Visual Basic Upgrade Companion and Upgrade Wizard: Comparative Summary

VBUC v2.2

December 2008

Index

VBUC vs. UW: Comparative Summary

The Upgrade Wizard is a migration tool that ships with Microsoft Visual Studio .NET 2003, 2005 and 2008 platforms. This tool was specifically designed to assist in the upgrade process from Visual Basic 6.0 to Visual Basic.NET. It partially automates the migration from VB6 to VB.NET by performing the refactoring needed to adapt most of the simplest expressions and commands found in Visual Basic 6.0. It also contains mappings from inherent VB6 libraries to .NET compliant equivalents. So it's basically a tool used to sketch the path to follow while upgrading from VB6 to VB.NET, working as an assistant to demonstrate all the challenges during the whole migration process. This way, the programmer will know where all the hardest spots while moving to the .NET framework are. Beside the issues identification, the Upgrade Wizard applies some automated translations over the VB6 code, reducing the time needed for the upgrade process by generating target source code.

Alternatively, the Visual Basic Upgrade Companion (VBUC) is a significantly more powerful migration tool, also dedicated to upgrade Visual Basic 6.0 applications to the .NET platform. To start with, the VBUC is able to generate C# source code as well as VB.NET. In the end, the Visual Basic Upgrade Companion's architecture, scope, and target audience are extremely different from the Upgrade Wizard's.

The Upgrade Wizard is a valuable tool for VB6 to .NET migration projects; however, there are several areas where the upgrade process can be highly improved. ArtinSoft believes that many more code conversions can take place automatically through the use of artificial intelligence, to greatly reduce the time and cost of any given migration project, and here is where another main feature of the Visual Basic Upgrade Companion kicks in: this tool not only offers additional standard functionality but also excels in terms of customization. It has the ability to be customized according to your needs, increasing even more the percentage of automation of your Visual Basic 6.0 to .NET migration project.

Long time ago, ArtinSoft realized the market required a tool capable of handling larger, more complex migration tasks, with all the implications this carries. Therefore, the Visual Basic Upgrade Companion was conceived, incorporating the vast experience acquired by ArtinSoft during years of successful migration projects.

One of the most common questions is **how much effort will the Visual Basic Upgrade Companion be able to reduce compared to the Upgrade Wizard?** However, there is no standard answer, since it depends on the characteristics of each application to be migrated. Some data from real projects may be useful to shed some light on the subject, but the precise numbers for each case can only be obtained after an analysis of the VB 6.0 code and the customer's target requirements. For example, using the VBUC on a **600,000 LOC** application allowed one of our customers to **save 15,000 hours**. And that was using an older version of the standard tool, that is, without any customization. Plus a comparison between both tools cannot be based only on how much code does each converts automatically from VB6 to VB.NET, or the percentage of migration automation, but also on the quality of the resulting code. As you will read in the last chapters of this document, the Visual Basic Upgrade Companion does further code analysis to detect patterns that can be upgraded to more .NET-like, native structures, making the output more readable and maintainable.

But let's start with some of the productivity enhancements the Visual Basic Upgrade Companion has vs. the Upgrade Wizard that will simplify your Visual Basic 6.0 to .NET conversion experience.

• 3rd Party Library Extensibility

The Upgrade Wizard cannot be modified in any way. On the other hand, the Visual Basic Upgrade Companion can be customized so its core functionality is extended to satisfy your specific VB to .NET migration needs. New extensions augment the translation dictionary without changing the VBUC's core.

This feature allows you to automatically upgrade your specific programming patterns, to add some new functionality on the upgraded application, and, even more, to upgrade the ActiveX controls that you have in the original application to .NET Framework components or newer versions of those specific third-party controls, saving manual effort, time and money.

Below you can find a list with the ActiveX controls that are currently supported by the VBUC:

Original Component\Library	Vendor	Target Component\Library	Vendor
COMSVCSlib	Microsoft	.NET intrinsic	Microsoft
CSTextLib	Crescent Software	C1Input	ComponentOne
FPSpread	FarPoint	Spread	FarPoint
MAPI	Microsoft	.NET intrinsic	Microsoft
Mh3dlblLib	MicroHelp	.NET intrinsic	Microsoft
MSACAL	Microsoft	.NET intrinsic	Microsoft
MSComCtl2	Microsoft	.NET intrinsic	Microsoft
MSComCtlLib	Microsoft	.NET intrinsic	Microsoft
MSComDlg	Microsoft	.NET intrinsic	Microsoft
MSDataGridLib	Microsoft	TrueDBGrid	ComponentOne
MSDBGridLib	Microsoft	TrueDBGrid	ComponentOne
MSFlexGridLib	Microsoft	FlexGrid	ComponentOne
MSMask	Microsoft	.NET intrinsic	Microsoft
MSWLess	Microsoft	.NET intrinsic	Microsoft
MSXML2	Microsoft	.NET intrinsic	Microsoft
MTxAS	Microsoft	.NET intrinsic	Microsoft
Vb.Printer	Microsoft	Helper class	ArtinSoft
RichTextBox	Microsoft	.NET intrinsic	Microsoft
Scripting	Microsoft	.NET intrinsic	Microsoft
SHDocVw	Microsoft	.NET intrinsic	Microsoft
SSActiveTreeView	Sheridan	.NET intrinsic	Microsoft
SSCalendarWidgets	Sheridan	.NET intrinsic	Microsoft
SSDataWidgets_B	Sheridan	TrueDBGrid	ComponentOne
SSDataWidgets_B	Sheridan	UltraWinGrid	Infragistics
SSDesignerWidgets	Sheridan	.NET intrinsic	Microsoft
SSListBar	Sheridan	UltraWinListBar	Infragistics
SSSpliter	Sheridan	.NET intrinsic	Microsoft
Threed	Sheridan	.NET intrinsic	Microsoft
TrueDBGrid50Lib	APEX	TrueDBGrid	ComponentOne
TrueDBGrid60Lib	APEX	TrueDBGrid	ComponentOne
TrueDBGrid70Lib	APEX	TrueDBGrid	ComponentOne
VSFlex7Ctl	VideoSoft	FlexGrid	ComponentOne
VSFlex7LCtl	VideoSoft	FlexGrid	ComponentOne
VSOcxLib	Sheridan	.NET intrinsic	Microsoft
XArrayCustom	APEX	Helper Class	ArtinSoft
XArrayObject	APEX	.NET intrinsic	Microsoft

• Type Inference

An Artificial Intelligence-based type inference engine has been incorporated into the Visual Basic Upgrade Companion, which can infer the most appropriate data types for variable parameters and return values, avoiding the use of "generic" data types (i.e., Object). When an Object or Variant variable is found, the Visual Basic Upgrade Companion declares the variable with the appropriate type and avoids unnecessary migration Errors, Warnings and Issues (EWIs).

By following the type inference approach, the amount of manual work required to check for Upgrade Warnings is drastically reduced.

On the other side, the Upgrade Wizard employs generic data types (object) and has no type inference functionality.

Please read Appendix 1 for some source code examples about type inference.

• Legacy Data Access to .NET

The VBUC upgrades the legacy data access models (ADO, RDO, DAO, ADOR) to .NET equivalents, employing special transformation rules and helper classes (for some specific source/target combinations). On the other hand, the UW generates a target application that still uses those legacy data access models to communicate with the database via COM Interop wrapper calls.

The .NET platform offers a new and completely redesigned collection of classes for data access, which take into consideration modern application requirements for distribution, reliability and scalability, plus a provider-independent data access model (System.Data.Common) which enables developers to write ADO.NET code that works against any .NET data provider.

This new data access model provides the following advantages:

- **Interoperability:** All data in ADO.NET is transported in XML format. The data is provided as a structured text document that can be read by anyone on any platform.
- **Scalability:** ADO.NET promotes the use of disconnected datasets, with automatic connection pooling bundled as part of the package.
- **Productivity:** ADO.NET can improve overall development time. For example, typed DataSets help you work more quickly and allow you to produce more bug-free code.
- **Performance:** Because ADO.NET provides disconnected datasets, the database server is no longer a bottleneck and application performance is improved.

All the legacy data access technologies can be upgraded to the different target technologies following the combinations described here:

Source technology	ADO.NET	ADO.NET using Common
ADO	Yes	Yes
RDO	Yes	Yes
DAO	No	Yes
ADOR	Yes	Yes
Data Controls		
MSRDC	Yes	Yes using RDOCOMMON
ADODC	No	Yes using ADOCOMMON

Please read Appendix 2 for some source code examples about data access.

• C# Generation

The Visual Basic Upgrade Companion is able to generate C# directly from the Visual Basic 6.0 source code, as an alternative to Visual Basic .NET. Since there are several differences between C# and VB6, the Visual Basic Upgrade Companion applies special transformations for:

- C# syntax
- C# language constructions
- Strict typing
- Event declaration and invocation
- • *Structured Error* Handling
- VB6 Modules to classes
- Parameters (optional, default values, by-ref, out, paramarrays)
- Lower bounds to Zero
- Array dims and redims
- Default instances for forms, classes and user controls
- Indexer properties
- With statement
- Return statements
- Interfaces support for C#
- Case sensitive corrections
- Brackets generation for array access
- Variable initialization

The Upgrade Wizard was designed to generate VB.NET source code only, meaning this tool is unable to generate any other .NET compliant language, including C#.

All the Appendixes on this document show source code examples of resulting C# code generated by the VBUC.

• Structured Error Handling

The Visual Basic Upgrade Companion includes features to remove unstructured “spaghetti” code and replace it with structured flow control statements in .NET. All unused labels are removed from the resulting code. Plus the most commonly used *“On Error* ” patterns are currently recognized and replaced by the tool.

The UW upgrades the application using the same *“On Error” statement* patterns that Visual Basic 6.0 uses for error handling. The VBUC generated code is easier to understand and conforms to the coding standards used when programming with .NET languages.

Visit Appendix 3 for a detailed error handling source code example.

