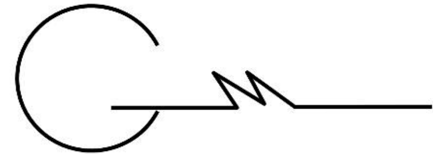
Introduction to the Great Migrations Technology



Mark E. Juras, Partner
Great Migrations LLC
November 2008

Executive Summary 2
  Overview 2
  What is gmBasic? 2
  Why gmBasic? 2
  Why should I read this document? 2
Introduction 3
  Document Purpose 3
  Intended Audience 3
The ScanTool Sample Application 3
Desired Migration Target(s) 4
Toolset Overview 4
A PiecePort Migration 4
  The Problem with PiecePorts 4
A SmartPort Migration 5
  Steps to SmartPorting the ScanTool Sample 5
  The problem with SmartPorts 5
A CleanPort Migration 6
The Problem with SmartPorts Revisited 6
Choice of .NET Language 6
Migrating to .NET Components 7
  Steps To Migrate A Third-Party COM Component To A .NET Component 7
The FMStocks Sample (ASP to ASP.NET) 8
  Tool-Assisted ASP Migration 8
    Analysis 9
    Migration 9
Other Translation Topics 9
  Error Handling to Exception Handling 9
  Control Arrays to Arrays of Controls 10
  Late Binding 10
  Weak to Strong Typing 10
Special Features 11
  Source Code Analytics 11
  Build Order Report 11
  Translation Control Scripts 11
  Dealing with "Bad Code" 11
Appendix A: Great Migrations Methodology 13
  Tool-Assisted 13
  Iterative 13
  Test-Driven 14
  No-Code Freeze 14
  Measurable 14
  Repeatable and Documented 14
  Balanced Application of Automated and Manual Development 14
Appendix B: Navigating the Sample Files 15

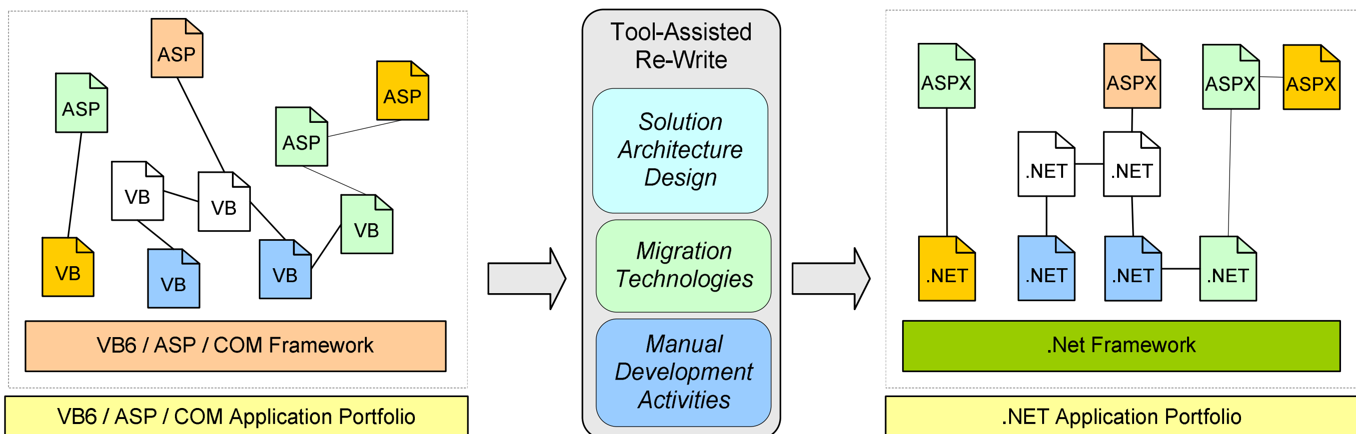
Executive Summary

Overview

For almost past two decades, Microsoft Visual Basic (VB6) and Microsoft Active Server Pages (ASP) have been useful and popular tools in software development. However, their long term viability ended when Microsoft released the .NET platform and started aggressively promoting it as the dominant programming paradigm for Windows. VB6 and ASP Classic are now being replaced by Microsoft .NET and they are rapidly losing community support. Because the .NET platform is so different from VB6/ASP, many organizations are finding the upgrade to be extremely difficult. Great Migrations has developed a methodology that can help in making this transition. Central to our methodology is gmBasic -- a powerful computer language translation technology. This document provides an introduction to the gmBasic technology.

What is gmBasic?

gmBasic is a computer language translator designed for large and complex VB6/ASP to .NET migrations. Its purpose is to help organizations fully leverage their legacy code as they adopt Microsoft .NET.



