FAK EKI

Margaret Mburu and Robert Josefson

Conversion of a Visual Basic 6 Application to Visual Basic .NET

Subject

C-level thesis

Date/Term:	08-06-03
Supervisor:	Donald F. Ross
Examiner:	Martin Blom
Serial Number:	C2008:06

Conversion of a Visual Basic 6 Application to Visual Basic .NET

Margaret Mburu and Robert Josefson

This report is submitted in partial fulfillment of the requirements for the Bachelor’s degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Approved 080603

Advisor: Donald F. Ross

Examiner: Martin Blom

Abstract

This dissertation describes the process of converting an application from Visual Basic to a .NET programming language. This work was carried out on behalf of The Prevas Company [0] based in Karlstad, Sweden. Prevas provides IT solutions and industrial systems for several world leading companies such as Ericsson, Nokia, Siemens and many others. The objective behind the conversion of this application was to facilitate a more compatible and flexible option suitable for the other products using the .NET environment. In addition, Prevas felt that the Visual Basic 6 environment had grown rather old and outdated. The task was therefore to convert the source code from Visual basic 6 to .NET language and to determine the most efficient method for the conversion.

The outcome of this project was to achieve a fully converted application using the .NET framework with its original functionality intact.

Contents

List of Figures

List of tables

1 Introduction

Upgrading an application from its current environment into a new or different environment involves converting the source code into another language. Basically, the new environment may provide new, different, improved or enhanced features which add value to the upgraded or converted application. Microsoft's discontinuation of support for Visual Basic 6 and introduction of Visual Basic.NET with new and different features resulted in most Visual Basic developers and organizations opting to either retain or convert their existing VB6 applications to VB.NET. However, the choice of which option to take depends on various factors as seen later in this report. Considering that most companies have invested time and money in development of VB6 applications there is no doubt that there has been some hesitation in adapting these applications to new .NET environment. However, in order to meet the demands of software development requirements and to take advantage of the benefits offered by the new features most companies and developers in the recent past have opted to convert their existing applications to .NET framework. An example of those companies that have opted to convert their applications to a .NET framework is Prevas.

Prevas is a company developing intelligence in products and industrial systems for several international companies such as Ericsson, Siemens, Nokia and many others. Prevas maintains an extensive library of technology platforms within technologies such as embedded processing, wired/wireless communication, signal processing and graphical user interfaces. Since it was founded over 20 years ago, Prevas has successfully delivered over 2000 different products. Prevas provides clients with services customized to meet their demands ranging from in-sourcing to complete outsourcing of the client's projects [0].

Our task entailed the conversion of a VB6 application called Reptile to the VB.NET environment and documenting the conversion process. Reptile is an application used for troubleshooting and repair actions for electronic components, with connection to two databases called QSP and Testnet. Fault reports are entered into the QSP database and test information is collected from both the QSP and Testnet databases. Reptile can also offer the possibility to view and print protocols, and to view an individual's history, i.e. previously performed tests and repairs.

We were assigned this task by Prevas because they felt that VB6 was getting rather outdated and since most of their newer products are using the .NET framework they wanted to upgrade their VB6 applications to .NET framework that could allow easier integration. The .NET framework provides a more compatible and flexible platform suitable for the other products using the same environment. The goal of this project was to achieve a fully converted application using the .NET framework with its original functionality intact.

This dissertation consists of a detailed description of issues that were taken into consideration before the upgrade, an upgrade report describing the problems we encountered during the conversion to VB.NET and the solutions to these problems. Finally the document presents our conclusions, lessons learned and recommendations on how to perform a conversion.

1.1 Document Outline

This report is structured as follows:

Chapter 2 contains the background information about the languages involved in the conversion process. It also points out differences between them. Chapter 2 also describes the .NET framework and the components that are a part of the NET framework.
Chapter 3 describes the conversion process and the result of this project. It also describes the issues we encountered during the conversion and how they were solved. Chapter 3 also describes other conversion issues that might occur during a conversion.
Chapter 4 describes the result of the conversion and introduces a checklist which may be used to perform similar conversion in the future. This chapter also describes each step in the checklist and the corresponding section in the conversion which was performed.
Chapter 5 contains recommendations for future work and conclusions drawn from this conversion process.

2 Background

2.1 Visual Basic 6.0

Visual Basic (VB) [8] is basically an easy to use and learn language that allows creation of simple/complex Graphical User Interface (GUI) applications [3]. VB introduced Windows Rapid Application Development (RAD) [13], a feature which allows rapid creation of applications such as demonstrations and quick user interface examples.

Initially, VB was never intended to be a complete development language but it was targeted for Microsoft Windows client applications. These client applications run on a workstation or personal computer and depend on server to perform some operations. VB includes all the necessary tools for writing programs for Windows. This was reported to be the reason why VB was so successful over the years in creating third party industrial components such as ActiveX controls [10]. ActiveX controls, also called Active Components, is a reusable software component based on the Component Object Model (COM) which is described more in detail below.

Visual Basic 6 was released in 1998 after a series of previous Visual Basic 1 to 5. VB6 was intended for developing Windows based applications. Indeed, VB was reported to be one of the first systems that made writing programs for Windows Operating System easier [3].

2.2 Visual Basic 6.0 Features

VB6 uses the Component Object Model (COM) [11] product, a mechanism for linking software components together. The COM interface is a platform and is a language independent technique used for communication between objects within a program or between different programs. The COM is also referred to as a binary standard and works with the codes that are interpreted at runtime. The COM enables creation of high performance components and applications such as ActiveX controls and objects. The ActiveX technology allows the building of ActiveX components that facilitate access to reusable, and reliable software components created by other programmers to perform common tasks. However, the

COM's biggest disadvantage is that all the COM components have to be registered in the Windows Registry.

The Integrated Development Environment (IDE) provided by VB6 allows programmers to create, run and debug Visual Basic Programs very easily. An IDE is a computer program containing features to make development easier such as a compiler, debugger and a text editor. An example of an IDE is the Microsoft Visual Studio [8].
Rapid Application Development (RAD) is an important feature in VB6. Just like the name suggests, RAD allows rapid creation of applications. With the use of RAD feature, VB6 may create a good demonstration and provide quick user interface examples [13].

Despite VB6's popularity, Microsoft saw the need to replace it with a more efficient and flexible VB.NET language [5]. VB6 is not a true Object Oriented language and could not support useful features such as inheritance, encapsulation, polymorphism, overloading and overriding and this was reported to be the reason why some programmers were moving from VB6 to Java. Additionally COM was reported to be rather complex and tightly coupled binary standard which did not support internet applications effectively. There was need for a language which could address the challenges of Web applications development more effectively. The few limitations mentioned, plus others to be discussed later led to the withdrawal of VB6 development in favor of a Visual Basic.NET.

2.3 .NET Framework

The Microsoft .NET framework is a software component that is a part of the Windows platform. .NET covers all the layers of software development from the operating system up. The .NET framework consists of two main elements: a runtime environment called Common Language Runtime (CLR) and a class library called the Framework Class Library (FCL) [4].

One of the most important advantages with the .NET framework is that it is fully language independent which means components written in .NET languages can interact with each other regardless of which .NET language they are written in. The language independence is made possible due to the Common Language Runtime (CLR) [2] which will be described in detail in the next section.

2.3.1 Common Language Runtime

The Common Language Runtime [2] is the virtual machine component of the .NET framework, as shown in **Error! Reference source not found.**. A virtual machine [18] is a software implementation of a machine. There are several advantages with virtual machines such as allowing several different operating systems on one computer working isolated from each other. The CLR is located at the core of the .NET platform and is responsible for managing the execution of the compiled source code.

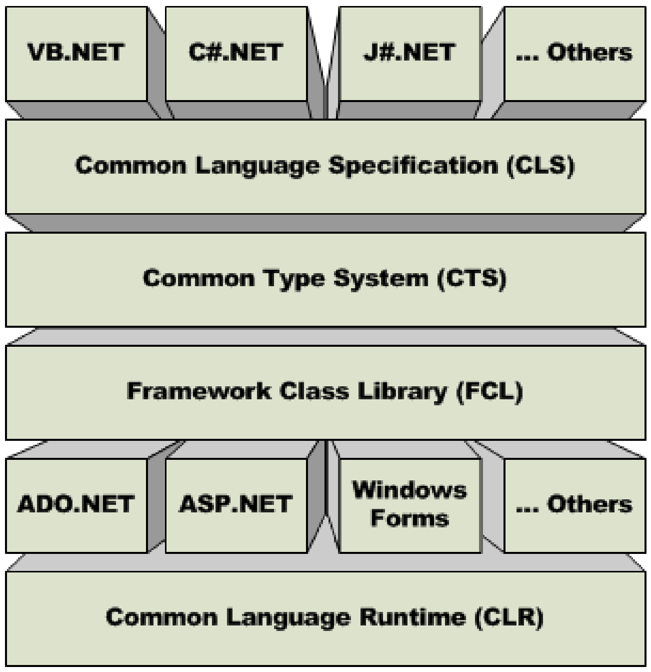

VB.NET
C#.NET
J#.NET
... Others
Common Language Specification (CLS)
Common Type System (CTS)
Framework Class Library (FCL)
ADO.NET
ASP.NET
Windows Forms
... Others
Common Language Runtime (CLR)

Figure 2-1 The Common Language Runtime Environment[2]

Among the services that the CLR provides are language independence, versioning and deployment support.

2.3.2 Compilation

Compilers that target the .NET framework differ from traditional compilers. Traditional compilers target a specific processor and produce binary files that may then be executed directly on the target processor. However, .NET compiler produces binary files containing an intermediate language known as the Common Intermediate Language (CIL formerly known as Microsoft Intermediate Language) [2].

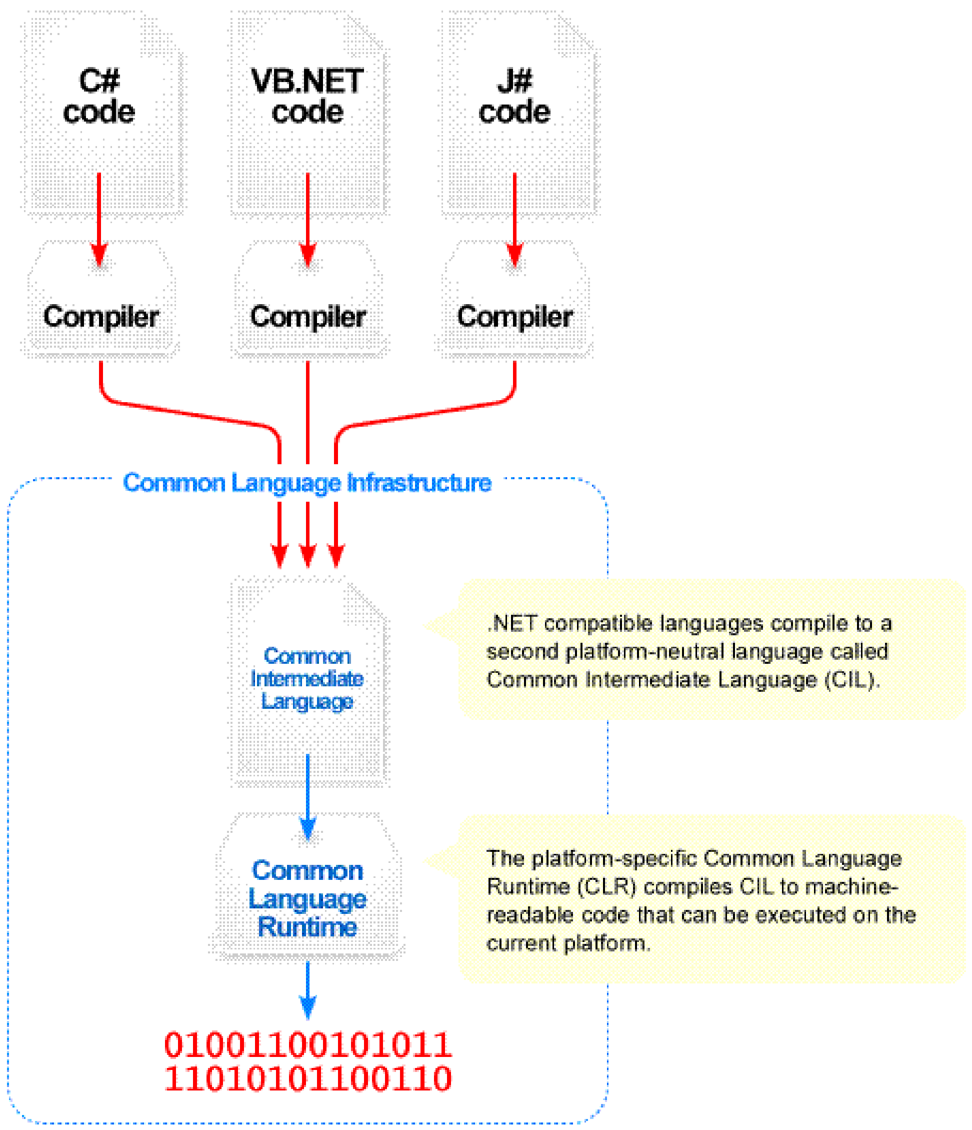

C# code
VB.NET code
J# code
Compiler
Compiler
Compiler
Common Language Infrastructure
Common Intermediate Language
.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).
Common Language Runtime
The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.
01001100101011
11010101100110

Figure 2-2 The Common Language Infrastructure[2]

This CIL is not interpreted but is rather compiled in a manner referred to as Just in Time Compilation (JIT) [19]. The CLR contains a couple of JIT compilers which convert the CIL code into native code which is binary code targeted at the processor on the computer on which the code resides.