• .NET Native Libraries

Instead of upgrading VB6 code using the Visual Basic Compatibility Libraries like the Upgrade Wizard does, the Visual Basic Upgrade Companion promotes the use of .NET native libraries whenever possible.

There are several functions that when converted by the UW, still rely on the Visual Basic compatibility library. Once again, this does not mean that your code will not compile; however, your code will be better off using the native libraries that the .NET framework offers. By using native libraries, you are making your code easier to read, easier to maintain, and in some cases, you will be improving the performance of the application. As an example, in the following table you can compare the end result of upgrading the Left, InStr and Len functions with UW and VBUC:

VB6 Code	VB.NET with UW	VB.NET with VBUC
Left(strvar,1)	VB.Left(strvar,1)	strvar.Chars(0)
Left(strvar,cant)	VB.Left(strvar,cant)	strvar.Substring(0,cant)
InStr(strvar1,strvar2)	InStr(strvar1,strvar2)	strvar1.IndexOf(strvar2) >=0
Len(strvar)	Len(strvar)	strvar.Length

The code generated by the Upgrade Wizard relies on the same functions that were used in Visual Basic 6.0., and therefore uses the Visual Basic compatibility libraries. On the other hand, the VBUC upgrades functions such as Len to the length property of the .NET String class. The table also shows the conversion result of the Left and InStr functions, which uses properties from native classes, improving the resulting application speed by eliminating the overhead involved with interoperability.

• Custom Mappings

The Visual Basic Upgrade Companion is able to upgrade 35+ different ActiveX libraries from Microsoft and other vendors into .NET equivalents, but in some cases there are 3rd party libraries and controls that can make the jump from VB6 to .NET very painful.

The Visual Basic Upgrade Companion Custom Maps extensibility, allows the user to upgrade their non-supported libraries and controls using a simple set of user-defined transformations.

Simple yet powerful, these transformations have been used to upgrade very large and complex ActiveX libraries into inherent .NET libraries, 3rd party equivalents and customized helper libraries.

The overall upgrade process precision can be enhanced by creating personalized mappings to handle conflictive or particular elements as well to enhance and fine tune the existing support for a particular legacy construction.

The Upgrade Wizard that ships with Visual Studio .NET contains a series of maps for intrinsic VB6 libraries to be upgraded into .NET constructions. These mappings are embedded into the application's core and can't be neither modified nor extended.

• MultiProject Conversion

The Upgrade Wizard is able to convert one single VB6 project file (*.vbp) at a time. Hence, if the user wants to upgrade a complex VB6 application with this tool, several considerations must be taken in order to accomplish full functional equivalence.

The Visual Basic Upgrade Companion allows the conversion of multiple Visual Basic 6.0 projects. It performs a separation between the pre-processing and migration stages in order to fix problems like the use of by-ref parameters, interfaces, renaming and typing among the different projects. The pre-processor environment solves the references among projects and simplifies the overall migration process, because those references included in the solution will be calculated and there will be no error messages or warnings about missing members.

The multi-project migration requires a preliminary stage to solve all the possible dependencies between the solution members and to create a common scope among all those items. This situation generates a series of details that needs to be considered by the VBUC to achieve an effective migration of the entire set of projects as a whole, and not as a series of independent conversions. The most critical details are:

- References Resolution
- Global Pre Processing
- Interfaces
- By-ref Parameters
- Renaming
- Typing
- Shared files structure distribution

• User declarations advanced refactoring

The Visual Basic Upgrade Companion manages all the user declarations to perform tasks such as:

- Migrating Visual Basic 6 code to VB.NET or C# using standard Naming Conventions via a compound of common naming conventions for the .NET languages.
- Recognizing conflictive user-declaration names to assign new denominations to the faulty identifier and all of its references.

These tasks improve the resulting source code readability and hasten the manual change stage.

The Upgrade Wizard is able to detect some naming issues found during the upgrade process and applies a simple renaming, but it is not able to refactor the resulting code to comply with .NET naming convention standards.

For more information on user declaration handling please read Appendix 11

• COM class exposure

When this optional feature is enabled, the Visual Basic Upgrade Companion will generate attributes for the COM-exposed classes and their members, in order to keep the resulting assemblies exposed through a COM interface. This enables the resulting components to be called from other non-managed components via COM.

This feature is not available in the Upgrade Wizard, which means this backwards-compatibility feature must be manually implemented if needed.

A brief code sample can be found in Appendix 4:

• Stub generation

To ease the compilation process, the Visual Basic Upgrade Companion generates an empty declaration (stub) in a stub-dedicated source code file and into the converted project for each library element which occurs in the original application and does not have an equivalent in .NET. All the references to these not-converted elements are translated into references to their corresponding stub declarations.

This strategy does not fully resolve the lack of .NET equivalent elements, since the stubs will require manual implementation. However, it saves an important amount of time by achieving the following goals:

- It makes all the not supported elements to compile avoiding an important amount of compilation errors and helping to understand the required manual effort.
- For each not-supported element a solution can be implemented in a unique location in the target source code, instead of requiring changes for all its references.

This functionality is not available in the Upgrade Wizard.

