



# Migrating a VB6 application in 10 easy steps

VB Migration Partner is a revolutionary software that has changed the way VB6 applications can be ported to VB.NET or C#. Much of the manual labor that is necessary with other migration tools can be avoided (or significantly reduced) if you adopt VB Migration Partner and take advantage of its pragmas and its innovative convert-test-fix methodology.

This document outlines all the steps you take when migrating a VB6 application using our tool. In spite of its highly optimistic title, not all these steps are equally easy. On the other hand, not all of them are necessary in every project.

Here's the summary of what you'll read:

- Run VB6 Bulk Analyzer and get free migration assessment
- Download, install, and register VB Migration Partner
- Prepare the VB6 code for migration
- Run VB Migration Partner to convert the VB6 application
- Generate wrappers for 3-rd party ActiveX controls
- Reach the zero-compilation-errors stage
- Reach the zero-runtime-errors stage
- Achieve functional equivalence with original VB6 code
- Optimize the converted .NET code
- Extend the .NET application

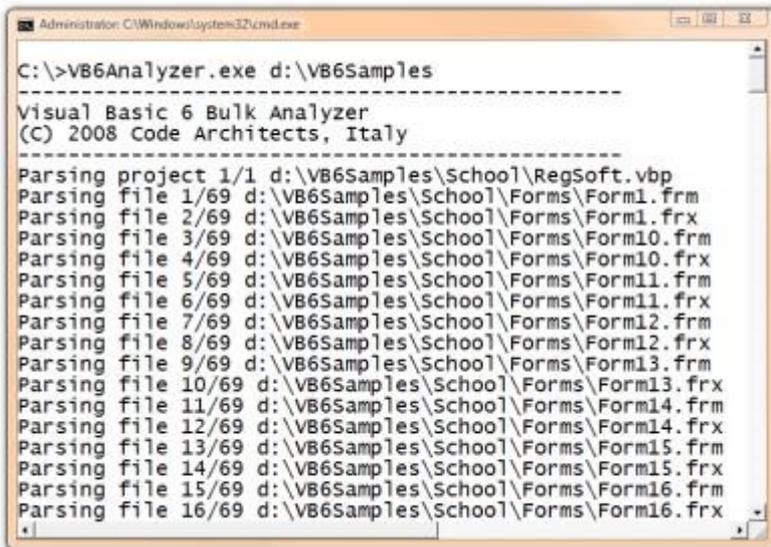
Let's start from the very beginning...

## 1. Run VB6 Bulk Analyzer and get free migration assessment

The first step in any migration process consists in running our VB6 Bulk Analyzer tool on the VB6 project(s) you want to migrate. (You can download this tool from this [page](#).)

VB6 Bulk Analyzer can quickly analyze one or more directory trees, therefore it is quite capable to handle all the projects that are part of your application. It generates a simple textual report that contains the most important information about your projects, including code metrics, the list of type libraries and ActiveX controls it uses, and – above all – the VB6 features that might require some extra efforts during the migration.

## Migrating a VB6 application in 10 easy steps



```
Administrator: C:\Windows\system32\cmd.exe
C:\>VB6Analyzer.exe d:\VB6Samples
-----
Visual Basic 6 Bulk Analyzer
(C) 2008 Code Architects, Italy
-----
Parsing project 1/1 d:\VB6Samples\School\RegSoft.vbp
Parsing file 1/69 d:\VB6Samples\School\Forms\Form1.frm
Parsing file 2/69 d:\VB6Samples\School\Forms\Form1.frx
Parsing file 3/69 d:\VB6Samples\School\Forms\Form10.frm
Parsing file 4/69 d:\VB6Samples\School\Forms\Form10.frx
Parsing file 5/69 d:\VB6Samples\School\Forms\Form11.frm
Parsing file 6/69 d:\VB6Samples\School\Forms\Form11.frx
Parsing file 7/69 d:\VB6Samples\School\Forms\Form12.frm
Parsing file 8/69 d:\VB6Samples\School\Forms\Form12.frx
Parsing file 9/69 d:\VB6Samples\School\Forms\Form13.frm
Parsing file 10/69 d:\VB6Samples\School\Forms\Form13.frx
Parsing file 11/69 d:\VB6Samples\School\Forms\Form14.frm
Parsing file 12/69 d:\VB6Samples\School\Forms\Form14.frx
Parsing file 13/69 d:\VB6Samples\School\Forms\Form15.frm
Parsing file 14/69 d:\VB6Samples\School\Forms\Form15.frx
Parsing file 15/69 d:\VB6Samples\School\Forms\Form16.frm
Parsing file 16/69 d:\VB6Samples\School\Forms\Form16.frx
```

VB6 Bulk Analyzer is a command-line tool that must be run from a prompt window. You can copy it to the root folder of the directory tree that contains your source code, type **VB6Analyzer**, and press the Enter key. The tool supports a few additional options, but you rarely need them. (You can list these option by adding the **/help** option on the command line.)

If your VB6 codebase is spread in different folders, it is essential that you specify all source code folders on the command line, to create a consolidated report:

```
VB6Analyzer c:\myapp\folder1 c:\myapp\folder2 c:\myapp\folder3
```

In all cases, the utility creates a report file named **VBAnalyzer\_Report.txt** in the current directory. The contents of this file should be enough self-explanatory, but you don't need to interpret it yourself. Instead, go to our [Write Us](#) page, fill a few fields with your name and email, and use the dedicated field to upload the report file.

We typically analyze the report as soon as we receive it, of course if our office is open when you send it. (Please account for time zone difference... we live at G.M.T. +1.) We typically reply in a few hours, even though complex reports might take longer.

In return for your report you will receive a very detailed textual assessment of your VB6 application, with tips on how to leverage VB Migration Partner for a smooth migration experience. If a given issue has multiple solutions, the assessment document explains the pros and cons of each option. For example, the assessment often includes recommendations on the best way to migrate your ActiveX controls and your database-intensive code, among the other things.

The mail from Code Architects usually contains also information about the VB Migration Partner license that is most appropriate for your needs. For more information about our license terms, read the FAQ section and download the [End User License Agreement](#) (EULA).

You can purchase the product or just apply for a temporary Trial License. VB Migration Partner Trial Edition is identical to the Full edition except for four details:

- it expires in 14 days

## Migrating a VB6 application in 10 easy steps

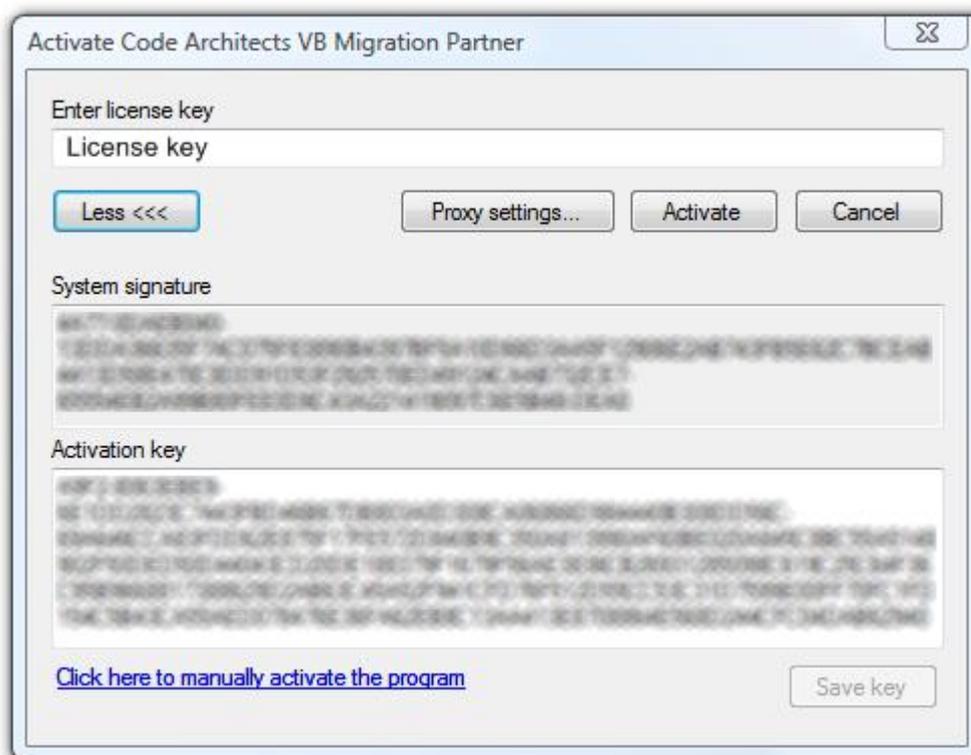
- migrations require an active Internet connection
- a few pragmas have been disabled
- the VB.NET or C# code that it generates can't be expanded with new forms and classes

Notice that users of the Trial Edition enjoy the same degree of support that regular users do, therefore it is an opportunity to test how good and reactive our [tech support](#) is.