2.3.3 .NET Framework - Class Library

The Framework Class Library (FCL) is a standard library which consists of 2500 reusable classes available to all .NET languages [2] .The library is built on the CLR and provides services needed by modern applications. Applications targeting the .NET framework interact directly with the FCL while the CLR serves as the underlying engine.

2.3.4 Other .NET Related Languages

There are several languages which are compatible with the .NET framework such as J sharp (J#) [15] and the .NET version of Delphi called Delphi.NET [14] however the most common languages are C# (C sharp) [4] and Visual Basic.NET (VB.NET) [5]. A key piece of functionality that makes it possible to use several different .NET languages in the same application without any conversion between them is the Common Type System (CTS) [4]. The CTS defines all possible data types and programming constructs and how they may or may not interact with each other. So since all languages using the CTS are using the same library of data types calling one language from another does not require any conversion between them.

2.3.5 C# (SHARP)

C# is a programming language designed to specifically target the .NET framework. It was designed by a Danish engineer called Anders Hejlsberg who also created Turbo Pascal and Delphi. C++ is the language most similar to C# however one great advantage with C# is the ability to create graphical user interfaces relatively easy. With C# and other .NET languages which uses Win Forms the developer can design a user interface simply by dragging and dropping controls onto a form from a toolbox and then write the code to handle the control events whereas in C++ designing a graphical user interface can be a rather complex task.

2.4 Differences between Visual Basic 6 and Visual Basic.NET

Understanding the differences between Visual Basic 6 and Visual Basic .NET is a fundamental step towards successful conversion process. There are quite a number of differences between VB6 and VB.NET, this report will discuss and highlight some of them.

VB.NET could have been an upgrade of VB6 with probably a few changes or improvement of a few features like many upgrades but this was not the case. The move to Visual Basic.NET introduced new, different additional features and significant improvements in comparison with VB6. VB.NET is not entirely backward compatible with VB6. The Integration Development Environment (IDE) in VB.NET was designed to accommodate all Visual Studio languages such as C#, Visual Basics and Visual C++ [5].

2.4.1 Multithreading

VB.NET supports multithreading [5] a very useful technique that gives a program ability to execute several threads simultaneously. With the use of Common Language Runtime (CLR) VB.NET is capable of creating free threaded applications a feature that allows multiple threads to access similar set of shared data. VB6 has no direct support for free threading due to restrictions in the runtime. Code written in VB6 is synchronous in that each line of code must be executed before the next one is processed.

2.4.2 Object Oriented Feature

VB.NET is a true Object Oriented Language [5] in contrast to VB6 whose Object Orientation is based on Common Object Model (COM). Being a true Object Oriented language VB.NET supports features such as inheritance, polymorphism, overloading, overriding and encapsulation such that classes and objects are created and classes are easily derived from other classes. Again full Object Orientation in VB.NET facilitates efficient and easier development of Windows applications.

2.4.3 Console Applications

Unlike VB6, VB.NET supports Console Applications [16]. Console Applications are command line/text based applications which include some Graphical User Interface (GUI) operating systems that allow characters to be written to and read from the console and executed in the DOS version. Console Applications are created and supported by the System.Console namespace where namespace is a collection of different classes. A good example of console applications is Win32 Console in Microsoft Windows applications such as one illustrated in Figure 2-3.

```
Module Module 1
Sub Main ()
System.Console.Write ("Welcome to my Console Application")
End Sub
End Module
```

Figure 2-3 Example Win32 Application in VB.NET

2.4.4 Memory Management

There is a significant difference in memory management between VB6 and VB.NET. In VB6, memory associated with variables is deterministic in that, you know when the memory is free and allocated. VB.NET uses a form of memory management called garbage collection. Garbage collection is a mechanism for releasing the memory from objects and components that are no longer in use. VB6 uses the COM memory management mechanism of reference counting which stores the number of references to an object [7]. The reference counting mechanism counts the number of active references to the object and when the object's reference count drops to zero the object is de-allocated. Reference counting makes it possible to determine where and when an object will be de-allocated and also to manipulate the lifetime of an object in a deterministic way. VB.NET does not use reference counting and therefore it is no longer possible to determine the exact lifetime of an object. However, the Common Language Runtime (CLR) in VB.NET takes care of garbage collection by releasing resources as soon as an object is no longer in use. This relieves the developer from thinking of ways to manage memory.

2.4.5 Runtime Environment

Programs in .NET framework execute in the Common Language Runtime (CLR) environment. VB6 and the previous VB Programs execute their own runtime library known as MSVBVM60.DLL. The CLR uses better code translation via the Just in Time compiler while VB6 Runtime interprets the code [7].

2.4.6 Data Types and Properties

There is a Common Type System (CTS) for all the languages under the .NET framework and these languages must support the same data type as required by the common language runtime (CLR). This eliminates incompatibilities between all the .NET languages. There are differences related to data type in VB.NET as compared to VB6. For example, VB.NET does not support *variant* data type. *Variant* data type in VB6 was the default universal data type with *Empty* as default value. This *variant* data type has now been replaced by *Object* data type in VB.NET with *Nothing* as default value [5]. An example is shown in Dim var1 *as Variant // in VB6 can hold any data type*

Changes to:

Dim var1 as Object // in VB.NET can hold any data type

.

Dim var1 as Variant // in VB6 can hold any data type

Changes to:

Dim var1 as Object // in VB.NET can hold any data type

Figure 2-4 Example of Variant Data Type

Other data type changes are illustrated by Table 2-1 below:

Data Type	Visual Basic 6	Visual Basic.NET
Integer	16 bit size	32 bits size
Long	32 bit size (Integer)	64 bit size
Date	Date used to store as Double (64 bit Double)	Introduces Date Time data type to dates in different formats
Currency	Currency	Replaced with Decimals – more accurate for rounding numbers

Table 2-1 Differences between VB6 and VB.NET

2.4.7 Declaration and Initialization of Variables

There are differences in declaration and initialization of variables. In VB6 several variables can be declared in the same statement but the data type of each variable must be specified as shown in Dim var1 *as Variant* *// in VB6 can hold any data type*

Changes to:

Dim var1 as Object *// in VB.NET can hold any data type*

5 below:

Examples

Dim X, Y As Integer	*// where X is declared as variant and Y is an integer (A variant can hold any data type)*
Dim K As Integer M As Integer	*// where K is declared as an integer and M as an integer.*
Dim J As Integer Z As Double	*// where J is declared as an integer and Z as Double*

Figure 2-5 Variable declaration example

In VB.NET, all variables must be declared as being of a specific type and multiple declarations of variables of the same data type can be declared on the same line without repeating the type keyword. For instance, a statement as *Dim x, y As Double* specifies that both variables have data type *Double*. In VB6, only y would have type *Double*.

Initialization and declaration of a variable in VB.NET can be done on the same line such as the example shown below:

```
Dim name As String =”Moses”

System.Console.Write (name)
```

Figure 2-6 Initialization and declaration in VB.NET

2.4.8 Structures and User – Defined Types

Unlike VB.NET, VB6 uses *Type, End Type* to create structures and user-defined types as shown on the figure below:

```
Type StdRec
StdId As Integer
StdName As String
End Type
```

Figure 2-7 VB6 Structure

In VB.NET, structures are like classes which can have methods and properties as well. The Common Language Runtime (CLR) uses the name *Type* in a broad sense to include all data types and the statement *Type* is changed to *Structure.* This new syntax structure uses Structure… End Structure which is similar to C++, where the access scope of every member can be specified either as Public, Private, Friend or Protected as in figure below:

```
Structure StdRec
Public StdId As Integer
Public StdName As String
Private StdInternal As String
End Structure
```

Figure 2-8 VB.NET Structure

2.4.9 Arrays

In VB.NET arrays are zero based which means that zero is the default lower bound and the upper bound specifies the index of the last element in the array. VB6 had an *Option Base* to specify the lower bounds of all arrays to be either 0-based or 1-based. This *Option Base* is not supported by VB.NET.

An array in VB.NET declared as:

Dim Items (10) As Integer – gives 10 items from index 0 through 9

Figure 2-9 VB.NET Array

The same declaration would give 11 items from index 0 to index 10 in VB6. This needs to be checked carefully because one may end up with an extra element than may not have been intended while declaring the array. Arrays in VB6 are either fixed length or dynamic. There are no fixed arrays in VB.NET; all arrays are dynamic meaning that an array can always be resized regardless of how it is declared. However, *ReDim* a keyword used to size or resize a dynamic array, cannot be used in the initial declaration of an array but can only be used to change the size of an array that has been declared.

2.4.10 Fixed Length Strings

In VB6, a string can have a fixed length by specifying the length in the declaration such as *Dim Name as String *20*. Fixed length strings are not supported by VB.NET and are implemented as objects of the string class, which is part of the .NET system's namespace.

2.4.11 Structured Exception Handling

VB6 uses Unstructured Exception Handling, *On Error Goto* and *On Error Resume Next* statements to handle exceptions at runtime. Microsoft introduced Structured Error Handling in VB.NET, an improved error handling which uses *Try...Catch...Finally* control. With VB.NET exception handling, exceptions are returned as objects which can be examined to determine the actual nature of the error. Moreover, each exception has its own string Message property which can be used to display information concerning the error [5].

2.4.12 Compatibility

VB.NET supports language interoperability with other languages which uses a .NET framework. In this case then, code written in any other .NET compliant language can be used and enhanced easily and similarly the code written in VB.NET can be used and enhanced by other .NET compliant languages. VB6 also provided this functionality through COM but it was rather limited. However, to make language interoperability easier VB.NET uses Intermediate Language (IL) and Common Language Specification (CLS) of the .NET framework.

VB.NET is platform independent and as such programs written in VB.NET can run on any platform under .Net framework.

2.4.13 Security

VB.NET provides a new security model, Code Access security, as compared to role-based security in VB6. Code Access security controls what the code may access. For example security can be set to a component in such away that a component cannot access the database. This type of security is crucial because it facilitates building components that can be trusted.

2.4.14 Forms

VB6 forms were replaced by Windows Forms in VB.NET which are based on System.Windows.Forms namespace. Windows forms allow automatic resizing and have many new and improved features such as *in- place* menu editing and better *Graphical Display Interface (GDI+)* support which provides better graphics handling.

3 Conversion Process

Converting an application from one language to another can be a rather complex task. Differences between languages can result in that substantial changes have to be made in the code during the conversion process. The list of differences that are set to cause issues during conversion can be long. One area to think about is flow of control; two examples of this area are the *goto* and *gosub* statements that are supported in VB6 but not in VB.NET. Another area to take into consideration is array index issues, a good example of an array index issue is when an array is declared in VB6 with size 6 will mean it has room for 7 elements with index ranging from 0 to 6. An array declared in VB.NET with size 6 means it has room for 6 elements with index ranging from 0-5 this is something that can cause problems while upgrading from VB6 to VB.NET.

There are several aspects which have to be taken into consideration before upgrading an application. The first decision which has to be made is whether it is really necessary to upgrade the application to .NET at all. However, by not upgrading to .NET would mean that an application may not take the advantage of the features provided by the .NET framework, which were mentioned in section 2.3.

Another way to get the job done is to partially upgrade the application. Furthermore, there are several different techniques to upgrade the application. There are automated Upgrade Wizards that perform the upgrade but these do not perform the upgrade 100%. A lot of work has to be done with the code in order to get the application up and running. The other technique is to simply rewrite the program into a .NET language but this technique also requires a lot of work and time. While using automated wizards to perform the conversion there are few large VB6 applications which will be converted without significant rewriting of the code after the conversion. Features such as the App class and declaring a variable of the *as any* which was ok in VB6 are no longer supported in VB.NET and the Upgrade Wizard will not upgrade them. Such problems would have to be re implemented which in turn may affect other parts of the program. Facing such problems simply re-writing the application can in some cases go faster than using an Upgrade Wizard. However which technique to use is determined on a case by case basis.

