thus, this *Assigned value not used* represents an optimization opportunity.

An extra assignment can also represent a real error. Suppose your program writes an important result to variable *x* on line A. On line B, the program uses *x* for some important purpose. On a bad day, you accidentally remove line B. Alternatively you accidentally overwrite the value of *x* by adding a new assignment statement between A and B. Now you would have a real problem, since the important value stored in *x* is no longer accessed. In this case, *Assigned value not used* would represent bad logic in the program. That's a bug!

A few words on coding style

Coding style is something that programmers will never agree on. Project Analyzer reports a wide range of style issues for you to consider. It is up to you to decide whether to follow the suggestions or rather ignore them. Ignoring is easiest by configuring your own problem filters.

Too many parameters?

Function ShowFileOpenDialog in PicMain.bas has 7 parameters. With the <Default> filter, Project Analyzer shows this as a problem called *Too many parameters*. Because of the many parameters, the function seems hard to understand. Even though all the parameters are commented, it takes time to learn what each parameter does and how to use it.

There are several ways to fix this issue. One way is to simply remove some parameters and use defaults. You could remove the DialogTitle parameter, for example, and always use “Open file” as the title. You could provide an alternative simple function with fewer parameters, leaving this function as the advanced option for the power user. Alternatively, you could encapsulate the functionality in a class. The parameters would be properties of the class. Each property would be well described and all properties would have default values. When no value was given, the default would be used. This is the approach of the CommonDialog class in ComDlg32.ocx.

Out parameters

What is not apparent at first sight is the extra complexity added by 2 *in/out* parameters. They are FilterIndex and Flags. These parameters both take a value *and* return one. When you pass a variable to these parameters, the value is subject to change. To see the *out* status, browse to *ShowFileOpenDialog*, click the *Function ShowFileOpenDialog* tab at the bottom of the main window and select *Local vars*. Here you see the *in/out* variables.

Variable/Parameter	Dead	In/Out	Declaration
hwndOwner		in	ByVal hwndOwner As Long
DefaultExtension		in	ByVal DefaultExtension As String
Filter		in	ByVal Filter As String
FilterIndex		in/out	Optional ByRef FilterIndex As Long
InitialDir		in	Optional ByVal InitialDir As String
DialogTitle		in	Optional ByVal DialogTitle As String
Flags		in/out	Optional ByRef Flags As EFileDlgFlags = OFN_OPENDEFAULTS

Project Analyzer can warn about *out* or *in/out* parameters. The warning is not on by default, though. The warning is named *ByRef parameter returns a value*. It is raised each time Project Analyzer detects a ByRef parameter whose value may change. While this may be a desirable function, it can also cause subtle errors when a value changes unexpectedly.

Comments

Although the sample project is rather well documented, there are some areas where the documentation is below average. There are long uncommented blocks in both PicMain.bas and PicBMP.cls. Project Analyzer reports these blocks as *Too many uncommented lines*. Especially those blocks that are related to API calls would require more explanation. Not all developers are familiar with the API data structures in question. A few words would be handy if the code needs to be changed later. One might even benefit from a short description for each structure field.

There is also a shorter piece of code that is lacking comments. It is *Property Get IPicInfo_MaxColors* in PicBMP. There is no explanation on what this function does. It is a short procedure and was not reported by the *Too many uncommented lines* problem rule, which defaults to more lines before triggering a problem.

Project Analyzer can report uncommented procedures regardless of size, even though this rule is not on by default. You can quickly enable it by selecting the <Style> problem filter (*Options* menu, *Problem Options*). A problem icon ⓘ will appear next to *Property Get IPicInfo_MaxColors*. If you right-click the icon, you will see *Uncommented code*.

Miscellaneous problems

Project Analyzer also found the following additional problems in PicInfo using the <Default> problem filter. Here is a short explanation.

Form missing Icon

There is no icon specified for PicForm. It shows the standard VB form icon.

Line too long

There are a few long lines that trigger this warning. You can set the maximum line length in characters and watch for overly long lines.

Case branch(es) missing for Enum

There are a couple of `Select Case` blocks that branch off an Enum value, but not all constants in the Enum are handled.

```
Select Case BMPType
    Case bmpCoreHeader
        IPicInfo_MaxColors = 2 ^ CoreHeader.bcBitCount
    Case bmpInfoHeader
        IPicInfo_MaxColors = 2 ^ InfoHeader.biBitCount
End Select
```

In this example, BMPType is one of bmpCoreHeader, bmpInfoHeader or bmpUnknown. The last value bmpUnknown is not handled, though. The program works incompletely when BMPType has the value bmpUnknown.

One should add `Case bmpUnknown`. Alternatively, `Case Else` could be used to handle unexpected input values.

Optional parameter never passed a value

This rule reviews procedure calls to determine whether Optional parameters are being passed or not. While it's not strictly necessary to pass an Optional parameter, a never-passed parameter is a red flag. Since the parameter never gets any value, other than its default value, why does the parameter exist? Is it a leftover from previous functionality, or is there a feature that's not in use even if it should be?

In the PicInfo sample, there are 4 unused optional parameters in Function ShowFileOpenDialog (file PicMain.bas). The parameters FilterIndex, InitialDir, DialogTitle and Flags appear useful and fully functional, yet they are not being passed to the function. Why is that and is it problem?

These 4 extra parameters are a remnant of copy & paste coding. The functions ShowFileOpenDialog and FileDialog have been copied here from another program. The functions have been tested earlier and it makes no harm keeping these extra parameters around. One could leave out the parameters to simplify the code (perhaps to make it easier to comprehend), but it's not necessary to do so.

An extra Optional parameter can mean useless functionality that could be removed. Alternatively, it can show useful functionality that should be brought back to use. In the ShowFileOpenDialog example, DialogTitle sounds like useful functionality. Instead of removing the parameter, we could add our *Open File* dialog a reasonable title.

Conclusion

All in all, Project Analyzer found more than 40 problems in the small sample program, which consists of just 5 source files and some 700 lines. Many of the issues revealed problematic coding and things to improve on. We found a bug, some redundant code, suggestions for making the code easier to understand, and even ideas for future features.

This was just the beginning. If you enable the problem filter <Strict – Show all problems>, you get to see over 200 other coding issues and more than 100 naming standards warnings. We leave these problems as an exercise to you.

URN:NBN:fi-fe201011152964