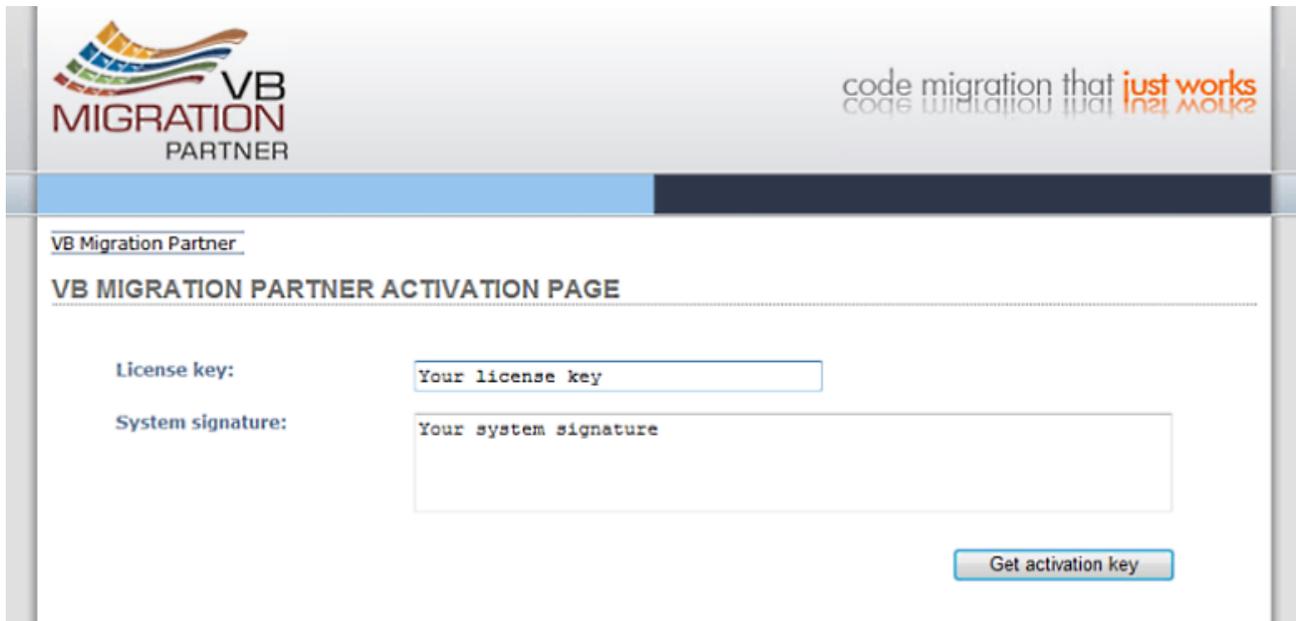
### 2. Download, install, and register VB Migration Partner

This is surely the simplest step in the entire process.

Regardless of whether you purchased the commercial edition of VB Migration Partner or are just testing the Trial Edition, you will receive a download URL. Follow the link, download the file, unzip it, and run the setup procedure.



The first time you run VB Migration Partner you'll have to register your copy. This procedure is fully automated and runs as soon as you attempt your first migration. In some rare cases – for example, if your firewall is very picky about which packets can run over the network – the automated procedure fails and you'll have to perform a manual registration.



Each time you launch it, VB Migration Partner checks whether a new version is available and asks you whether you want to download it. We recommend that you always run the most recent build that is available on our servers.

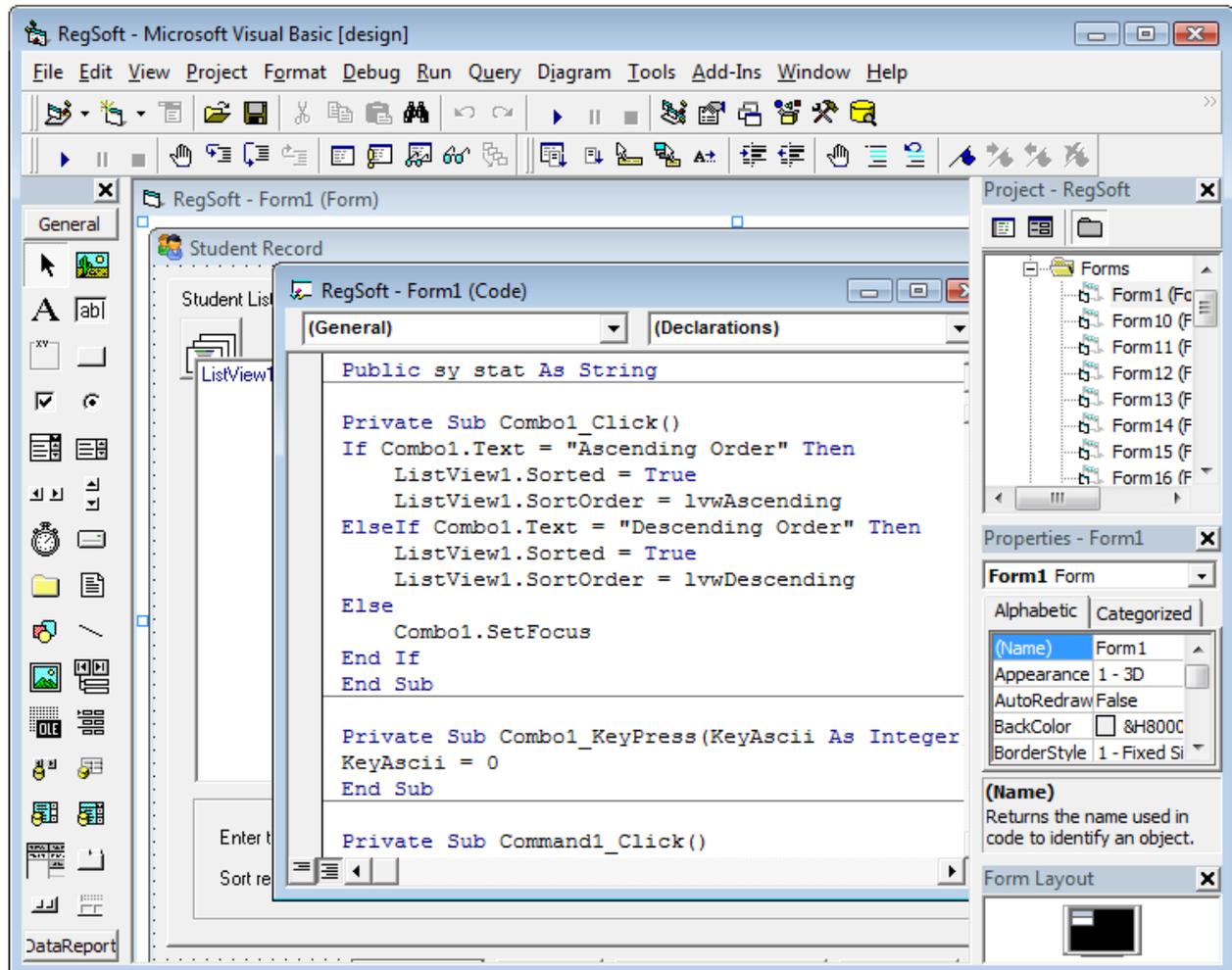
Before using VB Migration Partner you should have a quick read at its [manual](#), with a keen eye on the section related to [migration pragmas](#). If you master pragmas you can make VB Migration Partner do whatever you want, therefore the time you devote to studying them won't be wasted.

Also, we recommend that you give at least a quick look at our knowledge base. Don't read each and every article, just a quick read of their titles should be enough to ring a bell in your brain if and when you bump into a migration issue that we know how to solve.

### 3. Prepare the VB6 code for migration

VB Migration Partner is capable to deal with virtually the entire set of documented VB6 features, therefore this step is seldom required, and you can often migrate your VB6 code without having to prepare it in any way. (In this respect, VB Migration Partner is *much* more powerful than any other VB conversion tool on the market.)

# Migrating a VB6 application in 10 easy steps



The reasons why you might need to edit the VB6 code before attempting the migration are pointed out in the assessment document you receive from the VB Migration Partner team. Here are a few VB6 features that may require some care in this stage.

## Windows API wrapping

VB Migration Partner supports all VB6 features related to Declare statements – for example, As Any and callback parameters are virtually always translated correctly.

However, some calls to Windows API or other external DLLs might use the undocumented VarPtr, StrPtr, and ObjPtr methods. These VB6 methods aren't supported under VB.NET/C# and there is no automated way to migrate them. The report that VB6 Bulk Analyzer generates highlights whether your application uses these keywords. If it does, you should use the Find command in the VB6 IDE to locate each one of them and see whether you can remove individual occurrences.



# Migrating a VB6 application in 10 easy steps

In many cases you can prepare your Declare-intensive VB6 code for migration by wrapping all calls inside methods. For a detailed discussion of this technique, please read the whitepaper “[Tips for smooth migration of Windows API calls](#)”.

## ActiveX EXE projects

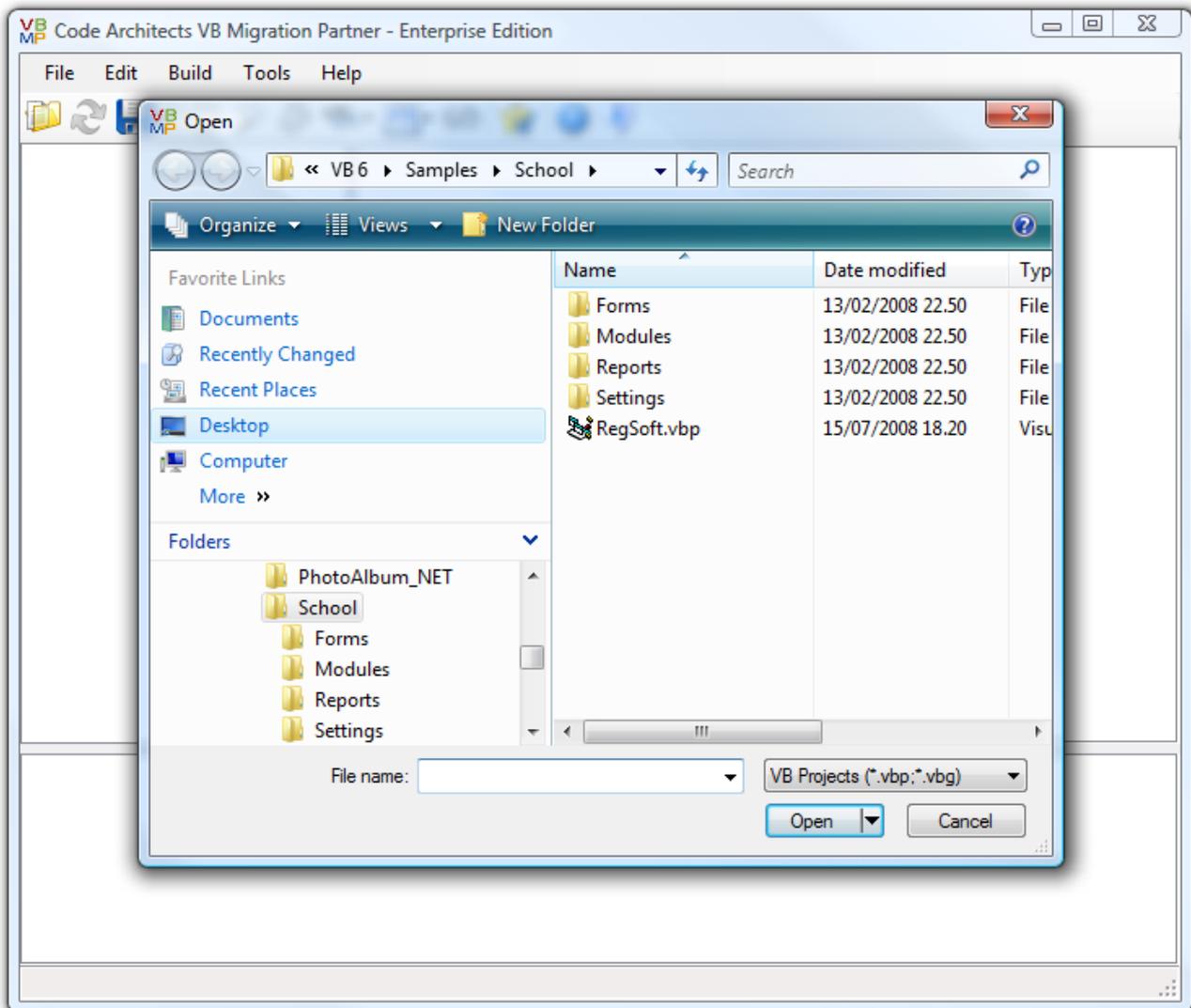
The .NET Framework doesn't support anything that can be considered as equivalent to ActiveX EXEs. VB Migration Partner allows you to choose whether the project should be converted into a standard EXE project or a .NET DLL, therefore you must decide what is most appropriate in each specific case.