3.1 Language Decision:

Since we are converting a VB6 application to .NET we had two languages to choose from: C# and VB.NET. Since both of them are .NET languages they use the same Framework Class Libraries. VB.NET however has more in common in syntax with VB6 than C#. Another advantage with converting the code to VB.NET is that we then can use the Microsoft Upgrade Wizard to perform the conversion. Those are the main reasons for us choosing to convert to VB.NET over C#.

3.2 Before the Conversion

Before upgrading any systems, existing or current systems need to be evaluated before upgrading in order to identify the changes that should be made before the upgrade. To prepare the application for upgrade, an initial assessment is needed in order to access the current and target architecture. Since our task is to upgrade an application from Visual Basic 6 platform to a VB.NET environment, the issue at hand is which recommendations or practices we should take into account before the upgrade process. Given the fact that any upgrade would lead to some modifications, there is need to understand the functionality of the application and establish the effort required, as this could help to eliminate the number of changes that may be required and thus saves work and reduces the amount of time required by the upgrade.

Given the considerable differences between VB6 and VB.NET, the fundamental step is to understand and identify the areas affected by the differences in order to modify and minimize the number of changes that may conflict with the new environment. Syntax differences should be understood and identified. Some VB6 language elements such as declaring a variable with "*as any*" as its data type are not supported by VB.NET and such declarations therefore, must be identified and rewritten. Other elements may no longer be meaningful to VB.NET such as *Go Sub, Defnt, Return* and *VarPtr*. For example, pointer elements like *VarPtr* are no longer valid in the .NET framework.

One option is to use the *Upgrade Wizard*- an inbuilt feature provided by Microsoft Visual Studio [22]. The Upgrade Wizard fixes some of the problems involved in the upgrade process reducing a great deal of upgrade time. For instance problems such as variables and variants can be fixed with little or no effort. Some problems are more easily solved before the upgrade rather than waiting to fix them after the upgrade (in VB.NET) however, the decision on which option is better depends on the option taken.

3.3 Results

The Microsoft Visual Studio Upgrade Wizard upgraded the project and generated an upgrade report describing the results in detail. The upgrade report gave a breakdown of 1049 error messages out of which 335 were referred to as compile errors. Compile errors mean that the program will not compile and we would not be able to even try to run the program until those 335 compile errors has been solved. Further the upgrade generated some 559 so called design errors and 155 upgrade warnings. The design errors are concerning properties that were set during the design time of the application which could not be upgraded because there is no equivalent property in VB.NET. It should also be noted that design errors can also generate compile errors. The difference between a compile error and a design error is that the design error refers to a property set during design time. The upgrade warnings relate to differences that may cause unexpected results while the application is running. The most common upgrade warning in our project is the *“Cannot determine the default property of object”* referring to the fact that default properties of objects are no longer supported in VB.NET. This problem will be described in detail in the next section. Our strategy is to first of all address the compile errors in order to be able to compile the program. Once we are able to compile the program then we can start to work with the upgrade warnings.

3.4 Compile Errors

The compile errors are the most severe type of errors we faced after the application was upgraded. Until the compile errors are fixed we are unable to run the program and perform further tests of the functionality of the application. So after the upgrade we started to look for solutions and fix the compile errors

3.4.1 Property was not Upgraded (Compile error).

There are several features in VB6 which have no direct equivalent in VB.NET. There is no general solution to these problems. To some of them there are features in VB.NET which are

similar that can be used and in other cases there is nothing in VB.NET that can be used instead the entire feature has to be implemented manually.
The App class in VB6 is not present in the VB.NET however there are some features that are similar in the VB.NET application class [30]. In our project we encountered two properties in the App class for which there was no equivalent in VB.NET. The first problem we encountered was regarding the helpconstant *App.helpfile* where the Upgrade Wizard generated the message *App property App.helpFile was not upgraded* [7]. Since that specific line of code was not converted it was refering to a class which is not present in the VB.NET class library and therfore generating a compile error. The *app.helpfile* specifies the path and name to the helpfile used by the application. In our project the app.helpfile was used as a parameter in the winhelp function Figure 3-1.

```
Declare Function WinHelp Lib "user32"  Alias "WinHelpA"
ByVal hWnd As Integer,_
ByVal lpHelpFile As String,_
ByVal wCommand As Integer, _
ByVal dwData As Integer) As Integer
```

Figure 3-1 WinHelp function

The WinHelp function is used to create pop-up context sensitive help by using the windows API. There are several different ways to solve this problem. If the application has a helpfile associated to it then the solution is just to change the *app.helpfile* to the name and path of the helpfile. So for example if the helpfile is called name MyProject.hlp and is stored in the c:\ directory the *app.helpfile* should simply be changed into “c:\MyProject.hlp”. If the application does not use a helpfile the *app.helpfile* property could be changed into an empty string “ ”. Since our application uses a helpfile we solved our problem by simply changing all appearances of the *app.helpfile* into the name and path of our helpfile.

The other case where the App class was being used was with the *App.PrevInstance*. The PrevInstance is used to determine whether there is another instance of the application already running. This is an important feature since several instances of the application running at the same time can have serious consequences, as for example when two instances of the same program manipulate the same data in the database which may lead to the data actually stored in the database being inconsistent. Since there is nothing in VB.NET similar to the *App.PrevInstance* we had to create a function where we implemented the functionality of the *App.PrevInstance*.

```
Function PrevInstance() As Boolean
If Ubound(Diagnostics.Process.GetProcessesByName
(Diagnostics.Process.GetCurrentProcess.ProcessName)) > 0 ThenReturn True
Else Return False
End If
End Function
```

Figure 3-2 App.PrevInstance replacement

We created a Boolean function since we only needed an answer to whether there are other instances of the application running or not. To determine whether there are more instances of the application running we uses the Process class [26] which is a member of the *System.Diagnostics* namespace. An object of the process class provides access to the processes running on a system. To retrieve the process name of our application we used the *GetCurrentProcess.ProcessName* properties of the Process class and the answer from that statement was in turn used as an argument in the *GetProcessesByName* property of the Process class. The *GetProcessesByName* property creates an array containing all the recourses associated with the process or file which is used as argument to the function. In our case we pass the process name of our application. The *Ubound* function is used in the if statement to determine if the upper bound of the array created by the *GetProcessesByName* property is greater than 0 indicating that there are more instances of the application running.

3.4.2 Declaring a parameter 'As Any' is not supported

After the upgrade was completed the first compile error we encountered referred to the *as any* statement. In VB6 when referring to an external procedure using the declare statement it was possible to declare *as any* for the data type of the parameters and return type. Using the *as any* as data type disables type checking making it possible for any data type to be passed or returned. However, VB.NET does not support the use of the *as any* as data type. In a declare statement the data type of the parameters and return type has to be specifically declared. A great advantage with having to specifically declare the data type is type safety, meaning reducing the risk of undesirable and unpredictable behavior of the program.

In our project we had several references to external procedures stored in Dynamic Linked Libraries (DLL) files. Dynamic linking means that the subroutine is loaded into the application at runtime instead of during compile time. The problem with these references is that each one of them had one or several parameters the data type of which was declared *as any*. There are several ways to solve this problem. One way is to simply trace the call and from there, find out with which data type the external procedure is being called. If it turns out that the external procedure is being called with several different data types the overloading technique is a good approach. Overloading means that several functions with the same name are declared but with different data types on their parameters and return type then the function that is actually being used depends on with which data type the function is being called [5].

An example of a function which is used in our project and has a parameter with “as any” as its data type is the *GetPrivateProfileString* function. The *GetPrivateProfileString* function is a part of the Windows platform and is supported by all versions of the Windows operating system from the Windows95 edition and all later versions. The declaration statement of the function is shown in **Error! Reference source not found.**:

```
Private Declare Function GetPrivateProfileString _
Lib "kernel32" Alias "GetPrivateProfileStringA" _
(ByVal lpSectionName As String, _
ByVal lpKeyName As Any, _
ByVal lpDefault As String, _
ByVal lpReturnedString As String, _
ByVal nSize As Long, _
ByVal lpFileName As String) As Long
```

Figure 3-3 The GetPrivateProfileString Function Declaration

From the declaration above one can see the *lpKeyName* parameter has *as any* as its data type. In our project we solved this by tracing all the references to this function and checked the data type which was being passed into the *lpKeyName* parameter in every call to the

GetPrivateProfileString function. Visual Studio has built in features which facilitate tracing of calls to a specific function. Simply by clicking the right mouse button on the function name and choosing the “find all references” Visual Studio presents a list with all calls and declarations of the function concerned. By using this method we ensured that all the references to the function regarding the *lpKeyName* parameter were of the string data type. We had the same problem with several references to external procedures stored in DLL files concerning the ODBC interface. An example of a function which we had to modify in order to be able to compile is the *SQLColAttributes* which obtains the attributes for a column in the result from a query sent to the database. *SQLColAttributes* can also be used to determine the number of colums attained in the result from the query as in Figure 3-4.

```
Declare Function SQLColAttributes Lib "odbc32.dll"
(ByVal hstmt As Integer,_
 ByVal icol As Short,_
 ByVal fDescType As Short,_
 ByRef rgbDesc As Any,_
 ByVal cbDescMax As Short,_
 ByRef pcbDesc As Short,_
 ByRef pfDesc As Integer) As Short
```

Figure 3-4 Example function with SQL Attributes

From the declaration above it can be seen that the *rgbDesc* parameter has the *as any* declared as the data type. The problem with this parameter was solved by checking for other references where the function is being used. We found that the most probable data type of this is the string data type. The general sollution for this problem is to trace the call to the function having a parameter using *as any* as datatype and from there determine which datatype which is used in the call to the parameter concerned. If there are several calls to the function using different datatype one sollution would be to use overloading. Overloading is a techniqe where several copies of a function is decleared but with different parameter signatures. When the function is called the call would be bound to the function whith parameter signature matching the parameter signature of the call.

3.4.3 Unable to determine which constant to upgrade vbNormal to

In VB6 it is possible to define a set of named constants known as enumeration constants [7]. Every member of the enumeration is assigned an integer value and in the code the constant is evaluated to its assigned integer. Since enumerations in VB6 represent integers and not distinct types the program can only verify their underlying value. Unfortunately this allows developers to use constants intended for different purposes interchangeably and also to use simple integers which represent the underlying values instead of the constants. In our project this problem was exclusively related to the *vbNormal* constant [7]. The constant *vbNormal* was being used in connection with the *Screen.MousePointer* property. The *vbNormal* constant is often used instead of the *vbDefault* constant. Both the *vbDefault* and the *vbNormal* constants have 0 as their underlying value. **Error! Reference source not found.** shows code examples showing 3 statements that all have the same underlying value:

Correct MousePointer constant

Screen.MousePointer = vbDefault

Incorrect constant, same underlying value

Screen.MousePointer = vbNormal

The same underlying value

Screen.MousePointer = 0

Figure 3-5 Example of Code with the vbNormal Constant

Since the upgrade wizard is unable to upgrade the vbNormal constant this generates a compile error after the conversion. The solution to this problem is to determine which class in VB.NET has a similar behavior. In most cases we changed the vbnormal constant to the *cursors.default* statement which gets the default cursor, usually an arrow cursor [20]. In other cases in our code where the application is busy and the user should wait for a specific operation to complete we used the *cursors.waitcursor* statement which changes the cursor into an hourglass [21].

3.4.4 Number of Indices Exceeds the Number of Dimensions of the Indexed Array

The declaration of arrays is among the issues raised by the Upgrade Wizard which must be fixed before the program will run. Arrays in VB.NET can be declared and initialized in one statement, or can be declared in one statement and initialized in another. However, when an array is declared and initialized in the same statement then, the type and the number of elements in the array must be specified. In VB6 both the upper and lower bound of an array are defined. All arrays in VB.NET have a lower bound of zero by default.

The main issue with arrays during the conversion process was multidimensional arrays [22]. The dimensions of an array correspond to the number of indexes used to identify an individual element. 'Multidimensional arrays should be handled with care because when dimensions are added to an array the total storage may increase considerably depending on the size of the array.

Our VB.NET application generated the following compile error: *Number of indices exceeds the number of dimensions of the indexed array.* This means that the Upgrade Wizard was unable to determine how many dimensions the array has and therefore the array is treated as a one dimensional array. This problem is solved by using commas in the declaration of the array in order to specify how many dimensions it has as shown is shown in Figure 3.6.