For a brief usage example of the Stub generation feature please read Appendix 5.

• Reflection helper

The Visual Basic Upgrade Companion has a mechanism used to solve references to some objects that in compilation time their members and attributes are not totally clear and inferable for the typing engine. For example, when a variant type is used in the original VB6 source code, the VBUC is capable to infer the most appropriate type to migrate this item into .NET, avoiding the usage of castings and unnecessary conversions in the resulting code. The typing engine could get into trouble if the variable implements complex patterns of late binding in the current code scope, so the VBUC will create a call to the reflection helper that will have the object name and the attribute as parameters. As a result, the compilation will be successfully completed in less time, and the programmer will be able to identify the unsolved reference by looking for all the usages of the reflection helper methods.

This functionality is not available in the Upgrade Wizard.

For a brief usage example of the Reflection Helper please read Appendix 6.

• .NET Enumerations

Another important Visual Basic Upgrade Companion feature, absent in the Upgrade Wizard, is the replacement of numeric literals assigned to several control properties with .NET enumeration equivalents when possible, so that the generated Visual Basic .NET code is more legible and maintainable.

Enumerations source code examples can be found in the Appendix 7.

Comparison between the VBUC and the UW in terms of quality of the generated code

As mentioned before, another big difference between the Visual Basic Upgrade Companion and the Upgrade Wizard is the quality of the code generated by each tool. The objective of this section is to show some of the code quality improvements that the VBUC has over the UW, which produce more .NET-like, native structures and make the output more readable and maintainable.

These improvements are separated in two large areas: data type enhancements and grammar pattern transformations and detailed code improvements.

Data type enhancements

- Collections are upgraded to ArrayList or HashTable depending on their usage.
- Integer to enumerate: when integers are used where an enum name is expected, The VBUC converts it to the corresponding VB6 enum field, and the maps it to the semantic equivalent in .NET.
- IIF expressions: if the two expressions returned by the IIF have equivalent types, only one coercion is generated for the whole IIF invocation. Otherwise each expression is handled in the most appropriate way to make it match the expected type.
- Several cases for conversions between types: objects to scalars (unboxing cases), dates vs. strings, arrays to arrays, fixed strings, octals and hexadecimals, etc.
- Advanced transformations for arrays in most Dim and ReDim scenarios.
- Integer data type variables can be optionally upgraded to Short or Integer depending on their arithmetical usage.
- Enum Advanced conversions for:
    - Enums in comparison operators.
    - Enum values being transmitted as By-Ref.
    - Coercions between Enums and other primitive data types.
- Transformations for fixed size strings.

Grammar pattern transformations and detailed code improvements

- New objects declarations including the initialization value.
- Long "If..ElseIf" constructs are upgraded to the "switch" construct (“Select Case” in VB.NET), in order to improve performance and use better programming practices.
- VBUC uses the "Return" keyword instead of the function name to set the return value within a function. See Appendix 8 for a code example
- "For...Each" blocks are used instead of cycles that use iteration variables. See Appendix 9 for a code sample
- Initialization values for variables are moved to the variable declaration. See Appendix 10 for a code example.
- Typical nested "If" statements used in VB6 to produce short circuit evaluation are upgraded to a single "If" statement with short circuit evaluation operators (AndAlso and OrElse).

Appendix 1: Type Inference

This example demonstrates how the Visual Basic Upgrade Companion infers the appropriate data types for variables.

The following VB6 source code sample contains three non-typed and one fixed-length string variables being used and assigned in a particular way.

Original VB6 Source Code:

```
Private Sub typeInference()
  Dim var1
  Dim var2
  Dim var3
  Dim var4 As String * 50
  var4 = App.Path & "\createDsn.ini" 'Declare your ini file !
  var1 = ArgTypeInference(var2, var3, var4, Len(var4))
End Sub

Public Function ArgTypeInference(ByVal arg1 As Integer, ByVal arg2 As
String, ByVal arg3 As String, ByVal arg4 As Integer)
    MsgBox arg1
    MsgBox arg2
    MsgBox arg3
    MsgBox arg4
    ArgTypeInference = 1
End Function
```

The Upgrade Wizard generates VB6 source code that uses Generic Data Types for non-typed variables. The user must check all the upgrade Errors, Warnings and Issues (EWIs) to fix possible complications.

Code Generated by UW

```
Private Sub typeInference()
      Dim var1 As Object
      Dim var2 As Object
      Dim var3 As Object
      Dim var4 As New VB6.FixedLengthString(50)
      var4.Value = My.Application.Info.DirectoryPath & "\createDsn.ini"
'Declare your ini file !
'UPGRADE_WARNING: Couldn't resolve default property of object var3.
'UPGRADE_WARNING: Couldn't resolve default property of object var2.
'UPGRADE_WARNING: Couldn't resolve default property of object
ArgTypeInference().
'UPGRADE_WARNING: Couldn't resolve default property of object var1.
      var1 = ArgTypeInference(var2, var3, var4.Value, Len(var4.Value))
End Sub

Public Function ArgTypeInference(ByVal arg1 As Short, ByVal arg2 As
String, ByVal arg3 As String, ByVal arg4 As Short) As Object
      MsgBox(arg1)
      MsgBox(arg2)
```

```
        MsgBox(arg3)
        MsgBox(arg4)
'UPGRADE_WARNING: Couldn't resolve default property of object
ArgTypeInference.
        ArgTypeInference = 1
End Function
```

Due to its advanced type inference mechanism, the VBUC's resulting source code contains clearly defined types for most of the user declarations. For this example, var1, var2 and var3 were typed based on their usage (Byte, Integer and String respectively). Also the "`ArgTypeInference`" function return value was upgraded to an explicit call of the "`return`" keyword.