There can be several reasons why a VB6 project had to be compiled as an ActiveX EXE. One of the most common was that you needed a stand-alone executable that could also expose objects to the outside, for example an order processing application that exposes an object model for other apps (or plug-ins) to access and manipulate the data using a simplified and consistent interface. In such cases, you can solve elegantly your migration problems by splitting the ActiveX EXE project in two parts: an ActiveX DLL project that exposes the object model and a standard EXE project that shows the user interface and references the DLL.

## 4. Run VB Migration Partner to convert the VB6 application

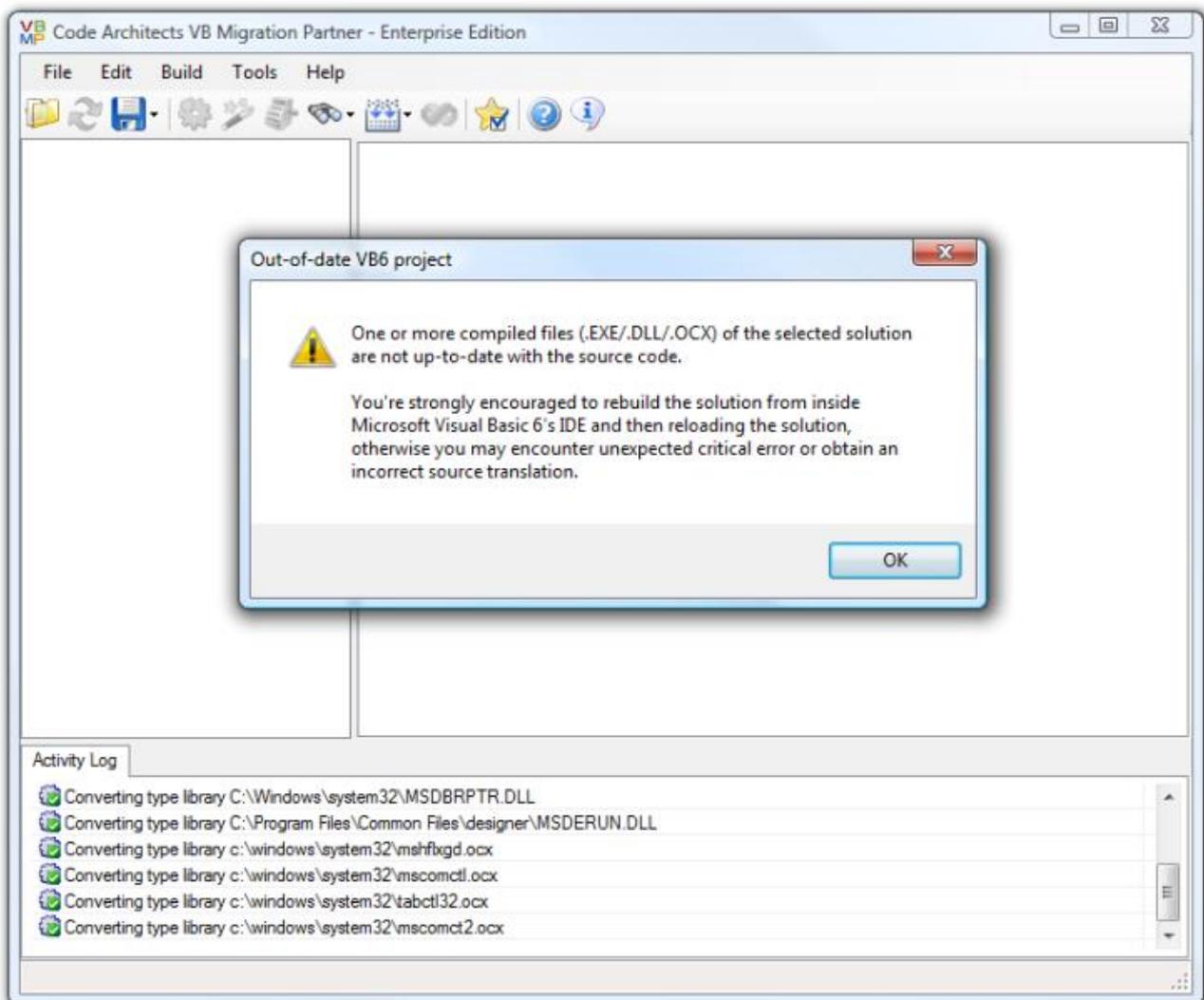
It's now time for the first run through VB Migration Partner. If you have a project group (a .vbg file) that gathers all the VB6 projects of your app, load it in VB Migration Partner. If you have separate .vbp projects, consider whether you should create a .vbg file for the only purpose of migrating them in one single operation. In general, migrating multiple projects in one step delivers better results, because VB Migration Partner can generate better code for inter-project calls. However, if you need to convert more than three or four projects, converting them separately allows you to reduce the number of compilation and runtime errors and have a working .NET project sooner.

# Migrating a VB6 application in 10 easy steps



When loading the project, VB Migration Partner checks that the EXE or DLL executable file is found and its creation date is earlier than all the source files in the project. If this isn't the case, the following message appears when loading the project:

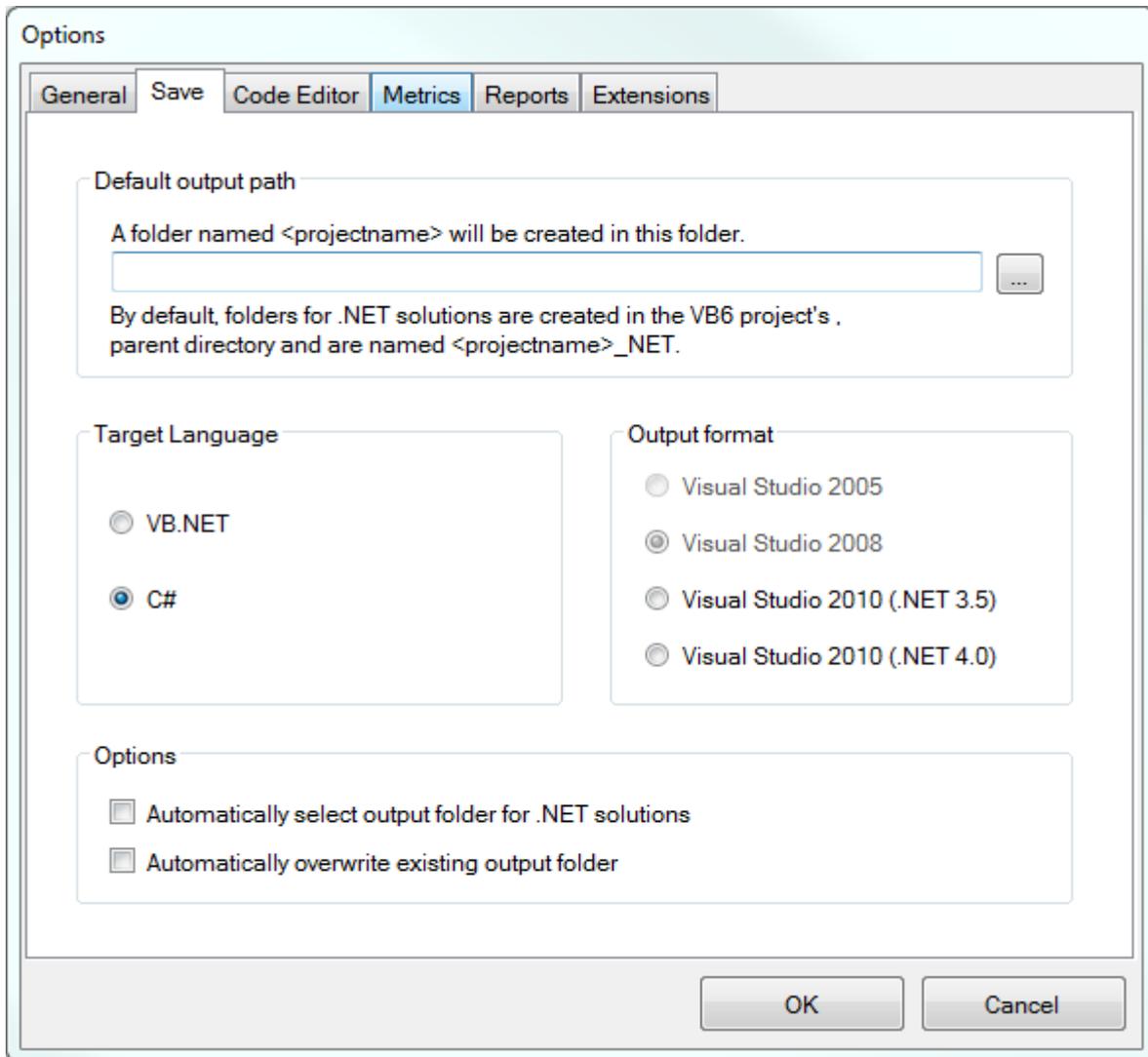
# Migrating a VB6 application in 10 easy steps



VB Migration performs this test because it guarantees that the VB6 source code is syntactically correct and has compiled without errors. If you are sure that all post-compilation changes are OK and haven't introduced any syntax error, just press OK and continue. If you don't want to be bothered by this message in the future, you can disable the test by selecting the "Omit warning if VB6 executable is missing or outdated" option in the General tab of the Tools-Options dialog box.