When the Reptile application performs queries on the database the result from the queries is stored in a matrix called *query_result_matrix*. The problem with the *query_result_matrix* was that it was declared as an array without any dimensions. In VB6 this worked fine since the dimensions would be determined once the array was assigned values. However in VB.Net this feature is no longer supported and dimensions has to be specified in the declaration of the array. It is possible to change the size of the array using the ReDim operator but the number of dimensions cannot be changed.
The solution to this problem was that the declaration has to be made multidimensional so we had to change the declaration from:

Public *query_result_matrix* () As String

To

Public *query_result_matrix* (,) As String

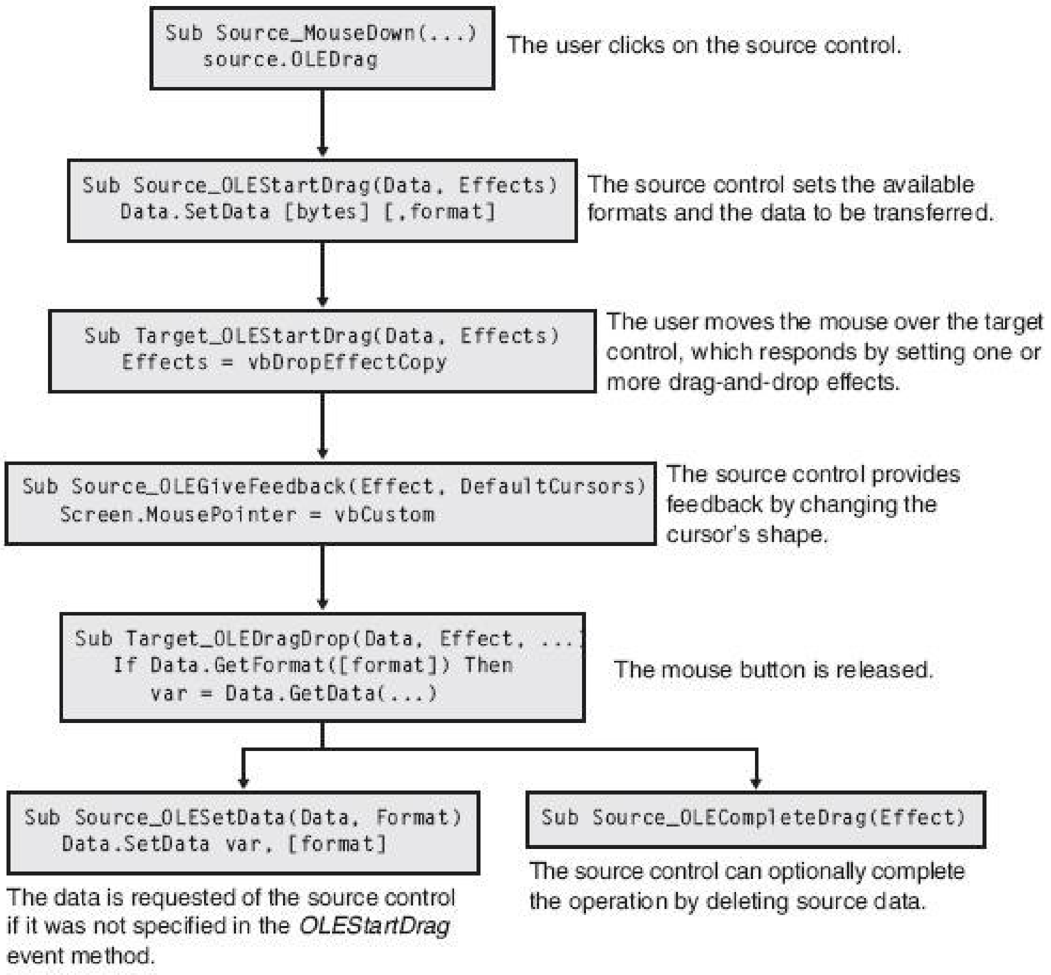
Figure 3-6 Declaration of a multidimensional array

This is because in VB.NET a multidimensional array is declared by adding one or more commas within the parentheses of the array declaration. Each comma added specifies an additional array dimension.

3.4.5 Other Possible Compile Errors

Apart from the compile errors we encountered, mentioned in the section above, there are several other upgrade issues which may occur depending on which components the project uses. VB6 is backward compatible towards the older VB versions such as VB5, if it is an old project it might originally have been written in for example VB5 and then ported up to VB6. This means that there might be some objects from VB5 or earlier VB versions in the code. When upgrading to VB.NET the wizard may not be able to recognize these objects and therefore the wizard will be unable to upgrade these objects, resulting in compile errors in the code.

Another area which the Upgrade Wizard is unable to upgrade is the drag and drop features. Drag and drop is the action of clicking on a virtual object and dragging it to a different location. Drag and drop is supported in both VB6 and VB.NET however it is implemented differently. VB6 supports two kinds of drag and drop operations. One is the standard drag which is intended to support drag and drop operations within a single windows form. The other is the OLE drag and drop operations which are used to support drag and drop capability between applications [7].


The user clicks on the source control.
The source control sets the available formats and the data to be transferred.
The user moves the mouse over the target control, which responds by setting one or more drag-and-drop effects.
The source control provides feedback by changing the cursor's shape.
The mouse button is released.
The data is requested of the source control if it was not specified in the OLEStartDrag event method.
The source control can optionally complete the operation by deleting source data.

```
Sub Source_MouseDown(...)
   source.OLEDrag

Sub Source_OLEStartDrag(Data, Effects)
   Data.SetData [bytes] [,format]

Sub Target_OLEStartDrag(Data, Effects)
   Effects = vbDropEffectCopy

Sub Source_OLEGiveFeedback(Effect, DefaultCursors)
   Screen.MousePointer = vbCustom

Sub Target_OLEDragDrop(Data, Effect, ...)
   If Data.GetFormat([format]) Then
      var = Data.GetData(...)

Sub Source_OLESetData(Data, Format)
   Data.SetData var, [format]

Sub Source_OLECompleteDrag(Effect)
```

Figure 3-7 Life cycle of OLE VB6 drag and drop operation

In VB.NET drag and drop operations between applications and between controls in a windows form has been consolidated into a single framework [7]. These changes simplify the implementation of drag and drop. However, it requires re-writing of the drag and drop code for existing applications.

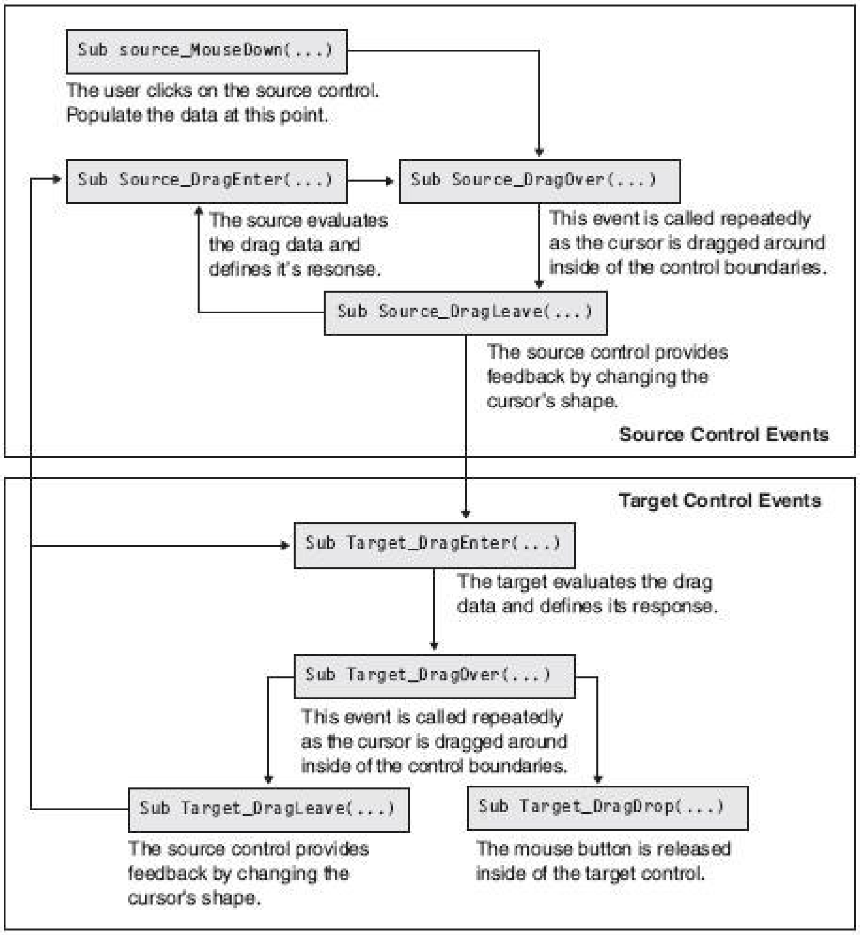

Sub source_MouseDown(...)
The user clicks on the source control. Populate the data at this point.
Sub Source_DragEnter(...)
Sub Source_DragOver(...)
The source evaluates the drag data and defines it's resonse.
This event is called repeatedly as the cursor is dragged around inside of the control boundaries.
Sub Source_DragLeave(...)
The source control provides feedback by changing the cursor's shape.
Source Control Events
Target Control Events
Sub Target_DragEnter(...)
The target evaluates the drag data and defines its response.
Sub Target_DragOver(...)
This event is called repeatedly as the cursor is dragged around inside of the control boundaries.
Sub Target_DragLeave(...)
Sub Target_DragDrop(...)
The source control provides feedback by changing the cursor's shape.
The mouse button is released inside of the target control.

Figure 3-8 Life cycle of drag and drop operation in VB.NET

It should be noted that in Figure 3- the life cycle applies to drag and drop operations between applications and between controls within the same form.

3.5 Upgrade Warnings:

The Upgrade Wizard has a set of rules for converting from VB6 to VB.NET. These rules are used for each property, event and method of a particular object type. The Wizard does not know how to interpret properties of a variable that is bound at run-time. Error Warning Issue (EWI) is the term used for errors, warnings and issues inserted into the code after upgrade. EWI comments and messages are inserted on the line before the problem and they include an underlined link to appropriate help topics. There are different types of EWIs and the major ones include Upgrade Issues, Upgrade warning and Design issues.

Apart from the compile errors, another 155 upgrade warning messages were generated. Upgrade warnings do not cause any compile errors but the application might behave differently during run time. An example of a run time warning is the *dir* function. In VB6 the Dir function was used to return the name of a file, folder or directory. It was also possible to use the strings "." and "..." which is in the path argument to represent the current directory or the parent directory. These strings are not used in the VB.NET but this will not generate a compile error. Consequently applications using the "." and ".." strings will have a different behavior while applications using the *Dir* function without the strings mentioned above will work as expected. Not all runtime warnings require any changes in the code however the application need to be tested thoroughly in order to discover components with unexpected behavior.

3.5.1 Event may Fire when Form is initialized

The differences between Visual Basic 6 forms and Visual Basic .NET forms are quite minor. Both VB6 and VB.NET designer create forms, set properties, and add controls to a form in a similar manner. However, the difference is that some events do not fire in the same order meaning some events are not triggered. An example of an event is when a button is clicked then the click event of that particular button is triggered.