VB.NET Code Generated by VBUC

```
Private Sub typeInference()
        Dim var2 As Integer
        Dim var3 As String = ""
        Dim var4 As New FixedLengthString(50)
        var4.Value = My.Application.Info.DirectoryPath & "\createDsn.ini"
'Declare your ini file !
        Dim var1 As Byte = ArgTypeInference(var2, var3, var4.Value,
var4.Value.Length)
End Sub

Public Function ArgTypeInference(ByVal arg1 As Integer, ByVal arg2 As
String, ByVal arg3 As String, ByVal arg4 As Integer) As Byte
        MessageBox.Show(CStr(arg1), Application.ProductName)
        MessageBox.Show(arg2, Application.ProductName)
        MessageBox.Show(arg3, Application.ProductName)
        MessageBox.Show(CStr(arg4), Application.ProductName)
        Return 1
End Function
```

All the transformations applied in the previous example are available for C# source code generation as well.

C# Code Generated by VBUC

```
private void  typeInference()
{
        int var2 = 0;
        string var3 = String.Empty;
        FixedLengthString var4 = new FixedLengthString(50);
        var4.Value = Path.GetDirectoryName(Application.ExecutablePath) +
"\\createDsn.ini"; //Declare your ini file !
        int var1 = ArgTypeInference(var2, var3, var4.Value,
var4.Value.Length);
}

public byte ArgTypeInference( int arg1,  string arg2,  string arg3,
int arg4)
{
        MessageBox.Show(arg1.ToString(), Application.ProductName);
```

```
        MessageBox.Show(arg2, Application.ProductName);
        MessageBox.Show(arg3, Application.ProductName);
        MessageBox.Show(arg4.ToString(), Application.ProductName);
        return 1;
}
```

Appendix 2: Legacy Data Access

This example demonstrates a simple usage of ADO for data access with a simple query execution.

Original VB6 Source Code

```
Public Sub DataAccessExample()
    Set db = New ADODB.Connection
    Set rs = New ADODB.Recordset
    db.Open "Provider=MSDASQL;DSN=TikkisDb;Password=Password1;"
    rs.Open "SELECT * FROM Table1", db, adOpenKeyset, adLockPessimistic
End Sub
```

The Upgrade Wizard generated source code utilizes the same ADO technology to interact with data sources via COM interop wrapper calls, as seen in this example.

Code Generated by UW

```
Public Sub DataAccessExample()
        Dim rs As Object
        Dim db As Object
        db = New ADODB.Connection
        rs = New ADODB.Recordset
        'UPGRADE_WARNING: Couldn't resolve default property of object
db.Open.
        db.Open("Provider=MSDASQL;DSN=TikkisDb;Password=Password1;")
        'UPGRADE_WARNING: Couldn't resolve default property of object
rs.Open.
        rs.Open("SELECT * FROM Table1", db,
        ADODB.CursorTypeEnum.adOpenKeyset,
        ADODB.LockTypeEnum.adLockPessimistic)
End Sub
```

The Visual Basic Upgrade Companion generated source code employs .NET data access constructions. The user can select from different available techniques to upgrade the data access methodology from the tools ***Profile Manager*** screen. Besides the target language, the user can also select the data access library to be employed (SqlClient or System.Data.Common).

VB.NET Code Generated by VBUC

```
Public Sub DataAccessExample()
        Dim db As New SqlConnection
        Dim rs As New DataSet
        db.ConnectionString =
        "Provider=MSDASQL;DSN=TikkisDb;Password=Password1;"
```

```
        db.Open()
        Dim com As SqlCommand = New SqlCommand()
        com.Connection = db
        com.CommandText = "SELECT * FROM Table1"
        Dim adap As SqlDataAdapter = New SqlDataAdapter(com.CommandText,
        com.Connection)
        rs = New DataSet("dsl")
        adap.Fill(rs)
End Sub
```

VB.NET Code Generated by VBUC using System.Data.Common namespace

```
Public Sub DataAccessExample()
        Dim db As DbConnection =
        AdoFactoryManager.GetFactory().CreateConnection()
        Dim rs As RecordSetHelper = New
        RecordSetHelper(AdoFactoryManager.GetFactory())
        db.ConnectionString =
        "Provider=MSDASQL;DSN=TikkisDb;Password=Password1;"
        db.Open()
        rs.Open("SELECT * FROM Table1", db,
        LockTypeEnum.adLockPessimistic)
End Sub
```