Once the project is successfully loaded, you should select the target language (VB.NET or C#) and the target version of Visual Studio and .NET Framework, in the Save tab of the Tools-Options dialog box.

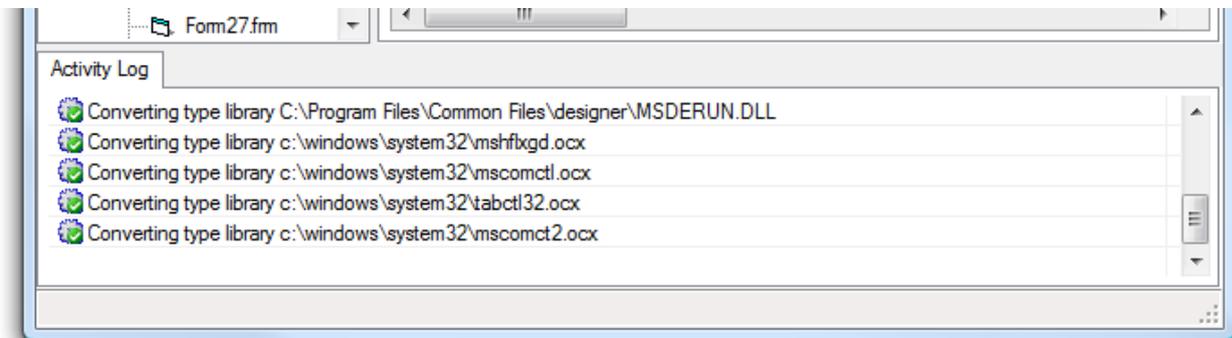
## Migrating a VB6 application in 10 easy steps



Finally you can convert to .NET by either selecting the Built-Convert to .NET command, or clicking on the magic wand icon, or typing the F5 shortcut key. At the end of the conversion process, VB Migration Partner generates as many as six different kind of reports. Each of these reports can be used for a different purpose:

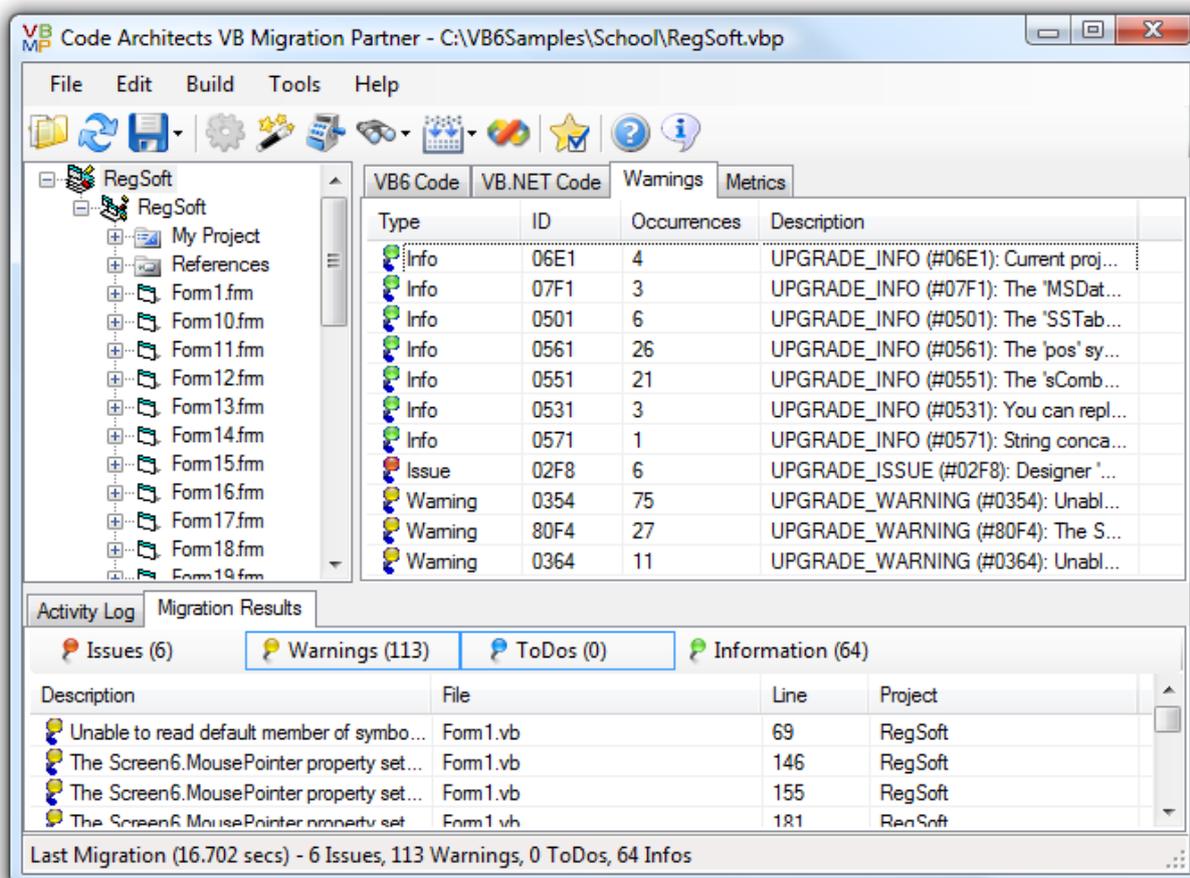
**Activity Log** – VB Migration Partner emits several messages in this window during the migration process. You should check these messages to ensure that all went well, that all type libraries and ActiveX controls were found, that no unrecognized syntax forms were found, etc. If an error message is found in this window, odds are that you tried to convert a VB6 project that can't compile, you didn't installed all COM type libraries correctly, or you run VB Migration Partner on a computer different from the computer where you compiled the VB6 project.

# Migrating a VB6 application in 10 easy steps



**Migration Results** – This window displays all migration warnings and errors. The presence of a warning or error doesn't necessarily indicate that the conversion has failed; it only means that you have to pay close attention to the specified portions of the VB.NET or C# code and ensure that they converted correctly.

You can also get a summary of all migration results – with all issues and warnings in a given project or file – grouped by their type – by clicking on the Warnings tab in the right pane.



**Compilation Results** – You can use the Compile command in the Build menu to have VB Migration Partner run the vbc.exe compiler behind the scenes, gather its output, and display all messages in the

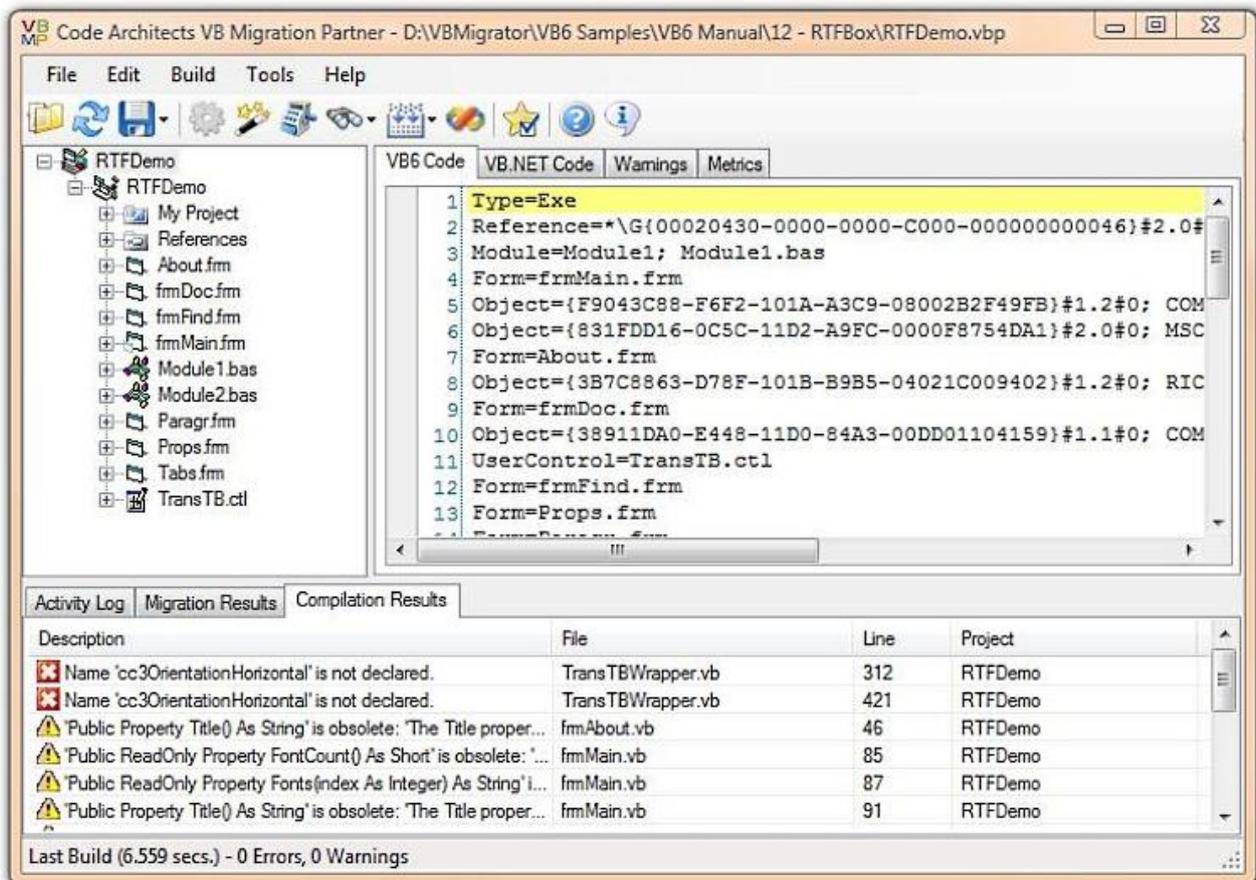
# Migrating a VB6 application in 10 easy steps

bottom pane. If you are lucky and your VB6 code isn't too complex, you'll see just very few errors in this window. (Our statistics show that .NET apps generated by VB Migration Partner have one compilation error for about 1,000 executable statements on the average.)