In VB6 *Load* event is the first event to fire while *Resize* event occurs before *Load* event in VB.NET. Other events may occur before Load event depending on the control properties set at design time [7]. In a case where an element of a control array is created in *Form.Event* and code to position the control array element in the *Form.Resize* an event is triggered. Everything functions as intended if the *Resize* event fires after the Load Event. However, if the *Resize* event fires before the *Load* event then an error occurs in Resize code since an attempt is made to access a control array element that does not exist

Initialization of forms and controls in VB.NET is handled by the *InititalizeComponent* method, and no modification is made to the order in which the objects are initialized. However, events may be triggered during initialization and a run time error may occur if the event handler references components that have not been initialized. In the Reptile system, after the upgrade, an upgrade warning was generated UPGRADE-WARNING:

```
EventCheckStateChanged may fire when the form is initialized.
Private SubCheck1_CheckStateChnaged(ByVal sender As System.Object _,
ByVal e As System.EventArgs) Handles Check1.CheckStateChanged.
TextBox1.Text=”Check1 was clicked”
End Sub
```

Figure 3-8 Subroutine not using IsInitializing

The above code means that an error may occur if the *TextBox* control is not initialized. This was fixed by adding the *IsInitializing* property to the form, setting the property to *true* in the form’s constructor before the *InitializeComponent* call and setting to *false* after the call. The code below was then added to the event procedure to check the value of the property before running the code.

```
Private SubCheck1_CheckStateChnaged(ByVal sender As System.Object _,
ByVal e As System.EventArgs) Handles Check1.CheckStateChanged.
If Form1.DefInstance.IsInitializing = True Then
Exit Sub
Else
TextBox1.Text=”Check1 was clicked”
End If
End Sub
```

Figure 3-9 Subroutine using IsInitializing

3.5.2 Structure may require Marshalling Attributes to be passed as an Argument in this Declare Statement.

In VB.NET, a *structure* or *user define type* passed as an argument in declare statement may require additional marshalling attributes to ensure that the arguments are passed properly to a subroutine or external function. However, *user define types* in VB6 could be passed as an argument in a declare statement [31].
After upgrade, arrays and fixed strings may not function as expected, in which case they may require marshalling attributes. An example below shows a structure before and after upgrade.

```
Type MyType
s As String * 100
End Type
```

Figure 3-10 VB6 code without marshalling

Code after upgrade to Visual Basic .NET

```
Structure MyType
<VBFixedString(100),
System.Runtime.InteropServices.MarshalAs(System.Runtime.Inte
ropServices.Unmanage d Type.ByValTStr, SizeConst:=100)>
Public s As String
End Structure
```

Figure 3-11 Visual Basic .NET Code with Marshalling

As seen from the above code, the fixed string requires additional marshalling attributes in order to be passed properly. To fix this an import statement to reference the Interopservice namespace was added, the structure and the string declaration was modified by including marshalling attributes as shown below:

```
Imports System.Runtime.InteropServices
<StructLayout(LayoutKind.Sequential,CharSet=CharSet.ANSI)>
Structure MyType
<MarshalAs(UnmanagedType.ByValTStr,SizeConst:=100)> Public s As String
End Structure
```

Figure 3-12 Modified code

3.5.3 The Lower Bounds of a Collection has Changed

While running the application we discovered one warning that actually caused the application to crash during run time. Our application was in the VB6 version using imagelists to access images used for example as icons. The problem comes from the changes in the way arrays are declared in VB.NET.

```
Dim Items (5) As String
```

Figure 3-13 Array Declaration

The array declarations in **Error! Reference source not found.** in VB6 will produce an array with its index ranging from 0 to 5 yielding 6 elements. In VB.NET, this same declaration will yield 5 elements from index 0 through index 4. In VB6 array indexes can begin with any positive integer but in VB.NET all arrays are now 0 bound meaning their index always starts from 0.

So when the Reptile application tried to access the last item in the imagelist as shown in **Error! Reference source not found.** an index out of bounds exception will be thrown.

flxRepairInfo.CellPicture = imgList.Images.Item(3)

Figure 3-14 Imagelist

The exception is thrown since in VB6 the declaration of the imagelist with a value of 3 would yield 4 elements with index from 0 to 3 however after the upgrade declaring the imagelist with size 3 would yield only 3 items in the list with index ranging from 0 to 2. The solution to this problem was to change the references to the imagelist so instead of using the indexes 1 to 3 the references where changed so they now use the indexes 0 to 2.

3.5.4 Other Possible Upgrade Warnings

Apart from the upgrade warnings we encountered as mentioned in the above section there are also several other possible upgrade warnings which may be encountered when upgrading a VB6 application.
The *Null* value is no longer supported in VB.NET, when upgrading the Upgrade Wizard will upgrade the Null value into the closest equivalent [7]. As an example *System.DBNull.Value.IsNull* would be upgraded into *DBNull*. In VB6 functions could accept Null as an argument and could also return *Null*, in VB.NET this is only possible with strings. If the Null value is used in an arithmetic expression this would cause a run time exception.
There are also certain parameters which the wizard will completely remove during the upgrade such as the *Helpfile* and the *Context* parameters of the *MsgBox* and *InputBox* classes. These parameters are no longer supported in VB.NET but what makes them special is that the Upgrade Wizard completely removes them during the upgrade [7].

3.5.5 Could not Resolve Default Property

In VB6 variables a variable could be referenced without first declaring it with a Dim , private or public statement. When a undeclared variable is referenced is used VB6 would create them as *variant*. Since the actual type of *variant* variables is not known until run time, the number of late-bound variables increases and as a result the Upgrade Wizard cannot determine which operation to apply to these variables and hence Upgrade Warnings or conversion errors are generated. When the Upgrade Wizard finds undeclared variables, a warning comment is inserted on the line before the problem 'Error warning and Issue (EWI) 1037 (UPGRADE_WARNING: *Couldn't resolve default property of object 'x'*) for every use of the variable as shown in **Error! Reference source not found.** below [7].
VB6 supported default properties which could as be used as shortcuts in the code. For example, a *Textbox* control default property was the *Text* property and this could be typed as *TextBox1= "Hello"* instead of *TextBox1.Text = "Hello"* Default properties are no longer supported by VB.NET, all properties references must be fully qualified. During upgrade, early-bound objects default properties are resolved to the property name but late-bound objects cannot be resolved because it is impossible to determine a default property without knowing what the object is. The Upgrade Wizard report variables declared as objects and VB.Net declares them as Byte.
The example below illustrates a VB.Net code- generated by Upgrade Wizard with variable declaration warning.

```
Dim x As Object
Dim y As Object
Dim k As Object
UPGRADE WARNING: Couldn't resolve default property of object of x.
x =7
UPGRADE WARNING: Couldn't resolve default property of object of y.
y = 14
UPGRADE WARNING: Couldn't resolve default property of object of k.
K = 7+14
```

Figure 3-15VB.NET code with upgrade warnings

```
Private Sub Command1_Click(, , ,)
Dim x As Byte =7
Dim y As Byte =14
Dim k As Byte  = 7+14
End Sub
```

Figure 3-16 VB.Net code: with no upgrade warning

Other Upgrade Warnings include:

UPGRADE_WARNING*: Form event frmMain.Unload has a new behavior.* 'Click *for more: ms-help://MS.MSDNVS/vbcon/html/vbup2065.htm*

```
Private Sub frmMain_Closed(ByVal eventSender As System.Object, _
ByVal eventArgs As System.EventArgs) Handles MyBase.Closed
con.Close()
End Sub
```

Figure 3-17 New Behaviour

From the above warning, the Upgrade Wizard has changed the event handler for the Unload event to the Close event. Consequently, the wizard has added the Handles clause and changed the event handler signature to the .NET standard.

3.6 Design Errors

The design errors refer to properties that were set during design time. The design errors covered more than half of the overall issues generated in the upgrade report which will be shown in figure 3.33. These design errors are sub-divided into 4 different categories: *ItemData property cannot be upgraded, Property or method has a new behavior*, *Property or method was not upgraded* and *Late-bound default property reference* could not be resolved.

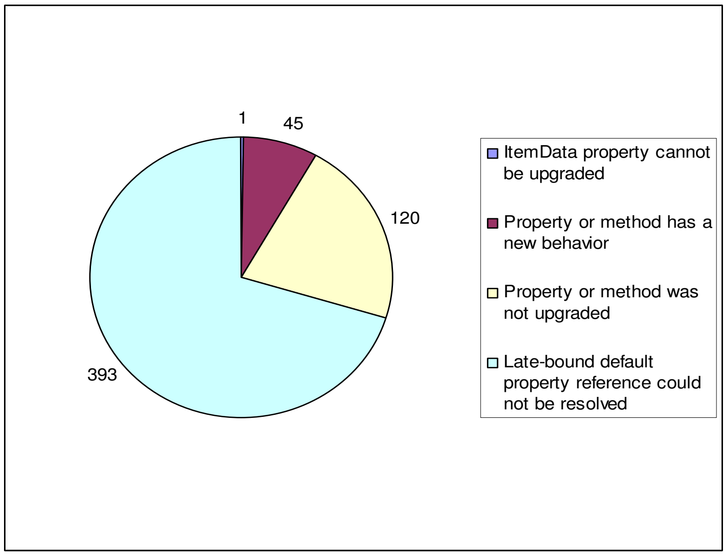


Figure 3-18 Design errors

Some of the design errors such as *property cannot be upgraded* may also generate compile errors since a property that was supported in the VB6 but is not supported in VB.NET will remain unchanged after the upgrade. This may result in the possibility that an object or a property will be declared in the code but not recognized by the .NET compiler and a compile error will occur. Other types of design errors such as *Property or method has a new behavior* will not generate any compile errors but may instead lead to unexpected behavior from the application.

3.6.1 Property not Upgraded (Design error)

A problem we encountered with regard to features that are set during design time which were not upgraded, was operations performed on the TreeNode class [24]. The Reptile application has a function where all the products are loaded from the database as a string into a cache file from which all the products are presented as a tree structure. After the upgrade the Upgrade Wizard generated the following error messages:

```
UPGRADE_ISSUE: MSComctlLib.Node property nodex.Sorted was not upgraded
nodex.Sorted = False

UPGRADE_ISSUE: MSComctlLib.Node property nodex.Selected was not upgraded
nodex.Selected = True
```

The operations “sorted” and “selected”, which in VB6 could be performed using the TreeNode class could no longer be performed directly on the TreeNode class in VB.Net.
The *selected* operation uses the get operator to check whether the node is selected or not and uses the set operator to select or deselect the node. The *sorted* operation uses the get and set operators in the same way as the *selected* operator to check if the node is sorted or to set whether it is sorted or not.
The Treeview class [25] displays hierarchical data in the form of a tree. The TreeView is made up from nodes. The nodes are objects of the TreeNode class. Nodes that contain other nodes are called parent nodes and nodes contained by other nodes are called child nodes. A node with no children is called a leaf node and the node that has no parent and is the parent to all other nodes is called the root node.
Both problems were solved by performing the operations directly on the TreeView object. So that when the node.Sorted() operation was used to sort all the nodes belonging to that particular node the statement was changed into TreeView.Sort() in order to sort all the nodes in the tree. In the same way we solved the problem with the selected property. In VB6 the node.Selected statement was used to get the node which the user had selected on the screen. This statement was changed into TreeView.SelectedNode which retrieved the node that the user has selected on the screen.

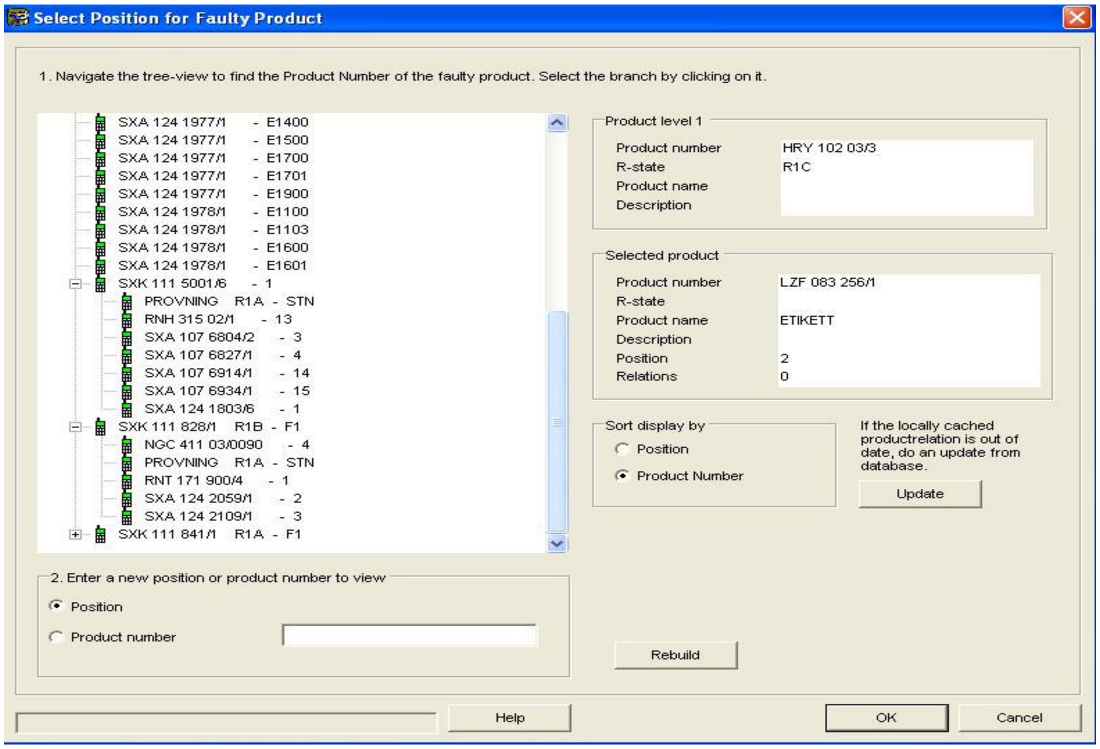
3.6.2 Other Possible Design Errors

Apart from the design errors we experienced during the conversion, there are several other design errors which may occur during an upgrade. There are several properties, events and methods which after the upgrade are marked with the warning that they now have a new behavior. This message may result in the application behaving unexpectedly since the new VB.NET property has a different behavior from the VB6 property. Examples of such issues are the *Form.Unload* and the *Control.Keypress* event [7]. These issues usually do not generate any compile errors but may eventually cause unexpected behavior during run time. There is no general solution to this problem. The solution is to test the features concerned and make the proper modifications to the code.