C#.NET Code Generated by VBUC

```
public void  DataAccessExample()
{
        SqlConnection db = new SqlConnection();
        DataSet rs = new DataSet();
        db.ConnectionString =
"Provider=MSDASQL;DSN=TikkisDb;Password=Password1;";
        db.Open();
        SqlCommand com = new SqlCommand();
        com.Connection = db;
        com.CommandText = "SELECT * FROM Table1";
        SqlDataAdapter adap = new SqlDataAdapter(com.CommandText,
com.Connection);
        rs = new DataSet("dsl");
        adap.Fill(rs);
}
```

C#.NET Code Generated by VBUC using System.Data.Common namespace

```
public void  DataAccessExample()
{
        DbConnection db =
        AdoFactoryManager.GetFactory().CreateConnection();

        RecordSetHelper rs = new
        RecordSetHelper(AdoFactoryManager.GetFactory());

        db.ConnectionString =
        "Provider=MSDASQL;DSN=TikkisDb;Password=Password1;";

        db.Open();
```

```
        rs.Open("SELECT * FROM Table1", db,
        LockTypeEnum.adLockPessimistic);
}
```

Appendix 3: Error Handling

The following VB6 extract shows a simple usage of the “On Error” statement to avoid a possible division by zero. The flow control of this source code extract will jump to the “`ErorHandler:`” label in case there is an arithmetic error.

Original VB6 Source Code

```
Public Sub ErrorHandling(arg1 As Integer)
On Error GoTo ErrorHandler
    Dim var1 As Integer
    var1 = 1 / arg1
    MsgBox var1
    MsgBox arg1
    Exit Sub
ErrorHandler:
    MsgBox Err.Description, , "Error"
End Sub
```

The Upgrade Wizard converts it using the same error management statements as in the original VB6 source code.

Code Generated by UW

```
Public Sub ErrorHandling(ByRef arg1 As Short)
On Error GoTo ErrorHandler
      Dim var1 As Short
      var1 = 1 / arg1
      MsgBox(var1)
      MsgBox(arg1)
      Exit Sub
ErrorHandler:
      MsgBox(Err.Description,  , "Error")
End Sub
```

The Visual Basic Upgrade Companion is able to generate VB.NET source code using the .NET “Try Catch” blocks by applying special refactor techniques, as well as the same error management patterns from VB6 if needed. If the resulting language is C# the “try catch” generation is mandatory.

VB.NET Code Generated by VBUC

```
Public Sub ErrorHandling(ByRef arg1 As Integer)
Try
      Dim var1 As Integer
      var1 = 1 / arg1
      MessageBox.Show(CStr(var1), Application.ProductName)
      MessageBox.Show(CStr(arg1), Application.ProductName)
```

```
Catch excep As System.Exception
      MessageBox.Show(excep.Message, "Error")
End Try
End Sub
```

C#.NET Code Generated by VBUC

```
public void  ErrorHandling( int arg1)
{
      try
      {
            int var1 = 0;
            var1 = Convert.ToInt32(1 / ((double) arg1));
            MessageBox.Show(var1.ToString(), Application.ProductName);
            MessageBox.Show(arg1.ToString(), Application.ProductName);
      }
      catch (Exception excep)
      {
            MessageBox.Show(excep.Message, "Error");
      }
}
```

Appendix 4: COM class exposure

In this example, the original VB6 source code was contained in an ActiveX dll.

Original VB6 Code:

```
Public Sub method1()
   MsgBox “Hello world”
End Sub
```

The Visual Basic Upgrade Companion generated source code will include some attributes to expose the resulting assemblies through a COM interface.

Resulting VB.NET Code:

```
Option Strict Off
Option Explicit On
Imports System
Imports System.Runtime.InteropServices
Imports System.Windows.Forms
<ComVisible(True)> _
<ProgId("Project1.Class1")> _
<ClassInterface(ClassInterfaceType.AutoDual)> _
Public Class Class1
      Public Sub method1()
            MessageBox.Show("hello world", Application.ProductName)

      End Sub
End Class
```

Generated attributes for COM visibility

Resulting C#.NET Code:

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;
using VB6 = Microsoft.VisualBasic.Compatibility.VB6.Support;

namespace Project1{

[ComVisible(true)]
[ProgId("Project1.Class1")]
[ClassInterface(ClassInterfaceType.AutoDual)]
public class Class1{

public void  method1(){
      MessageBox.Show("hello world", Application.ProductName);
}}
}
```

Generated attributes for COM visibility

This functionality is not available in the Upgrade Wizard.

Appendix 5: Stub Generation

In this example, the original VB6 code contains the LeftB function which is not supported by the migration tools.

Original VB6 Code:

```
Public Sub method1()
      LeftB "teststring", 5
End Sub
```

The Upgrade Wizard will add an Upgrade Warning about the not supported function but the resulting source code remains using this function.

Code Generated by UW

```
Public Sub method1()
            'UPGRADE_ISSUE: LeftB function is not supported.'
      LeftB("teststring", 5)
End Sub
```

The Visual Basic Upgrade Companion resulting source code will contain an EWI about the lack of support for the LeftB function, and if the Stub generation feature is enabled, will create a stub declaration for this function. The reference to the stub declaration is shown below.

Resulting VB.NET Code:

```
Public Sub method1()
```

```
        method2()
        'UPGRADE_ISSUE: (1040) LeftB function is not supported.
        UpgradeStubs.VBA_Strings.LeftB("teststring", 5)

End Sub
```

The Stub generation is available in C#.NET as well.

Resulting C#.NET Code:

```
public void  method1(){
        method2();
        //UPGRADE_ISSUE: (1040) LeftB function is not supported.
        UpgradeStubs.VBA_Strings.LeftB("teststring", 5);
}
```

Appendix 6: Reflection Helper

In this scenario, the VB6 source code contains a particular assignment pattern for the attributes extraction of the Variant variable “x”. This variable is being assigned to a previously defined type “class1” in some conditions and to the “Form1” type as well.

Original VB6 Code:

```
Private Sub ShowInfo(useForm As Boolean)
   Dim x As Variant

   If useForm Then
      Set x = Form1
   Else
      Set x = New Class1
   End If

   If useForm Then
      MsgBox x.formProp
   Else
      MsgBox x.classProp
   End If
End Sub
```

The Upgrade Wizard’s resulting source code will contain two EWIs about the default property conflicts, because of the assignment to different types and the lack of a type resolution engine.

Code Generated by UW

```
Private Sub ShowInfo(ByRef useForm As Boolean)
        Dim x As Object

        If useForm Then
                x = Me
```

```
      Else
            x = New Class1
      End If

      If useForm Then
'UPGRADE_WARNING: Couldn't resolve default property of object
x.formProp.'
            MsgBox(x.formProp)
      Else
'UPGRADE_WARNING: Couldn't resolve default property of object
x.classProp.'
            MsgBox(x.classProp)
      End If
End Sub
```

The Visual Basic Upgrade Companion attempts to resolve the variables type, but since the assignment will vary at runtime, a specific type cannot be established. For this scenario the VBUC employs the reflection helper to extract the members and attributes successfully.

Resulting VB.NET Code:

```
Private Sub ShowInfo(ByRef useForm As Boolean)
      Dim x As Object
      If useForm Then
            x = Me
      Else
            x = New Class1
      End If

      If useForm Then

      'UPGRADE_TODO: (1067) Member formProp is not defined in type
      Variant.
      MessageBox.Show(ReflectionHelper.GetMember(x, "formProp"),
      Application.ProductName)

      Else

      'UPGRADE_TODO: (1067) Member classProp is not defined in type
      Variant.
      MessageBox.Show(ReflectionHelper.GetMember(x, "classProp"),
      Application.ProductName)

      End If
End Sub
```

This feature is availble for both VB.NET and C#.NET

Resulting C#.NET Code:

```
private void  ShowInfo( bool useForm){
      object x = null;
      if (useForm){
            x = Form1.DefInstance;
      } else{
```

```
            x = new Class1();
      }
      if (useForm){
            //UPGRADE_TODO: (1067) Member formProp is not defined in
            type Variant.
            MessageBox.Show(Convert.ToString(ReflectionHelper.GetMember
      (x, "formProp")), Application.ProductName);
      } else{
            //UPGRADE_TODO: (1067) Member classProp is not defined in
            type Variant.
            MessageBox.Show(Convert.ToString(ReflectionHelper.GetMember
      (x, "classProp")), Application.ProductName);
      }
}
```

Appendix 7: .NET Enumerations

The Visual Basic Upgrade Companion is able to upgrade some control properties assignments to .NET enumeration equivalents, as shown in the following example.

Original VB6 Code

```
Sub Foo()
      num = vbArrow
      Me.MousePointer = num
      Me.MousePointer = 11
      Me.MousePointer = vbArrow
End Sub
```

The Upgrade Wizard generated code will contain the exact assigments done in the original source code.

Code Generated by UW

```
Sub Foo()
      Dim num As Object
      'UPGRADE_WARNING: Couldn't resolve default property …
      num = System.Windows.Forms.Cursors.Arrow
      'UPGRADE_WARNING: Couldn't resolve default property …
      'UPGRADE_ISSUE: Form property Form1.MousePointer does not support
      custom mouse pointers.
      Me.Cursor = num
      Me.Cursor = 11
      Me.Cursor = System.Windows.Forms.Cursors.Arrow
End Sub
```

The VBUC's resulting source code will contain references to the .NET enumerations when possible.

VB.NET Code Generated by VBUC

```
Sub Foo()
      Dim num As System.Windows.Forms.Cursor =
      System.Windows.Forms.Cursors.Arrow
```

```
      Me.Cursor = num
      Me.Cursor = System.Windows.Forms.Cursors.WaitCursor
      Me.Cursor = System.Windows.Forms.Cursors.Arrow
End Sub
```

C# Code Generated by VBUC