If you are very lucky, this window will display no error messages and the VB.NET or C# code is ready to compile and run. It doesn't happen often, though, but it happens.

If there are many compilation errors, don't feel discouraged. Most of these errors are likely to have the same cause. They will go away after adding a few well-chosen pragmas.

Also, notice that the earlier Visual Basic .NET compilers have a limit of 102 error messages, after which the compilation is aborted, therefore you will never see more than 102 errors. This means that you still see 102 errors even after fixing some of them, because you then see errors that weren't visible before. A similar thing happens with the C# compiler: when you fix some compilation errors, the C# parser can proceed in analyzing the source code and may find new compilation errors that it didn't display initially.



The screenshot shows the VB Migration Partner application window. The 'Migration Results' tab is active, displaying a table of compilation errors. The table has columns for Description, File, Line, and Project. The errors listed are:

Description	File	Line	Project
Name 'cc3OrientationHorizontal' is not declared.	TransTBWrapper.vb	312	RTFDemo
Name 'cc3OrientationHorizontal' is not declared.	TransTBWrapper.vb	421	RTFDemo
'Public Property Title() As String' is obsolete: 'The Title proper...	frmAbout.vb	46	RTFDemo
'Public ReadOnly Property FontCount() As Short' is obsolete: '...	frmMain.vb	85	RTFDemo
'Public ReadOnly Property Fonts(index As Integer) As String' i...	frmMain.vb	87	RTFDemo
'Public Property Title() As String' is obsolete: 'The Title proper...	frmMain.vb	91	RTFDemo

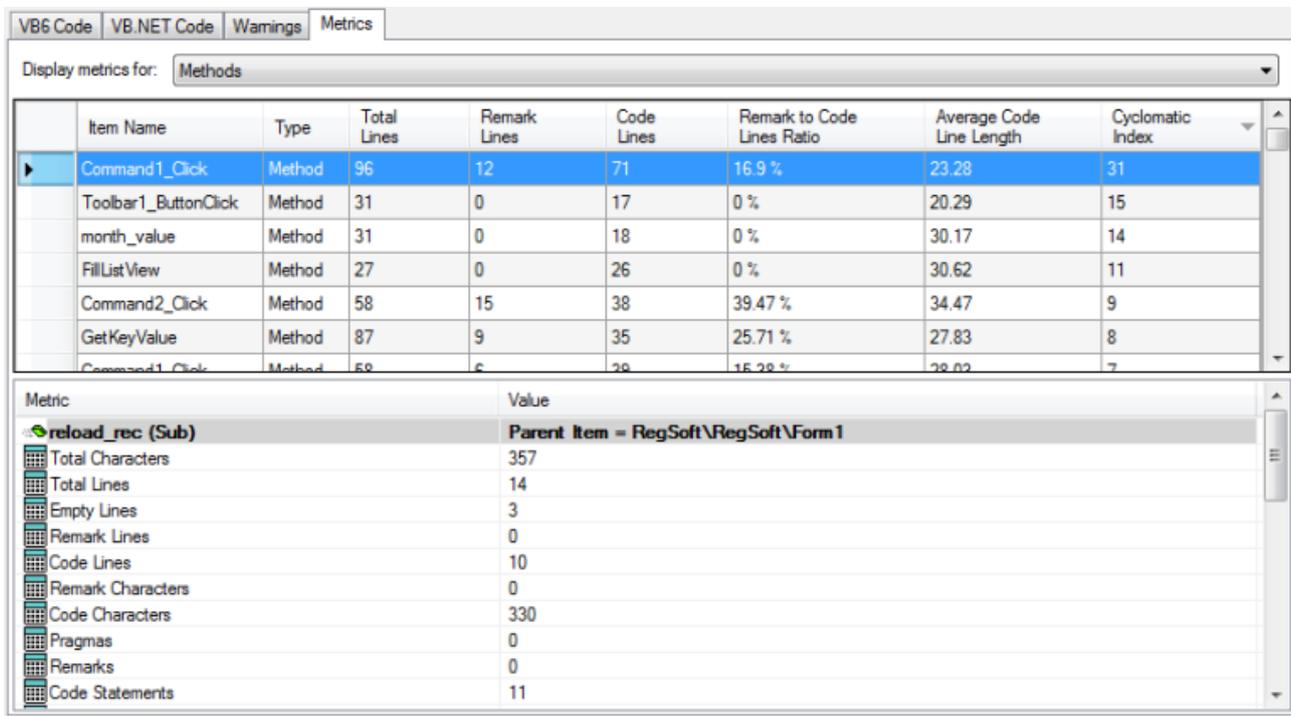
At the bottom of the window, it shows 'Last Build (6.559 secs.) - 0 Errors, 0 Warnings'.

**Code Metrics** – the Metrics tab in the right pane displays stats about the project or file selected in the tree view on the left. This information can be useful for a number of purposes. For example, you may sort projects and files on their size, so that you can start with the simplest project or you can assign each project to a different people in your team.

## Migrating a VB6 application in 10 easy steps

One of the most important values in the table is the **Cyclomatic index**, which measures all the possible execution paths inside a method. For example, a method that contains only an If block has a CI equal to 2; if the Else block contains a Select Case block with five Case sections, then the method's CI is equal to 6, and so forth. The CI is important in migration scenarios because it represents the approximate number of tests you should perform to ensure that the .NET code behaves exactly like the original VB6 code. By sorting all methods in a file on their CI you can quickly see which methods are more complex and should be give more attention during the test and QA phase.

The Metrics tab also show how many “problematic” VB6 items are in a given project or file, including Variant variables, Gosub keywords, On Error statements, auto-instanting (As New) variables, arrays with non-zero lower bounds, and so forth. High values for these counters are indicators that you might need to pay more attention to the corresponding classes and methods.



Item Name	Type	Total Lines	Remark Lines	Code Lines	Remark to Code Lines Ratio	Average Code Line Length	Cyclomatic Index
Command1_Click	Method	96	12	71	16.9 %	23.28	31
Toolbar1_ButtonClick	Method	31	0	17	0 %	20.29	15
month_value	Method	31	0	18	0 %	30.17	14
FillListView	Method	27	0	26	0 %	30.62	11
Command2_Click	Method	58	15	38	39.47 %	34.47	9
GetKeyValue	Method	87	9	35	25.71 %	27.83	8
Command1_Click	Method	58	6	28	15.28 %	28.02	7

Metric	Value
<b>reload_rec (Sub)</b>	
Parent Item = RegSoft\RegSoft\Form 1	
Total Characters	357
Total Lines	14
Empty Lines	3
Remark Lines	0
Code Lines	10
Remark Characters	0
Code Characters	330
Pragmas	0
Remarks	0
Code Statements	11

**Migration message explanation** – If you are new to migrations from VB6, odds are that many migration errors and warnings will look a bit obscure. Or maybe you understand what a message means and still have no cue on how to fix it. In such cases, you can use the **Explain Errors and Warnings** command in the Tools menu to produce a detailed explanation of each migration message and a description of available workarounds.

Notice that this report is created on Code Architects' Web server, therefore it requires a working Internet connection. Only the ID and the number of occurrences of each warning are sent to our server; no information about your application ever leaves your computer.



# Migrating a VB6 application in 10 easy steps



code migration that **just works**  
code migration that **just works**

VB Migration Partner

## MIGRATION MESSAGES

---

**UPGRADE\_WARNING (#80F4):** The `Screen6.MousePointer` property sets or returns the `MousePointer` property of the active form, but only if it's a `VB6Form`.

**[27 occurrences]**

The `Screen.MousePointer` property isn't fully supported. The `Screen6.MousePointer` replacement property sets or returns the `MousePointer` property of the active form, if the active form is a `VB6Form` object; else it returns the default mouse cursor (the arrow).

See also: [Get rid of warnings related to `Screen.MousePointer` property](#)

---

**UPGRADE\_INFO (#06E1):** Current project references the '`typelibrary_name`' COM type library.

**[4 occurrences]**

VB Migration Partner emits one such message for each type library that wasn't replaced by .NET classes. Examples of such libraries are DAO and ADO.

---

**UPGRADE\_INFO (#07F1):** The '`typelibrary_name`' type library is never used in current project. Consider deleting it from VB6 project references.