Another design error that may occur during a conversion is when a property of a control is not upgraded which generates the *<controltype> control <controlname> was not upgraded* message. This message concerns controls such as the OLE container and Active X and usually the issue will generate a compile error. The OLE container makes it possible to add insert-able objects to the forms in application [28]. The OLE container is not supported in the VB.NET. It is recommended to avoid using this feature at all in order to avoid being forced to deal with upgrade issues regarding OLE container after the upgrade, but it is also possible to use the Browser control in order to simulate the same functionality as the OLE container.

3.7 Changing caching from string to XML

The Reptile application has a form where all products are loaded from the database and stored as a string in a cache file from which the Reptile present the product structure in the form of a tree as shown in **Error! Reference source not found.**20.


Select Position for Faulty Product

1. Navigate the tree-view to find the Product Number of the faulty product. Select the branch by clicking on it.

```
SXA 124 1977/1    - E1400
SXA 124 1977/1    - E1500
SXA 124 1977/1    - E1700
SXA 124 1977/1    - E1701
SXA 124 1977/1    - E1900
SXA 124 1978/1    - E1100
SXA 124 1978/1    - E1103
SXA 124 1978/1    - E1600
SXA 124 1978/1    - E1601
⊟ SXK 111 5001/6    - 1
    PROVNING   R1A  - STN
    RNH 315 02/1     - 13
    SXA 107 6804/2    - 3
    SXA 107 6827/1    - 4
    SXA 107 6914/1    - 14
    SXA 107 6934/1    - 15
    SXA 124 1803/6    - 1
⊟ SXK 111 828/1    R1B  - F1
    NGC 411 03/0090   - 4
    PROVNING   R1A  - STN
    RNT 171 900/4     - 1
    SXA 124 2059/1    - 2
    SXA 124 2109/1    - 3
⊞ SXK 111 841/1    R1A  - F1
```

Product level 1

Product number	HRY 102 03/3
R-state	R1C
Product name	
Description	

Selected product

Product number	LZF 083 256/1
R-state	
Product name	ETIKETT
Description	
Position	2
Relations	0

Sort display by: Position [✓ Product Number]

If the locally cached productrelation is out of date, do an update from database.

2. Enter a new position or product number to view: [✓ Position] Product number

Figure 3-19 Reptile Treeview

Prevas requested the caching for the tree to be done using an xml file instead of being loaded as a long string into a file.

3.7.1 XML (Extensible Markup Language)

XML is a specification for creating custom markup languages. It is classified as an extendible language since it allows the users to define their own elements. XML is primarily used to facilitate sharing of data between different information systems. Data is stored within so called tags in the XML file in a structure which can be represented as a tree. This is achieved by creating tags within tags. Every XML file has to have a root element which can be seen as the root node of the tree structure.

```
<home>  </home>
```

Figure 3-20 XML Tags

Error! Reference source not found. describes an XML structure with the <home> tag as root node. The above statement can be viewed as a tree with only one level, the root node.

```
<home>
    <house>
        <address> Home street 4 </address>
                <price> 2 000 000 </price>
    </house>
</home>
```

Figure 3-21 XML entry with attributes

Error! Reference source not found. shows the previous example but with more attributes added to the <home> entity. In this example every <home> entity has a <house> attribute and every <house> entity has the attributes <address> and <price>. This can be represented as a tree with 3 levels.

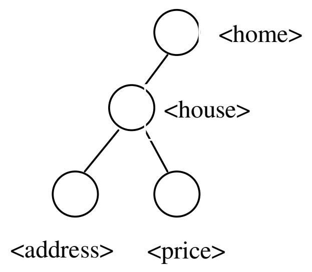


Figure 3-22 XML Tree

Previously the data was stored as a string in a file as shown in figure 3.23.

UKL 401 11/58HP;R3A;MIKROVÅGSENHET 23GHz;10;SXA 124 1958/1;R1;HUV

Figure 3-23 Caching as string

Changing the caching from a simple string in a file into an XML file has several advantages. First of all since the entire tree structure is already in the XML file the reptile application works faster since it has to read the data from the file and present the data according to the structure in the file.

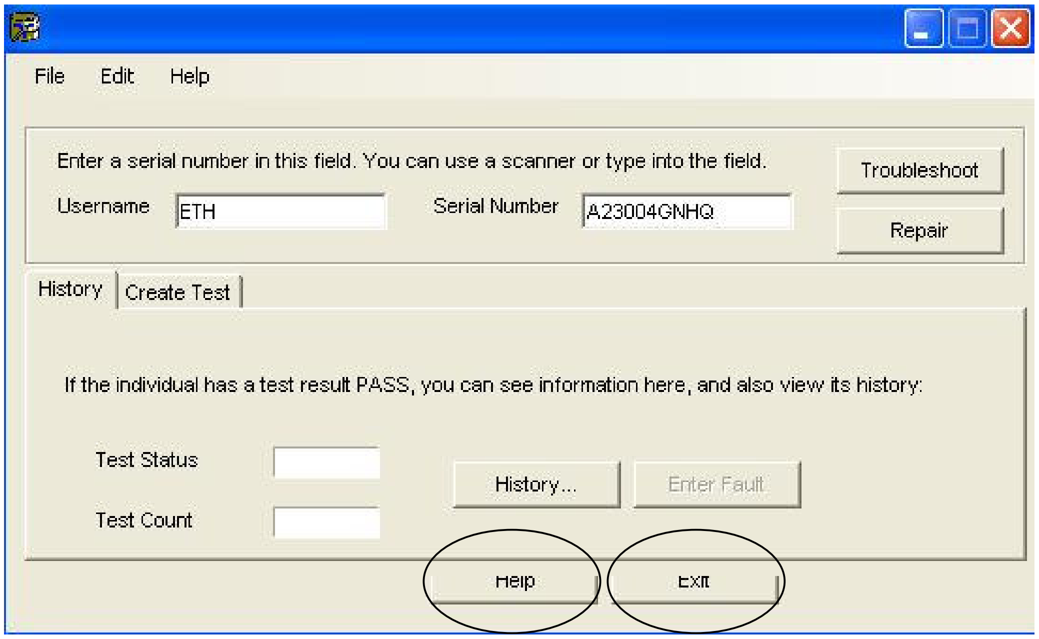
```
<Product ProductNumber="UKL 401 11/58HP" RState="R3A"
productName="MIKROVÅGSENHET 23GHz" Description="">
<Product Position="2" ProductNumber="UKM 110 60/1" RState="R2D"
ProductName="MIKROVÅGSENHET 23GHz" Description="">
```

Figure 3-24 XML based caching

Another great advantage with shifting the caching from a string in a file into and XML file is that now other applications which use XML files can use the same files with only minor adjustments.

3.8 Issues Discovered During Testing

The last phase of upgrading the VB6 project to VB.NET is the test phase. Even though the project compiles correctly, it is important to test all the features of the project thoroughly in order to discover issues which the Upgrade Wizard may not have detected. Most of the problems discovered during the test phase were related to visual features which had been changed such that text on buttons was no longer visible within the button, or text labels were hidden behind other objects.



Enter a serial number in this field. You can use a scanner or type into the field.

Username: ETH

Serial Number: A23004GNHQ

If the individual has a test result PASS, you can see information here, and also view its history:

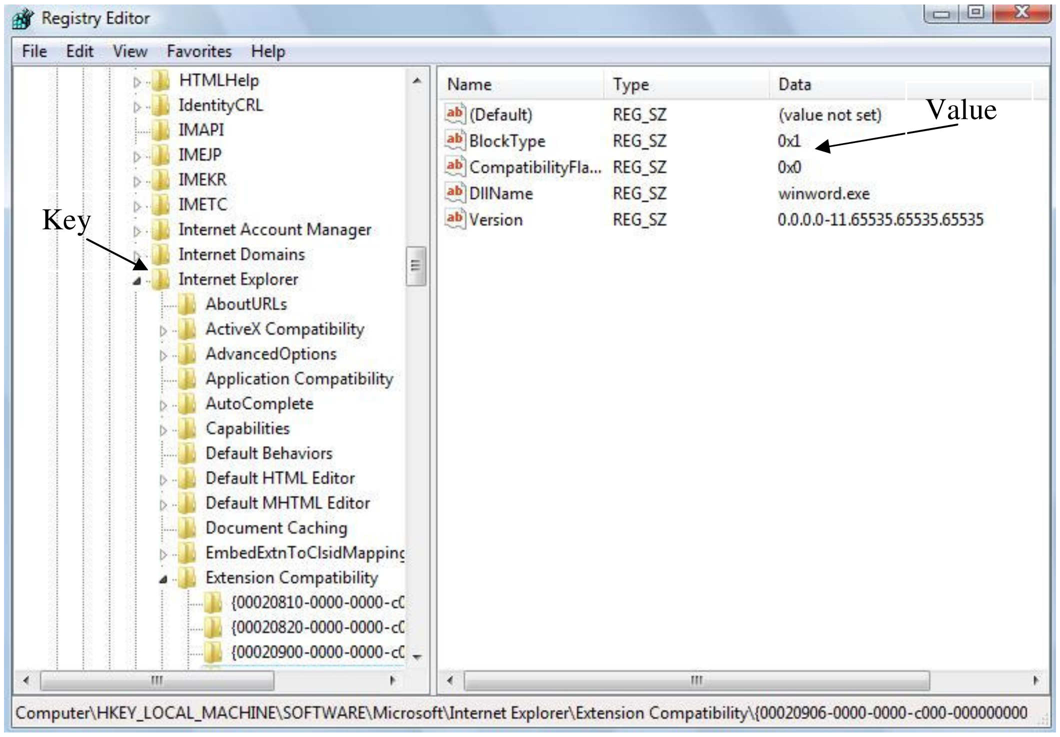
Test Status

Test Count

Figure 3-25 Buttons hidden behind Tab

The solution to the problems shown in the figure above was to either move the buttons or to resize them. Regarding the hidden text labels the solution was to pull them out and make them visible.

During testing we also encountered a problem which required more work regarding the project's interaction with the windows registry. The windows registry is a directory which stores settings and options for the windows operating systems. For example when a user make changes in the Windows control panel or most installed software those changes are reflected and stored in the windows registry. The registry contains two basic elements: keys and values.



Name	Type	Data
(Default)	REG_SZ	(value not set)
BlockType	REG_SZ	0x1
CompatibilityFla...	REG_SZ	0x0
DllName	REG_SZ	winword.exe
Version	REG_SZ	0.0.0.0-11.65535.65535.65535

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\Extension Compatibility\{00020906-0000-0000-c000-000000000

Figure 3-26 Windows registry

The keys may be viewed as folders as shown in **Error! Reference source not found.**. A windows registry key may in addition to contain values also contain sub keys. Registry values are stored within the keys and are referenced separately from the keys.

3.8.1 VB.NET Registry Changes

The Reptile application uses the windows registry to store settings such as which options should be visible and the width of the columns in the excel objects so that even if the application is terminated or the computer shut down the settings will remain the same the next time the application is started. Whenever the Reptile application is started it reads all the registry settings and when the application is terminated the current settings are stored into the windows registry.

```
Private Declare Function RegQueryValueEx
Lib "advapi32.dll" Alias "RegQueryValueExA"
(ByVal hKey As Long, _
ByVal lpValueName As String, _
ByVal lpReserved As Long, _
lpType As Long, _
lpData As Any_
, lpcbData As Long) As Long
```

Figure 3-27 RegQueryValueEx

Since this is a windows API the Upgrade Wizard did not change anything in this function therefore a compile error was generated since the *lpData* parameter was declared with *as any* as its data type. We found that when the Reptile application called the *RegQueryValueEx* function, the object that was bound to the *lpData* parameter was using byte as its data type. We therefore changed the data type of the *lpData* parameter to *as byte.*
Before the upgrade the Reptile application used the *RegQueryValueEx* function **Error! Reference source not found.**8 to read data from a specified registry key. To create new keys and modify existing settings in the registry the Reptile uses *RegCreateKeyEx* API **Error! Reference source not found.**9 to create keys and the *RegSetValueEx* API **Error! Reference source not found.** to modify existring values.

```
Declare Function RegCreateKeyEx
Lib "advapi32.dll"   Alias "RegCreateKeyExA
"(ByVal hKey As Integer,_
ByVal lpSubKey As String,_
ByVal Reserved As Integer,_
ByVal lpClass As String, _
ByVal dwOptions As Integer, _
ByVal samDesired As Integer, _
ByRef lpSecurityAttributes As SECURITY_ATTRIBUTES, _
ByRef phkResult As Integer, ByRef lpdwDisposition As Integer) As Integer
```