Why gmBasic?

gmBasic is based on a sophisticated and robust approach to software translation. It uses proven principles of compiler design to provide accurate, clean, and correct translations. It has several unique capabilities:

Multi-Project gmBasic is designed to migrate large, interdependent codebases, not only single projects. This means fewer Interops, which means cleaner translations, and less rework.

Choice of Language One of the most sacred principles of .NET is that developers should have language choice; we also believe in this, so gmBasic currently offers translations to both C# and VB.NET.

Flexibility gmBasic translations are controlled through configuration files. This facility is ideal for the radical restructuring needed when migrating from VB6/COM to the .NET Platform.

Performance gmBasic is fast -- orders of magnitude faster than the VB Upgrade Wizard. This speed makes experimenting with different restructuring configurations a pleasure.

Special Features gmBasic has a powerful reporting subsystem that provides source code analytics, and a job control language for directing translations.

Maturity gmBasic is being developed by Great Migrations and Promula Development Corporation (PDC). PDC has almost thirty years of software translation experience and an impressive history of delivering migration projects.

Why should I read this document?

This document illustrates the capabilities of gmBasic in the context of three introductory examples. If you have significant VB6 or ASP assets and you are interested in upgrading them to .NET in a timely and cost effective manner, then you may find this an informative document and our approach helpful in addressing this problem -- at lower cost, less risk, and without having to sacrifice architectural quality or control.

Introduction

Document Purpose

The purpose of this document is to introduce the fundamentals of the Great Migrations methodology and demonstrate some of the capabilities of the gmBasic technology. This is done in the context of a two small but non-trivial examples. The document also presents some of the special features of gmBasic such as:

- Combined VB6/ASP to .NET migrations
- Replacing third-party COM components with .NET assemblies rather than using COM Interop assemblies
- Supporting choice of language (VB.NET or C#)
- Dealing with VB6-to-C# incompatibilities (e.g., error handling, control arrays)
- Creating source code analytics reports

There are two sample applications:

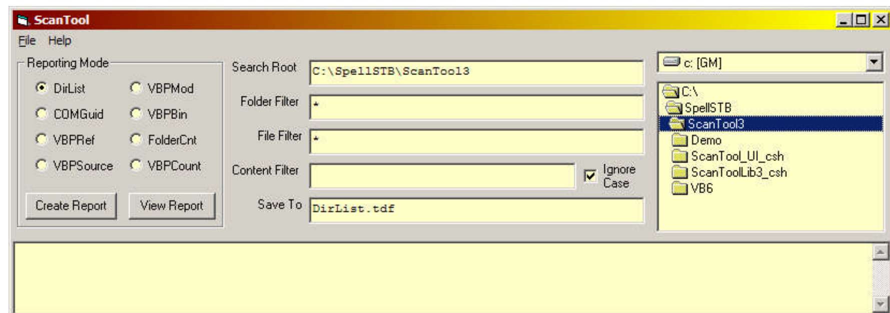
- **ScanTool** – a simple desktop application that illustrates solutions to a number of translation problems such as multi-project translations, translation of COM events, API replacements, GoTo style error handling, etc.
- **FMStocks** – a three-tier web application with a data access layer and business object layer written in VB6 and a web user interface written in classic ASP.

Intended Audience

Individuals planning an upgrade from VB6/ASP to .NET should read this document. Although this document tries to stay at a high level, a basic knowledge of VB6/ASP/COM is assumed. More experienced readers are invited to dig into the sample source codes and other supporting files which are hyperlinked throughout the text. An appendix provides guidance to readers who wish to inspect the sample source code and translations in detail.

The ScanTool Sample Application

The first sample application is called ScanTool. It is a desktop application that scans and analyzes source code directory trees and generates useful reports from the analysis of the code files. For example, one report shows the structure and size of VB projects in the directory tree. Another report shows information about COM components referenced by the code. The program has an object-oriented design, with a different report class handling each type of report.



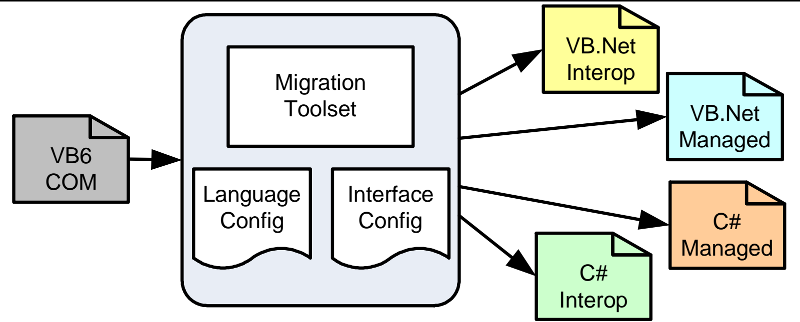
The ScanTool source is comprised of 2242 lines of code and is organized into two VB6 projects: an EXE (ScantoolUI) that provides a user interface and a DLL (ScanToolLib) that does file scanning, file parsing, and reporting. The DLL communicates its status to the EXE through COM Events. ScanTool uses several external COM libraries: MSXML, Scripting, Common Dialog, and TypeLibInfo, and it also calls Win32 APIs, like ShellExecuteForExplore.