**[3 occurrences]**

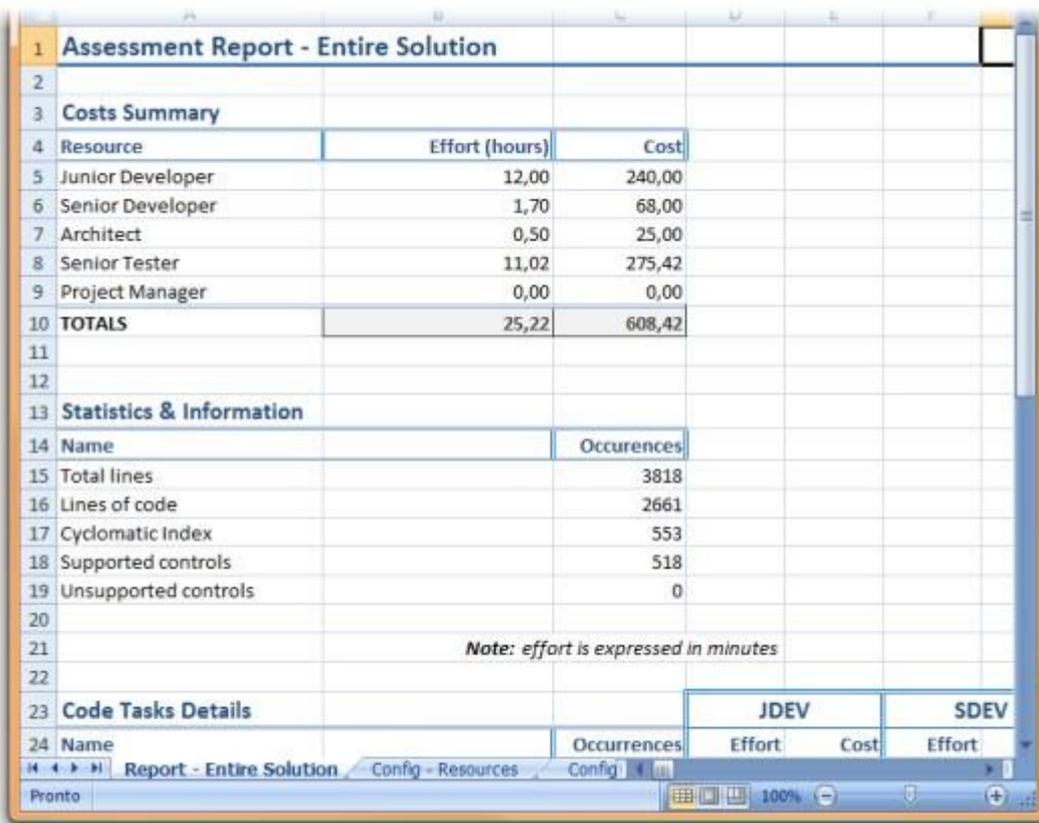
VB Migration Partner has found that one of the type libraries defined in the original VB6 project is never actually used and can be safely removed from the project VB6.

In general, leaving the reference in the project doesn't cause any major problem in the VB.NET code. However, by removing the reference in the original VB6 you can speed up the migration process. This action is especially convenient if you plan to repeatedly convert the project to leverage the convert-test-fix methodology.

**Assessment report** - You can generate a Microsoft Excel worksheet containing a very detailed report of all the migration errors and warnings, together with an estimation of the time required to fix them, get a fully working .NET application, and complete the migration process. The report also contains an estimation of costs, which are evaluated by multiplying the time required for the hourly wage of the people in your team. The report accounts for five professional roles: junior and senior developer, test, architect, and product manager.

All values in the spreadsheet can be customized and you can even provide a custom template if you wish, in the Reports tab of the Tools-Options dialog box.

# Migrating a VB6 application in 10 easy steps



Assessment Report - Entire Solution			
<b>Costs Summary</b>			
Resource	Effort (hours)	Cost	
Junior Developer	12,00	240,00	
Senior Developer	1,70	68,00	
Architect	0,50	25,00	
Senior Tester	11,02	275,42	
Project Manager	0,00	0,00	
<b>TOTALS</b>	<b>25,22</b>	<b>608,42</b>	
<b>Statistics &amp; Information</b>			
Name	Occurences		
Total lines	3818		
Lines of code	2661		
Cyclomatic Index	553		
Supported controls	518		
Unsupported controls	0		
Note: effort is expressed in minutes			
<b>Code Tasks Details</b>			
Name	Occurrences	JDEV Effort	SDEV Effort
Report - Entire Solution			

Before you produce your first assessment report, a word of caution is in order. **All time/cost estimations are based on the assumption that each warning/error must be fixed singularly.** This assumption always brings to very high numbers, and therefore to a very expensive migration.

For example, if the original VB6 code contains many Variant variables, odds are that the .NET code will contain hundreds of warnings #0354 ("Unable to read default member of symbol 'XYZ'. Consider using the GetDefaultMember6 helper method."). According to the standard Excel template, it takes 1 minute from a junior developer and 1 minute for a tester to fix this warning, which means that even a simple VB6 project might require many hours just to get rid of this warning.

In practice, however, you often need only a project-level pragma to avoid a whole set of warnings of same type. In this specific example, the following pragma makes VB Migration Partner generate a call to GetDefaultMember6 helper method, which nicely solves the problem and prevents the generation of #0354 warnings:

```
'## project:DefaultMemberSupport True
```

The bottom line is, be aware that cost/time evaluations you read in assessment reports are often over-estimated and should be taken with a grain of salt.

## 5. Generate wrappers for 3-rd party ActiveX controls

## Migrating a VB6 application in 10 easy steps

This step is necessary only if your project uses one or more ActiveX controls that aren't in VB6 toolbox. For such ActiveX controls it is necessary that you generate and compile a wrapper class, a process that requires the following actions:

1. Run the AxWrapperGen utility to generate the code for the wrapper class. The utility generates a single .NET class library project that contains one wrapper class for each control contained in an .ocx file.
2. Load the generated .NET project in Visual Studio, read the instructions at the top of each wrapper class, and edit the code accordingly.
3. Compile the project and deploy the resulting DLL in VB Migration Partner's install folder.

Editing the wrapper class is necessary only for complex ActiveX controls, such as grids. In most other cases, the code that AxWrapperGen generates compiles correctly at the first attempt.

Notice that these operations are optional, in the sense that you might prefer to immediately skip to next step, without having prepared any ActiveX control wrapper. In this case you'll end up with one or more red rectangles on your VB.NET forms, each one working as a placeholder for an unrecognized ActiveX control. The decision of not converting ActiveX controls with AxWrapperGen can be convenient in some cases, for example if the ActiveX control is used only a few times in the entire project and you plan to replace it with a native .NET control anyway when the migration is completed.

### 6. Reach the zero-compilation-errors stage

In this step you get rid of all the compilation errors in your VB.NET or C# code, so that you can finally launch it.

If VB Migration Partner were a "traditional" conversion tool – like the Upgrade Wizard or any other conversion software on the market – you'd need to fix each and every compilation error in the .NET code. In other words, you wouldn't run the conversion tool any longer and just work on a "snapshot" of the converted code.

Fortunately for you, VB Migration Partner is **not** a traditional conversion tool. Rather, it's a sophisticated piece of software that accompanies you during all the phases in your migration initiative, from the initial planning to the test stage. The key of VB Migration Partner's power is the so-called [convert-test-fix methodology](#).

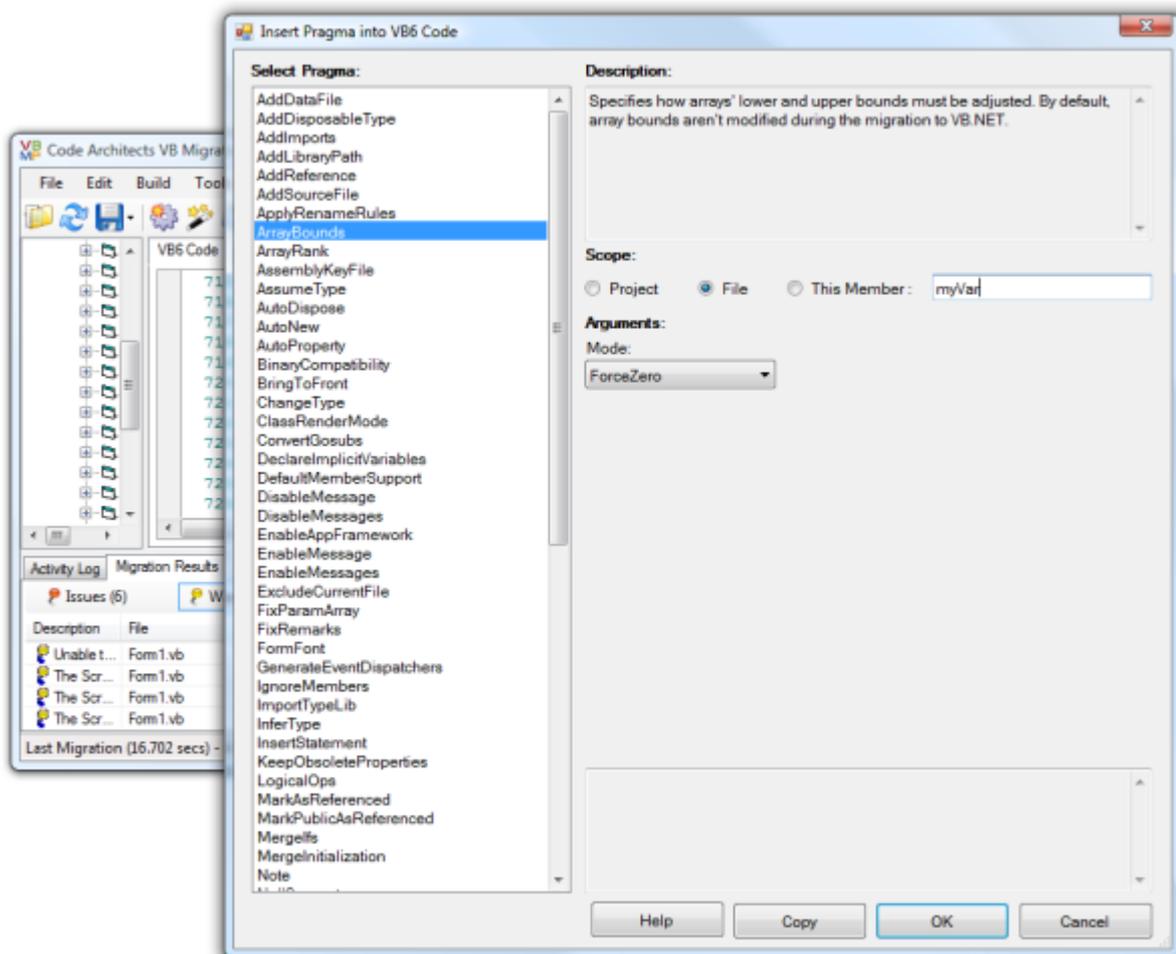
In a nutshell, the convert-test-fix methodology boils down to the following, simple statement: **never, ever manually modify the converted .NET code**. Instead, you apply one or more pragmas to the original VB6 code and re-run the migration. The great thing about pragmas is that you can apply them to the solution-, project-, file-, method- and variable-level. Quite often a few project-level pragmas can eliminate all compilation errors – or at least the large majority of them.