Figure 3-28 RegCreateKeyEx function

```
Declare Function RegSetValueEx
Lib "advapi32.dll"  Alias "RegSetValueExA
"(ByVal hKey As Integer,_
ByVal lpValueName As String, _
ByVal Reserved As Integer,_
ByVal dwType As Integer,_
ByRef lpData As Byte,_
ByVal cbData As Integer) As Integer
```

Figure 3-29 RegSetValueEx

However while testing the application we discovered that the settings were not saved in the registry. We soon discovered that VB.NET uses a completely different way to access and modify the registry. Instead of the API:s described earlier in this section VB.NET uses the Registry and the Registry Key classes within the Microsoft.Win32 namespace. The solution was to simply re-implement the read and write to registry operations. To get the value from a registry key we used the function *GetValue* feature on the registry class.

```
My.Computer.Registry.GetValue("HKEY_CURRENT_USER\"_
+ lpSubKey, szKey, Nothing)
```

Figure 3-30 VB.NET read from registry

The *lpSubKey* parameter in **Error! Reference source not found.** contains the path to the key and the *szKey* parameter contains the name of the key. The whole operation returns the value of the key as a string. In order to set values of a registry key we used the Registrykey class.

```
Dim oRegKey As Microsoft.Win32.RegistryKey
oRegKey = Microsoft.Win32.Registry.CurrentUser.OpenSubKey(lpSubKey, True)
```

oRegKey.SetValue(szKey, szValue)

Figure 3-31 VB.NET update registry key values

The first line of **Error! Reference source not found.** is the declaration of the key, and then on the second line the OpenSubKey statement in the Registry class opens the subkey which can be seen as the folder in the registry containing the key **Error! Reference source not found.**7Figure 3-26 Windows registry
Finally the *SetValue* statements are used which takes two arguments: the szKey containing the name of the key and the szValue containing the value to which the key should be set. A great advantage with the SetValue statement is that if the key which is being changed does not exist, VB.NET will create the key. Registry operations are a good example of how VB.NET has made operations easier. Instead of the old way in the previous VB versions where API:s were used and large functions such as the RegQueryValueEx function shown in Figure 3-30 *RegSetValueEx* requiring a lot of code now is simply changed into just a few lines of code.

3.9 Upgrade Summary

After the upgrade was completed the Upgrade Wizard generated a total of 1049 upgrade issues. These upgrade issues are divided into three categories: Upgrade warnings, compile errors and design errors.

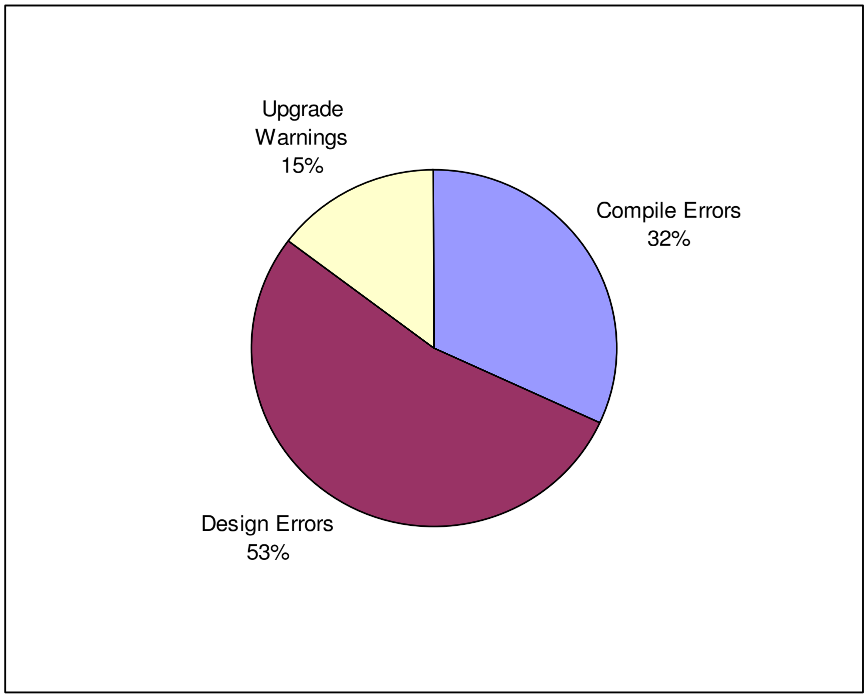


Figure 3-32 Upgrade issues

The majority of the upgrade issues we encountered after the upgrade were the design errors constituting of 53% of all the upgrade issues as shown in figure 3.32. Several of these design errors also generated compile errors since they referred to features which the Upgrade Wizard was unable to upgrade. After all the upgrade issues which generated compile errors were resolved the project had to be thoroughly tested for run time errors. While testing the project we discovered a few errors which were not mentioned in the upgrade report such as the problem with the registry operations mentioned in section 3.8-1.

4 Results

In most conversions between VB6 and VB.NET running the VB6 source code in the Upgrade Wizard is only a small part of the task. Once the wizard has completed the upgrade there are in most projects several upgrade issues such as compile errors, design errors and upgrade warnings which need to be addressed. There are several ways to prepare the code in advance in order to reduce the number of upgrade issues which have to be fixed after the upgrade in order for the project to run. One way to reduce the amount of work after the upgrade is to remove unused and duplicated code. Another issue is the removal or re-implementation features that are no longer supported in VB.NET such as declaring variables as variants which will be described in more in detail in the next section, and the App class which is no longer supported in VB.NET which means the Upgrade Wizard will not upgrade it.

4.1 Upgrade Checklist

One result of the upgrade process was a checklist consisting of 8 tasks. The checklist covers each stage of the work from the initial screening of the code before the upgrade, dealing with upgrade issues after the upgrade and finally testing the project for unwanted behavior.

4.1.1 Checklist

1. Manually pre-checking version 1 of the code for unsupported features which will reduce the number of upgrade issues after the upgrade.
2. Make changes to the code according to issues discovered during the manual pre check
3. Upgrade the project to VB.NET using the Upgrade Wizard which in turn produces an upgrade report
4. Use Microsoft Visual Studio to manually fix the upgrade issues presented in the upgrade report.
5. VB.NET project version 2. Now the project compiles and can be tested.
6. Test version 2 of the project and correct errors discovered, to give version 3
7. Hand the project version 3 to the client for testing, to give version 4 of the project.

8. Fix the faults discovered during testing of version 3 and deliver VB.NET project version 4.

4.1.2 Before the Upgrade (Checklist: steps 1 and 2)

The first step of the checklist is to prepare the code for the upgrade. The aim of preparing the code is to reduce the number of upgrade issues which have to be fixed after the upgrade is complete. First we were looking for duplicated and unused code. By removing duplicated and unused code we ensure that we will not be dealing with any upgrade issues from unused code. We were also looking for variables declared with *variant* as data type. *Variant* is a data type which can hold any other data type such as number [29], strings or references to objects. Variants are useful when a procedure is called with optional arguments [29].

```
Function ProcessAnswer( Optional ByVal Answer) as String
```

Figure 4-1 Process answer function

Figure 4.1 shows the procedure ProcessAnswer which takes one argument with *Optional* which is a variant as its data type. This means that the procedure can be called with any data type so the argument "Answer" can be for example a string or an integer. There are several disadvantages with the *variant* data type such as:

- It prevents strict type checking
- It may reduce the application performance, since variant data types require more memory and uses late binding meaning the type is determined during run time.
- It reduces readability since the code will be more difficult to understand

VB.NET does not support the variant data type. Instead VB.NET uses the *object* data type as default data type which implements all the functionality which the variant data type had. However as with the variant data type the object data type should be avoided when possible since it also has the same disadvantages as the variant data type mentioned above.

There are automated ways to check and adjust the code by using so called code analyzers. One such analyzer is Project Analyzer [27]. Project Analyzer has several useful features such as removal of unused code and a VB.NET compatibility check which detects syntax and feature conventions which have changed in VB. However we did not have access to such applications so we had to perform the code check manually in Visual Studio.

4.1.3 Application Version 2 (Checklist: step 3)

Once the code preparations were completed we ran the project through the Upgrade Wizard. When the upgrade was completed the Upgrade Wizard generated an upgrade report of 1049 upgrade issues divided into 3 categories: compile errors, design errors and upgrade warnings.

The next stage was to work with the upgrade issues. Our first target was to be able to compile the application so we started working with the compile errors. The strategy we used was to make as small changes to the code as possible. Since most of our compile errors were generated by features which the Upgrade Wizard could not upgrade we first of all tried to find equivalent features in VB.NET as in the case with the *Unable to determine which constant to upgrade vbNormal to* where we changed the declaration to *vbDefault* which is described more in detail in section 3.4.3. However in some cases we were forced to re-implement the missing functionality with a completely new function which was the case with the App.previnstance where the App class, is no longer present in VB.NET. In this case we had to implement the functionality by creating a new function as described in detail in section 3.4.1.

When all the compile errors were fixed and we were able to compile the project we could then start to work with the warnings. Some warnings required changes to the code while others did not require any action at all. One warning which required changes was when the array lower bound had changed from 1 to 0. If left unchanged this issue eventually causes the application to crash when the feature concerned is accessed. This issue is described in detail in section 3.5. The best way to find out which warnings required change is to rigorously test all the features of the application as described in section 4.1.1. Tests are performed by testing all the features of the application to ensure there is no abnormal behavior. Further we also matched some output from the VB.NET version against output from the VB.6 version of the reptile in order to make sure the functionality remains unchanged.

4.1.4 Test of application version 2 (Checklist: step 5)

After all the upgrade issues had been dealt with, we tested all the features of the project in order to discover any unresolved issues and that our solutions were working the way we intended them to. First, we discovered that there had been a number of changes in the design of the project. Label objects displaying crucial information were sometimes hidden behind other objects. These problems were easily solved in Visual Studio since it is simply a matter of changing the design by clicking on the object and dragging it to its correct location.

Further we discovered a problem with the treeview described in section 3.7 which fetches a product chosen by the user from the database and displays the product and sub products as a tree. After the upgrade the tree was not displayed correctly. The tree had the main product as the root and all the sub products lined up on the next level, while what was expected was a display of the products as a tree with several levels. Since we had a request from Prevas to change the caching of the tree from a simple string to an xml file we rewrote the function responsible for presenting the tree and the new xml based function produces a tree with the right structure.

4.1.5 Test of Application Version 3 (Checklist: steps 6, 7 and 8)

After testing version 2 and making changes to the issues we discovered in version 2 we now had version 3 of the application which we handed to Prevas for further testing. Since they originally created the Reptile application they can test features and discover abnormal behavior which we were unable to detect. When Prevas completed testing of version 3 we received a list of the following issues:

- Settings were not stored in the windows registry
- Combo boxes not visible
- Right click popup menus not appearing.

The issues with the settings and the right click popup menus were non-trivial, since they were due to new features in the .NET framework meaning we had to re-implement the functions. The missing combo boxes were simply a matter of design where they had been hidden behind other objects.

After the corrections were made to version 3 we now had version 4 of the project which is the final version and thereby the process is complete.

4.2 Project Overview

There are several ways to reduce the amount of time and work spent on the conversion process. By using a checklist, planning will be a lot easier and also more efficient. It is also important to understand what knowledge is required before beginning the conversion. By using a checklist the process can be seen as an organized process which makes planning and distribution of resources for the project more efficient.

4.2.1 Knowledge

During the conversion we learned that it is really important to have a good understanding of the two languages involved in the process. Further it is also important to know which features the upgrading software is unable to upgrade. This knowledge facilitates the necessary preparations of the source code before the conversion and thereby reduces the number of upgrade issues which has to be solved after the upgrade and consequently reduces the time spent.

4.2.2 Software Tools

There are many different types of software that will make the upgrade more efficient and reduce the time spent on getting the project running after the upgrade. When upgrading from VB6 to VB.NET, Visual Studio has a built in Upgrade Wizard which automatically upgrades the VB6 code into VB.NET. However the Upgrade Wizard does not perform the entire job.

There are objects such as the App.helpfile constant which the wizard is unable to upgrade and not always does the upgraded object have the same behavior as its predecessor as in the case of the *dir* function explained in section 3.5. This will lead to time which has to be spent after the upgrade to correct the features which have the wrong behavior or simply were not upgraded at all. To reduce this time there is software which can be used to detect such issues before the upgrade making it possible to perform changes in the code before the upgrade and thereby reduce the time that has to be spent after the upgrade to correct upgrade issues. One example of such a software tool is the Project Analyzer.

4.2.3 Planning

With the help of a checklist planning the process will also be easier. With planning a timetable can be made and also certain critical points can be determined. By using the checklist the person performing the conversion can easily know the next step in the conversion and make sure that proper preparations are made.