```
public void  Foo()
{
      Cursor num = Cursors.Arrow;
      this.Cursor = (int) num;
      this.Cursor = Cursors.WaitCursor;
      this.Cursor = Cursors.Arrow;
}
```

As can be observed, the code that was upgraded with the Upgrade Wizard presents several compilation and runtime problems. For instance, the literal '11' assigned to the Cursor property should be converted to its respective enumeration to make the code compile.

The Visual Basic Upgrade Companion has made various improvements to the converted code. First, it can be observed in the first line of the function that the variable 'num' was correctly identified as type Cursor, as opposed to the type Object that was defined by the Upgrade Wizard. If you look at the line corresponding to the assignment of a constant to the Cursor property, you can also identify that the VBUC has converted the value of '11' to the corresponding enumeration value.

Appendix 8: Return keyword for functions

The Visual Basic Upgrade Companion generates code that features explicit usages of the return keyword, to ensure the resulting source code will look as much .NET like as possible

Original VB6 Code

```
Public Function TrimSpaces(Text As String) As String
      Dim Loop1 As Long, SpaceCheck As String
      Dim FullString As String
      TrimSpaces = FullString$
End Function
```

The Upgrade Wizard uses the same assignation to the function name for the return values.

Code Generated by the UW

```
Public Function TrimSpaces(ByRef Text As String) As String
      Dim Loop1 As Integer
      Dim SpaceCheck As String
      Dim FullString As String
      TrimSpaces = FullString
End Function
```

The VBUC's output uses the Return keyword for VB.NET, as well for C#.NET.

VB.NET Code Generated by the VBUC

```
Public Function TrimSpaces(ByRef Text As String) As String
      Dim SpaceCheck, FullString As String
      Return FullString
End Function
```

C# Code Generated by the VBUC

```
public string TrimSpaces( string Text_Renamed)
{
      int Loop1 = 0;
      string SpaceCheck = String.Empty;
      string FullString = String.Empty;
      return FullString;
}
```

Appendix 9: Foreach block usage

When VB6 *For* loops iterate over collections and some arrays, the Visual Basic Upgrade Companion converts the statement to .NET *ForEach* statements, applying the necessary transformations to the collection/array references inside the *For* body and the counter variables.

Counter variables which are not needed any more after the *Foreach* transformation are removed from the resulting code block.

Original VB6 Code

```
Private Sub GetAuthorList()
      While (Not rs.EOF)
            List1.AddItem rs.Fields("Author").Value
            rs.MoveNext
      End While
End Sub
```

Code Generated by the UW

```
Private Sub GetAuthorList()
      While (Not rs.EOF)
            List1.Items.Add(rs.Fields("Author").Value)
            rs.MoveNext()
      End While
End Sub
```

VB.NET Code Generated by the VBUC

```
Private Sub GetAuthorList()
      For Each iteration_row As DataRow In rs.Tables(0).Rows
```

```
            List1.Items.Add(iteration_row.Item("Author"))
      Next iteration_row
End Sub
```

Appendix 10: Variable initialization

This example demonstrates how the VBUC is able to refactor the variables declarations with their initialization values.

Original VB6 Code

```
Private Sub Command2_Click()
     Dim year1 As Integer
    Dim year2 As Integer
    year1 = Format(SSDateCombo1.Date, "yyyy")
    year2 = Format(SSDateCombo2.Date, "yyyy")
    GetTitlesByPublishedYear year1, year2
End Sub
```

Code Generated by the UW

```
Private Sub Command2_Click(ByVal eventSender As System.Object, ByVal
eventArgs As System.EventArgs) Handles Command2.Click
      Dim year1 As Short
      Dim year2 As Short
      year1 = CShort(VB6.Format(SSDateCombo1.Date, "yyyy"))
      year2 = CShort(VB6.Format(SSDateCombo2.Date, "yyyy"))
      GetTitlesByPublishedYear(year1, year2)
End Sub
```

VB.NET Code Generated by the VBUC

```
Private Sub Command2_Click(ByVal eventSender As Object, ByVal eventArgs
As EventArgs) Handles Command2.Click
      Dim year1 As Integer =
      CInt(CDate(SSDateCombo1.Value.Date).ToString("yyyy"))
      Dim year2 As Integer =
      CInt(CDate(SSDateCombo2.Value.Date).ToString("yyyy"))
            GetTitlesByPublishedYear(year1, year2)
End Sub
```

Appendix 11: User declarations advanced refactoring

.NET naming conventions:

The Visual Basic Upgrade Companion is able to apply code refactoring and transformation to generate naming conventions-compliant VB.NET and C# source code.

The naming conventions standards are described here:

Identifier	Case	Example
Class	Pascal	class myclass -> class Myclass or class MYCLASS -> class MYCLASS
Enum type	Pascal	errorLevel -> ErrorLevel
Event	Pascal	userevent -> Userevent * note that only user events are changed; events implemented are not in this scope.
Interface	Pascal	IDisposable Note Always begins with the prefix I.
Private Method	DownCase	MyMethod ->mymethod
Public Method	Pascal	mymethod -> Mymethod
Class Fields		Field -> _field
Parameter	Downcase	TypeName -> typename
Property	Pascal	myproperty -> Myproperty
Local variables	Downcase first letter	TEST -> test