Basically, there are three ways for adding pragmas:

## Migrating a VB6 application in 10 easy steps

1. type the pragma in the VB6 editor, save the file, then switch to VB Migration Partner, reload the project (using the Reload command in the File menu), and re-run the migration.
2. use VB Migration Partner's integrated code editor to insert the pragma in the original VB6 code, save, and re-run the migration.
3. (for project-level pragmas only) use Notepad or another text editor to enter the pragma in a \*.pragmas file in the same folder as the .vbp file, then reload the project using the Reload command in the File menu, and re-run the migration.

You can use the Insert Pragma command (in Edit menu) to have VB Migration Partner help you in selecting the right pragma for a specific purpose. In scenario A, use the Copy button and then paste the code right in the VB6 editor. In scenario B, click on the OK button to insert the code in VB Migration Partner's integrated editor.



To reach the zero-compilation-error stage as quickly as possible, it is essential that you become familiar with the many pragmas that VB Migration Partner. Our online documentation provides many details about pragmas, therefore it makes little sense to rehash all that information in this article. However, our experience is that the following pragmas are most useful in this phase of the migration:

[ImportTypeLib](#) is useful if VB Migration Partner fails to correctly recognize a type library.



# Migrating a VB6 application in 10 easy steps

[SetName](#) allows you to assign a different name to a variable or member, in case the original VB6 name clashes with a VB.NET, C# or .NET Framework member.

[ArrayBounds](#) and [ShiftIndexes](#) can fix problems with arrays having nonzero lower index.

[ArrayRank](#) can be necessary if VB Migration Partner hasn't enough information to infer the number of dimensions of an array.

[PreProcess](#) and [PostProcess](#) allow you to tweak the original VB6 code or the generated .NET code, respectively, to get rid of various errors. (Our [Knowledge Base](#) contains many examples of how these pragmas can be useful.)

[MergeInitialization](#) can be necessary if VB Migration Partner fails to correctly merge a variable with the first assignment to it.

[ClassRenderMode](#) should be used if you are migrating a public classed used as an interface, but VB Migration Partner can't generate the correct interface because the current ActiveX DLL project doesn't include an Implements statement that references the interface.

If you are converting to C#, you may find the following pragmas very useful to reach the zero-compilation-error stage:

[UseDynamic](#) generates dynamic variables that support late-bound calls, rather than cumbersome calls to the VB6Helpers.Invoke method.

[CSharpOption OverloadsForByval](#) generates method overloads that contain by-value parameters instead of ref parameters, which in turn reduces the need to generate temporary variables when invoking those methods.

[CSharpOption ExplicitInterfaceImplementation](#) can help you implement an interface without name collisions.

[CSharpOption ConvertOnErrorResumeNext](#) generates lambda methods for each statement under the scope of an On Error Resume Next statement.

[CSharpOption UseOut](#) attempts to use out parameters instead of ref parameters, which in turn can simplify the code that invokes the method.

Even if the following pragmas don't directly reduce the number of compilation errors, it's a good idea to include them in the early migration attempts, so that you can test the behavior of code that is more similar to its final form:

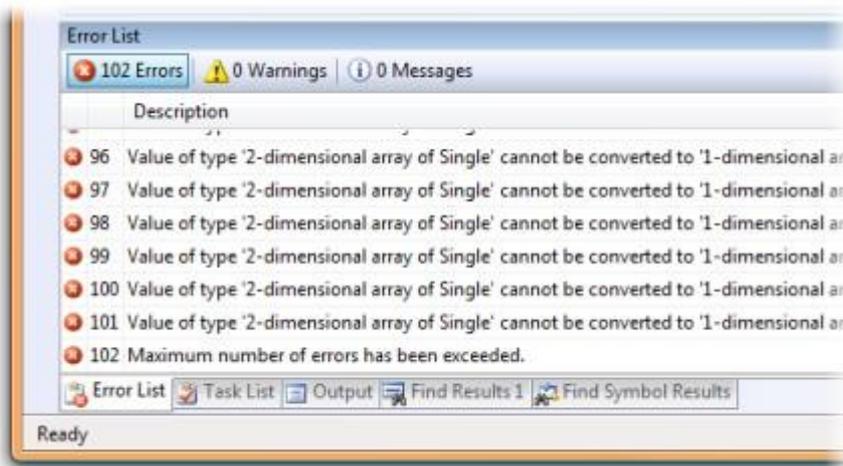
[DeclareImplicitVariables](#) is useful for modules that lack an Option Explicit statement.

[ConvertGosubs](#) attempts to convert Gosubs into calls to separated methods.

[Mergelfs](#) refactors nested IFs into a single IF that uses the AndAlso operator.

Once again, the compile-test-fix methodology is an iterative process: you add one or more pragmas, convert to .NET and then compile the project (or load it into Visual Studio) to have a look at all compilation errors.

# Migrating a VB6 application in 10 easy steps



Here's a simple tip that might significantly reduce your efforts to reach the zero-compilation-error stage: look for the following migration warnings in the generated .NET code:

**#0501:** Member isn't used anywhere in current application.

**#0511:** Member is referenced only by members that haven't found to be used in current application.

**#0521:** Unreachable code detected

These messages typically mark portions of VB.NET or C# code that will be never executed and that should be dropped during the migration process. In general, you'll take care of this code during the optimization step, but if the code contains one or more portions that are never used yet contain many compilation errors, it might make sense to drop those portions now. Instead of deleting the code in the original VB6 code, you can use the [RemoveUnusedMembers](#) and [RemoveUnreachableCode](#) pragmas with the Remarks argument, so that the code will be temporarily commented until you come to a final decision about whether you should really delete it.

## 7. Reach the zero-runtime-errors stage

If the .NET code has no compilation errors, you can run it inside Visual Studio and check that it behaves as expected and that no unexpected errors occur at runtime.

As for previous step, you should abide by the convert-test-fix methodology and refrain from manually editing the VB.NET or C# code. Instead, when you find an unexpected or unwanted runtime behavior you should insert or more pragmas in the original VB6 code and then run VB Migration Partner once again.

The pragmas that are most likely to be useful to complete this stage are:

[DefaultMemberSupport](#) allows to infer a variable's default member at runtime and is useful with Object variables and with Variant variables that contain an object reference.

# Migrating a VB6 application in 10 easy steps

[InsertStatement](#), [ReplaceStatement](#), [ParseReplace](#), [PreInclude](#) and [PostInclude](#) insert or replace VB.NET or C# code in the generated project.

[AutoNew](#) ensures that an auto-instanting (As New) variable preserves its semantics in .NET.

[UseSystemString](#) should be used with all Type...End Type structures that are passed to Declare methods.

[NullSupport](#) should be used for Variant expressions that deal with Null values.

[AddDataFile](#) copies one or more data files into the .NET project folder and avoids “File not found” errors.

[WrapDeclareWithCallbacks](#) is used to prevent the “orphaned delegate” problem that may occur when calling a Declare method that takes a delegate object (the AddressOf keyword).

The following pragmas are only useful if you are converting to C#:

[CSharpOption ConvertOnErrorGoto](#) forces the generation of try-catch block, even if they aren't fully equivalent to the original VB6 code. (it's your responsibility checking that the difference doesn't negatively impact on the way C# code behaves.)

[CSharpOption AssumeConcatenation](#) affects the way VB Migration Partner interprets the “+” symbol when used with Variant or Object variables.

[CSharpOption AssumeStringComparison](#) affects the way VB Migration Partner interprets comparison operators when used with Variant or Object variables.

Not all the runtime errors can be avoided or fixed by means of the above pragmas, though. The causes for unexpected errors can be just too many, but fortunately the VB Migration Partner's support library automatically prevents most of these problems. All the known problems that the support library doesn't handle automatically are documented in our Knowledge Base.

A common problem that you solve in this step is Null values. VB6 and VB.NET/C# greatly differ in how Null values are handled. In VB6 you can pass a Null value to most string functions, including Left, Mid, and Len (but not Left\$ or Mid\$), in which case the function returns Null. The .NET equivalent for Null is DBNull.Value, except you get an exception if you pass this value to any string function. Typically you solve this issue by a combination of the [NullSupport](#) and [ReplaceStatement](#) pragmas.