4.2.4 Project Summary

The result of the conversion produced a checklist consisting of 8 steps. The list began with a pre check of the code to look for features which may produce upgrade issues and thereby reduce the amount of work required to get the project running after the upgrade. After the pre check the project is upgraded and all the upgrade issues are dealt with in order to run and test the project. It is important to test the projects' features thoroughly in order to discover abnormal behavior in the application. Once all the issues discovered during testing are dealt with, the project is handed in to Prevas for further testing and most likely there will be a new list of issues to deal with. Once these issues are dealt with the conversion process is complete. With the help of the checklist the conversion process can be seen as an automated process. There are several factors which affect the result, time and cost of the conversion. Not enough knowledge of the languages and features of the languages involved can make the conversion more time consuming than needed.

4.3 Project Evaluation

It is difficult to predict how much time that has to be spent on converting a VB6 application to VB.NET. The size of the project and the components involved such as Active X or drag and drop will affect the amount of time spent on getting the project running after the conversion. Most of the time was spent on dealing with the upgrade issues which were raised by the Upgrade Wizard. However there are several ways as described in chapter 4 to prepare the code in advance in order to reduce the number of upgrade issues and thus reduce the amount of time and effort spent on the conversion.

There are several different ways to upgrade a VB6 project. One way is to simply re-write the entire application into a .NET language such as C# or VB.NET. However we chose to upgrade the project into VB.NET since we could use the Upgrade Wizard to perform the upgrade and VB.NET syntax is more similar to VB6.

4.3.1 What was learned

From this project we learned that the learning curve was higher than we expected. It is not only about understanding the languages involved in the conversion but also understanding of the code. Since the code was originally written by Prevas, it was significant to understand the code and how the different components in the application interact with each other. This would mean that any other developer performing the conversion written by another developer could experience the same learning curve.
Depending on the complexity of the application being converted we realized that substantial changes of the code may be required which increases the risk that changes in one component may affect other components of the application.
We also learned that for future conversions better preparations of the code before running it through the upgrade wizard such as, re-writing features which we no know the upgrade wizard will be unable to convert may reduce the number of upgrade issues which has to be dealt with and thereby reduce the time spent on the conversion.

4.3.2 Evaluation

Our performance in this project was to the expectation of Prevas Company and to our satisfaction too. We achieved our goal and accomplished our task by converting the Visual Basic 6 Reptile application to Visual Basic .NET. The Reptile application is now fully converted with its original functionality intact.

Having converted the Reptile application, Prevas Company can now benefit from the new and improved features offered by the .NET framework. Their products can now integrate easily with other products written in other languages under the .NET environment.

5 Future Conversions

Any successful conversion will be highly dependent on how well the code is structured. Understanding of the two languages or environment involved is essential as this would ensure that a programmer or developer is familiar with features of the languages. This would also reduce the amount of effort and time to carry out the process. In this case better understanding of object-oriented technology is crucial in understanding the .NET Framework. In our case we had no substantial knowledge of the two languages and we therefore used the conversion process as a way to learn. Even though we had some knowledge of object oriented technology, understanding the language functionality took us more time and effort than we expected.

The fact that VB.NET is not entirely backward compatible with the other previous Visual Basic versions like VB6 makes conversion a rather complex issue. Therefore to carry out any future conversion the following important issues will need to be outlined for consideration in order to carry out a more successful conversion:

- Strategy or approach
- Preparation which include evaluation and initial assessment of the target application.
- Identifying and making changes to help the upgrade go smoothly
- Tools used to assist in the process

5.1 Approach

Decision on how the conversion could be performed has to be made. The decision will determine the approach the conversion process will follow. The upgrade could take three approaches:

- Partial upgrade
- Complete upgrade
- Complete re-write

5.1.1 Partial Upgrade

In situations where a complete upgrade is not required then the target code can be passed through the Interoperability services stage by gradually adding VB.NET components into the existing VB6 application. However, this is a slower path of conversion where upgrade is performed on various sections of the application as needed.

5.1.2 Complete Upgrade

The motivation to perform an entire application upgrade could be that an application has gone beyond its original scope and therefore makes sense for conversion to a latest development platform. Again most developers perform this upgrade because they feel they want to take full advantage of the advanced functionality offered by the .Net framework. Therefore, depending on developers or organizational needs this approach could be necessary.

5.1.3 Complete Re-write

When the languages involved have very little in common or the application about to be upgraded uses a lot of components such as Active X controls and Drag and drop controls which we already know will cause upgrade issues. In some cases the amount of work involved to get the project running will exceed the amount of work required to re-write, then the easiest upgrade approach would be to re-write the application into the new language.

5.2 Preparations

Preparation is the initial stage before any given process. Most experts recommend that developers perform some manual preparations such as removal of duplicated and unused code. Moreover, better preparation as we noted may considerably reduce the number of errors that will require fixing after the upgrade. During preparation some important differences between VB6 and VB.NET must be understood and this forms a fundamental step for the process. Understanding the differences between the two languages plays an important role in understanding modifications required to the code.

5.3 Changes

Several changes should be applied to make the conversion process much more efficient. Data types such a variant data types in VB6 which accepts any data as input have to be re-written. Though this construct makes coding easier since no strict type definition is required, there is no available support in VB.NET instead the variant data type is replaced by object. Therefore searching VB6 Code for all instances of variant data type and replacing them with an object data types is essential at this stage.

Late binding should be avoided by identifying variant data types in order to eliminate the number of unresolved upgrade issues.
Use constants instead of underlying values as described in section 3.4.3. VB6 constants will convert to the correct value when upgraded to VB.NET, but if actual values are used then the constant may end up with wrong values. For example, for Boolean values using -1 and 0 instead of True and False will have an adverse effect in VB.NET because inVB.NET True is 1 [7].

5.3.1 Fixing Data Declaration

In VB6 it was possible to reference a variable without first declaring it with a Dim, Private, or Public declaration. However this feature is no longer possible in VB.NET. Adding proper type declarations on any undeclared variables and adding Option Explicit statements on every *Dim* statement would enable proper conversion of the code. In addition, certain array declarations and user defined structures should be changed. In VB6 an array can be declared with any positive integer as its lower bound but in VB.NET all arrays start from index value 0 (zero).

5.3.2 Obsolete Keywords

Certain language elements which are not supported in VB.NET should be removed or replaced in VB6 application before upgrade. Obsolete keywords such as GoSub, Option Base, IsMissing, Let, Wend, VarPtr, ObjPtr, StrPtr, DefBool, DefByte, DefInt, DefLng, DefCur, DefSng, DefDbl, DefDec, DefDate, DefStr, DefObj, and DefVar are no longer useful in VB.NET.

5.4 Tools

After making the necessary changes in the preparation stage then the next step would be to find the right tool to assist the conversion process. Microsoft has introduced several resources to assist in the conversion process. Depending on what is available such as Visual Studio Upgrade Wizard, these resources would definitely make the process easier. One of the tools used for this process is an Upgrade Wizard provided by Visual Basic .NET which performs an automatic upgrade. The Upgrade Tool is started when you open a Visual Basic 6 project or application in Visual Basic.NET. The Upgrade Tool converts the code but does not change the Visual Basic 6 project. Instead a new Visual Basic .Net project is created. Most of the upgrade work is handled by the Upgrade Wizard but there are areas where an upgrade issues are raised and manual modifications is required to fix the issues.

5.4.1 Migration Cost Estimation Tool

Another example of an available tool is, *migration cost estimation tool* [27]. This tool is recommended for:

- Analyzing the code and determining the complexity of the code
- Looking for issues that could make difficult using the Wizard and

By doing so, the developer can easily make a schedule for the process and understand how to deal with upgrade issues.

5.4.2 Project analyzer and Compatibility Check Tool

Project Analyzer is another tool available for Visual Basic which includes *automatic dead code removal* and *VB.NET Compatibility Check features* for understanding, optimizing and documenting Visual Basic Source Code. Project Analyzer helps in preparing the existing VB6 code for VB.NET by checking whether or not an application is upgradable.

Automatic dead code removal helps to remove the pieces of code that are not in use. This could also reduce the amount of the code that the Upgrade Wizard will have to scan through [27].

5.5 Conclusions

During this conversion we learned the importance of proper preparations of the code before the conversion. Proper preparation of the code before conversion can reduce the amount of work and time spent on the conversion. There are several tools available such as project analyzer and compatibility check tool which could also assist in code preparation.

We realized also that the Upgrade Wizard assisted a great deal in substantially increasing automation of the conversion process which indeed reduced the amount of manual modifications. When choosing which .NET language to upgrade to VB.NET has the advantage that the Upgrade Wizard can be used.

At the beginning of the assignment we had limited knowledge about the languages involved and the differences between them, this meant that part of the time had to be spent in learning and understanding the structures of the code. We learnt that prior knowledge of the languages involved in the conversion could play an important role especially in handling upgrade issues such as objects which could not be upgraded and had to be re-implemented.

References

Books references:

1. Bradley L. Jones, Sams Teach Yourself C#, Pub: Sams 2003
2. Conard James , Introducing .NET, Pub: Wrox Press 2001
3. Deitel, Paul J, Visual Basic 6 :How to Program , Pub: Prentice Hall 1998
4. Drayton Peter, C# In a Nutshell , Pub: O'Reilly Media 2002
5. Evjen Bill, Visual Basic.NET Bible, Pub: Wiley2001
6. Kastroulis Angelo, Visual Basic .NET for VB6 Developers, Pub: Peer Information 2002
7. Ed Robinsson, Upgrading Microsoft Visual Basic 6.0 to Microsoft Visual Basic .NET, Pub: Microsoft Press 2001
8. Smith, Eric A. h Valor Whisler , Hank Marquis, Visual Basic 6 Bible , Pub: Wiley 1998
9. Watkins Damien, Programming in the .NET Environment , Pub: Addison-Wesley Professional 2002

Web references:

10. Introduction to ActiveX Controls,
    http://msdn2.microsoft.com/en-us/library/aa751972.aspx
11. COM (Component Object Model)
    http://msdn2.microsoft.com/en-us/library/ms680573.aspx
12. Compiling MSIL to Native Code
    http://msdn2.microsoft.com/en-us/library/ht8ecch6.aspx
13. Rapid Application Development
    http://www.blueink.biz/RapidApplicationDevelopment.aspx
14. Delphi for .NET
    http://www.delphibasics.co.uk/NET.html
15. Visual J#
    http://msdn2.microsoft.com/sv-se/vjsharp/default(en-us).aspx

16. Console Applications

http://www.startvbdotNET.com/language/console.aspx

17. What is Visual Basic?

http://visualbasic.about.com/od/applications/a/whatisvb.htm

18. Virtual Machines

http://www.answers.com/topic/virtual-machine?cat=technology

19. Just in Time Compilation(JIT)

http://msdn2.microsoft.com/en-us/library/ht8ecch6.aspx

20. Cursors. default

http://msdn2.microsoft.com/en-us/library/system.windows.forms.cursors.default.aspx

21. Wait Cursor

http://msdn2.microsoft.com/en-us/library/system.windows.forms.cursors.waitcursor.aspx

22. Multidimensional Arrays in Visual Arrays in Visual Basic

http://msdn2.microsoft.com/en-us/library/d2de1t93.aspx

23. OLERequestPendingTimeout Property

http://msdn2.microsoft.com/en-us/library/aa245850(VS.60).aspx

24. TreeNode Class

http://msdn2.microsoft.com/en-us/library/system.web.ui.webcontrols.treenode.aspx

25. TreeView Class

http://msdn2.microsoft.com/en-us/library/system.web.ui.webcontrols.treeview.aspx

26. Process Class

http://msdn2.microsoft.com/en-us/library/system.diagnostics.process.aspx

27. Migrating from VB6 to VB.NET with Project Analyzer

http://www.aivosto.com/project/vbnet.html

28. OLE container

http://msdn.microsoft.com/en-us/library/hkfbayab(VS.80).aspx

29. Variant Datatype

http://en.wikipedia.org/wiki/Variant_type

30. App class

http://msdn.microsoft.com/en- us/library/fc353bw2.aspx

31. Marshalling attributes

http://msdn.microsoft.com/en-us/library/dh1fz21y(VS.71).aspx

32. Prevas Company

http://www.prevas.com/

A Glossary

VB	all previous versions of visual basic
VB6	Visual Basic 6.0
VB.NET	Visual Basic.NET
IDE	Integrated Development Environment
COM	Component Object Model
CLR	Common Language Runtime
CTS	Common Type System
FCL	Framework Class Library
CIL	Common Intermediate Language
JIT	Just in Time Compilation
EWI	Error Warning Issue
OLE	Object Linking and Embedding
RAD	Rapid application development
CLS	Common Language Specification
IL	Intermediate Language
API	Application programming interface
DLL	Dynamic Linked Libraries
ODBC	Open Database Connectivity