Keep in mind that unsupported methods and properties don't cause a runtime exception, by default. This behavior is intentional, because it allows you to run and test the .NET code, even if with reduced functionality. However, after completing the first set of tests, you might want to change the behavior by adding the following line of code in the .NET project:

```
' VB.NET
VB6Config.ThrowOnUnsupportedMembers = True

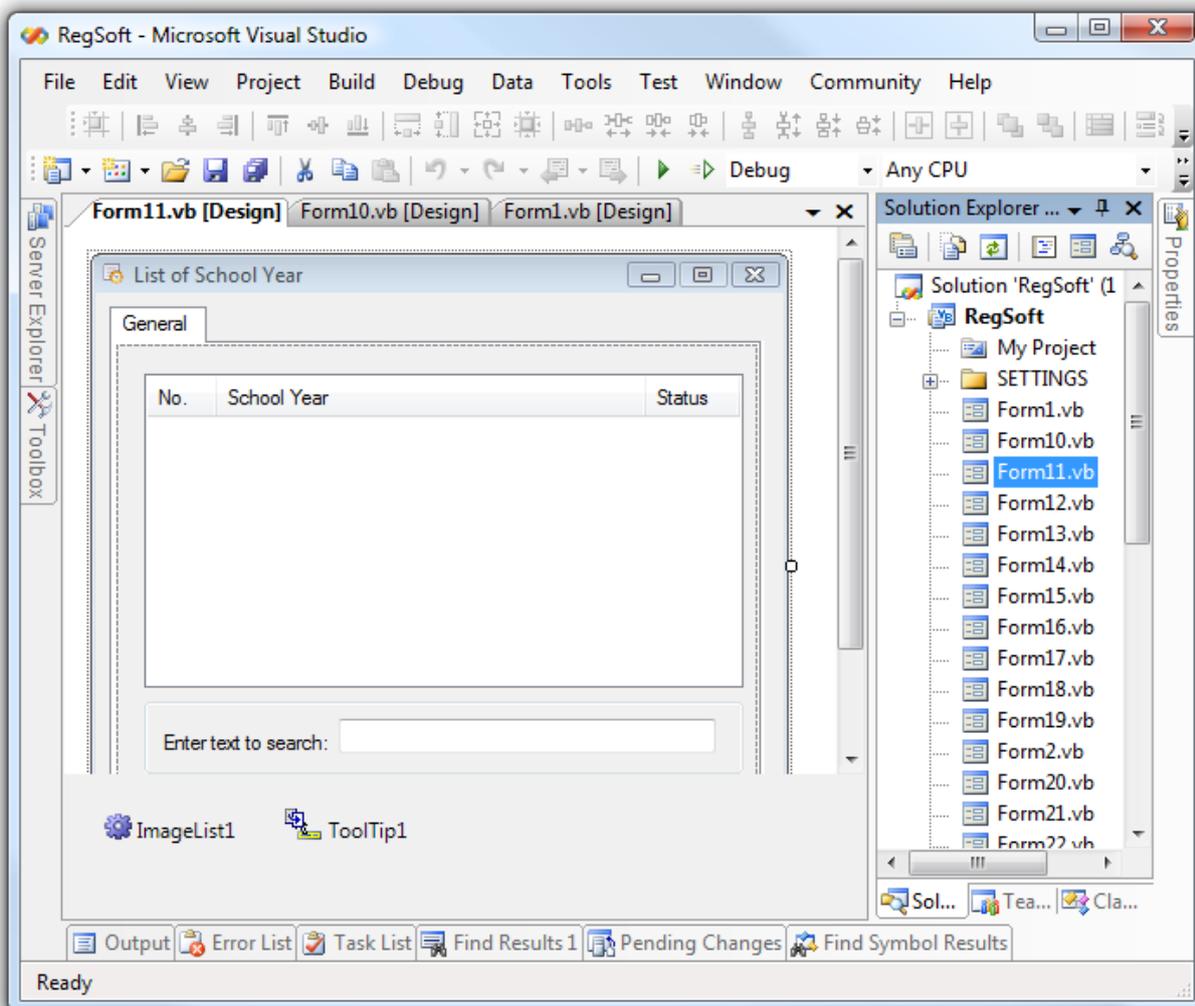
// C#
VB6Config.ThrowOnUnsupportedMembers = true;
```

# Migrating a VB6 application in 10 easy steps

Interestingly, VB Migration Partner's support library provides the ability to automatically generate a log for any fatal exception, which you enable as follows:

```
' VB.NET
VB6Config.LogFatalErrors = True

// C#
VB6Config.LogFatalErrors = true;
```



## 8. Achieve functional equivalence with original VB6 code

After you successfully run the .NET project without any unexpected exception you can focus on refining the .NET code to achieve full functional equivalence with the original VB6 code.

# Migrating a VB6 application in 10 easy steps

VB Migration Partner's support library ensures that this step takes much less time than a traditional conversion tool. However, you might need to add a few pragmas to account for minor differences, especially those related to the user interface.

For example, a common problem in the user interface is caused by .NET not supporting system fonts. VB Migration Partner must therefore convert them into similar (but not identical) true type fonts, which tend to occupy slightly more space. If a caption matches perfectly in a VB6 Label or Button control's client area, odds are that a portion of the caption will be invisible after the migration to .NET.

These are the pragmas that are most useful during this step:

[EnableAppFramework](#) generates Windows XP-like user interface, specifies a splash screen, and more.

[FormFont](#), [ReplaceFont](#), and [ReplaceFontSize](#) allow you to fix issues related to fonts, for example when a caption isn't fully visible inside a Label or Button.

[WriteProperty](#) permits to account for minor user-interface issues, for example to adjust the size or position of a control.

[BringToFront](#) and [SendToBack](#) can be useful if a control is mistakenly hidden by other controls on the converted VB.NET form.

[FixParamArray](#) should be used with methods that modify the elements of a ParamArray argument, to preserve the by-reference semantics.

[UseByVal](#) is often necessary to prevent exceptions when exiting a method that has received read-only properties.

[AddDisposableType](#) and [AutoDispose](#) help you to ensure that objects are released correctly and avoid exception when accessing a resource such as a file or a database connection.

[ThreadSafe](#) is necessary when migrating a DLL that can be used in multi-threaded scenarios, for example by ASP.NET clients.

It is understood that you can be sure that you have fully achieved complete functional equivalence only after stressing the .NET application with an exhaustive QA test, which usually takes a lot of time and energies. For this reason, you might want to postpone the QA test to the last step.

## 9. Optimize the converted .NET code

The ultimate goal of any migration process is to improve the application, for example by making it faster or more scalable. This is the kind of things you accomplish in this last step.

VB Migration Partner can be very helpful in this phase, because it emits warnings that draw your attention on the portions of your code that might require additional care, for example:

**#0531:** You can replace call to unmanaged method with a .NET member.

## Migrating a VB6 application in 10 easy steps

**#0561:** A symbol was defined without an explicit “As” clause.

**#0571:** String concatenation inside a loop. Consider declaring the variable as a `StringBuilder6` object.

**#05B1:** The variable wasn’t explicitly declared.

**#07F1:** Type library is never used in current project. Consider deleting it from VB6 project references.

You can avoid these messages by either modifying the original VB6 code or by inserting a pragma. The pragmas that are most useful in code optimization are:

[RemoveUnusedMembers](#) and [RemoveUnreachableCode](#) delete portions of VB.NET and C# code that aren’t actually used.

[InferType](#) and [SetType](#) allow you to generate a specific type for Object and Variant variables.

[ApplyRenameRules](#) generates code that is compliant with .NET Framework naming guidelines.

[BinaryCompatibility](#) ensures that the generated .NET DLL is compatible with existing COM clients (VB.NET only, not available under C#).

[AssemblyKeyFile](#) generates assemblies that are signed with a strong name.

[AutoProperty](#) converts a field into a property, for better data encapsulation.

[UseTryCatch](#) attempts to replace On Error statements with Try... Catch blocks (VB.NET only, it is assumed by default when converting to C#).

[AddImports](#) allows you to simplify long namespaces in code.

[UseNetMethods](#) attempts to replace calls to support library with calls to native .NET methods (C# only).

Arguably, the most effective optimization technique has to do with string concatenation (see warning **#0571**). This [article](#) explains how you can take advantage of the `StringBuilder6` class to speed up string operations by two or three orders of magnitude.

If you are migrating an N-tier application that is split in many DLLs, you might also want to improve the startup time by the `VB6Config.ParsingAssembliesAtStartup` property to `False`. For more info, read this [article](#).

In the optimization step you should also attempt to remove any dependency from unmanaged code, for example:

- discard any reference to COM type libraries and replace them with references to .NET DLLs.
- modify the wrapper classes generated in step 5, so that the class inherits from a native .NET control rather than the original ActiveX control.
- revise your ADO, DAO, or RDO code and use ADO.NET instead.
- attempt to replace calls to Declare statements with native .NET objects and methods.

# Migrating a VB6 application in 10 easy steps

## 10. Extend the .NET application

Strictly speaking, at this point the migration from VB6 has been completed and you can start selling it or deploy it at your existing customers' site. In practice, however, you often wish to add new features to the application before releasing it to your customers.

A converted application can be extended with new classes exactly like a VB.NET or C# project built from scratch inside Visual Studio. You can add new controls to existing (migrated) forms or add new forms that weren't included in the original VB6 project. The one thing you cannot do is dropping controls from the VB Migration Partner library onto a plain .NET form.

All the controls in VB Migration Partner's library – with few exceptions – inherit from a native .NET control. For example, the VB6TextBox control inherits from the System.Windows.Forms.TextBox control. This [article](#) shows how you can leverage the extra features offered by the .NET controls, thanks to inheritance and the special **NetObject** property.

For example, the VB6ComboBox control exposes several properties and methods that are missing in the VB6 ComboBox – including the DropDownWidth, DropDownHeight, and FormatString properties - because it inherits them from the .NET ComboBox. You can assign these properties in the code-behind portion of the form by using a [WriteProperty](#) pragma, or at runtime using an [InsertStatement](#) pragma.

Of course, you can also just manually edit the .NET form and code, however we recommend that you continue to use pragmas and the convert-test-fix methodology as long as possible, or at least until you are absolutely certain that you can throw the VB6 source away and focus exclusively on the .NET version of your application.

## Wrapping up

This document explains how you can convert a VB6 application of average complexity into VB.NET or C# using VB Migration Partner in ten relatively easy steps. Obviously, not all the steps described here equally easy.

Likewise, each migration project is a story of its own, therefore it's quite difficult to predict how long each step will take. In simpler projects you might be able to skip a few steps entirely, for example manual editing of the VB6 code (step 4) or the optimization phase (step 10).

In all case, the most effective route towards a successful migration begins when you [email us](#) the report from the VB6 Bulk Analyzer. From that point on, we can give you a hand.