VBPFILE	MODTYPE	File Count	Line Count
ScanToolLib.vbp	Class	8	1366
	Module	1	154
		9	1520
ScanToolUI.vbp	Form	1	523
	Module	1	199
		2	722
Grand Total		11	2242

Desired Migration Target(s)

For the purpose demonstrating the flexibility of our approach, we will perform four different migrations:

1. VB.NET with Interoped COM components
2. C# with Interoped COM components
3. VB.NET with Managed components
4. C# with Managed components



Toolset Overview

The gmBasic toolset includes the following components:

System Tools — The true workhorse of the toolset that performs the actual translation. Currently, the system tools are command line executables.

Language Files — A set of XML files that describe how the objects of the VB language map to .NET and describe standards for naming and other settings in the resulting .NET projects.

Interface Description Files and RefactorLibrary Files — A set of XML files that describe how external objects (COM Types) map to .NET Types.

A PiecePort Migration

The most direct approach to migration is called a PiecePort. A PiecePort converts one VB project at a time and uses Interop to integrate with external COM components. A PiecePort migration gives you a functionally equivalent, architecturally similar .NET code. The structure of the system remains the same unless it *must* change because of incompatibilities.

The Problem with PiecePorts

In the PiecePort of ScanTool, the ScanToolUI.exe Interops ScanToolLib.dll even though a managed ScanToolLib.dll is planned. This gives rise to a very unsavory complication of COM Interop we call *API-Crossing*.

Sidebar: API-Crossing

COM interfaces have COM types as member parameters and return types. If you Interop COM components, your .NET clients will also have to use the COM types. Your application will end up straddling the fence between COM and .NET and this will require more Interop code which runs counter to the premise of adopting .NET in the first place.

Consider for example a COM component that returns a COM Scripting.File object as part of its interface. Suppose your coding standards call for .NET's System.IO instead of COM Scripting. Interoping the COM component forces you to use Scripting.File in your .NET client because the COM component only speaks Scripting. The .NET client must contend with two File IO libraries.

Interoped COM Component
COM Scripting
.NET Application
Scripting.File
System.IO.FileInfo
COM Scripting
.NET System.IO

In the real world, migrating large codebases with many interdependent projects is the norm, and API-Crossing can create a lot of additional work and risk. One solution to this problem is called a **SmartPort** which is the topic of the next section.

A SmartPort Migration

In a SmartPort, related components are migrated as a logical unit so that *your* components are migrated and accessed as managed code rather than being Interoped. This means cleaner translations and less rework. gmBasic was specifically designed to do SmartPorts. SmartPort migrations are called "smart" because gmBasic is smart about how it deals with multiple, interdependent projects.

Steps to SmartPorting the ScanTool Sample

1) Verify that the VB codebase builds.

This ensures that all the files and dependencies are in place, the VB code is syntactically correct, and its references are in sync with your workstation.

2) Verify that Interface Description files are present.

Interface description files tell gmBasic how to map COM Types to .NET Types. ScanTool uses several external COM libraries and we need interface descriptions for these before we can translate any VB code that uses them. Great Migrations has already created Interface description files for many of the most popular COM components.

Migration Studio automates the process of creating interface descriptions for your codebase.

3) Translate the library project.

Translations are simple command-line operations. For example, to translate ScanToolLib.vbp, enter the following command:

```
gmBasic ScanToolLib.vbp
```

This creates a *code bundle* file called ScanToolLib_csh.bnd that contains all the translation results. Bundling the results into one file helps to keep things organized and manageable during large migrations.

4) Deploy the library code bundle.

The translation bundle for the library contains translations for all source files in the library. It also contains an interface description file for the DLL. The interface description will be used in translating the client project, ScanToolUI.VBP

Deploying a code bundle is a simple command-line operation. To deploy the ScanTool code bundles enter the following command:

```
deploy ScanToolLib_csh.bnd deploy replace verbose
```

5) Translate and deploy the user interface project.

```
gmBasic ScanToolUI.vbp
deploy ScanToolUI_csh.bnd deploy replace verbose
```

6) Open the ScanToolLib project in the Visual Studio IDE and compile or build using MSBuild.

7) Open the ScanToolUI project in the Visual Studio IDE and compile or build using MSBuild.

At this point, you will have a functionally equivalent C# version of the ScanTool application.

The problem with SmartPorts

SmartPorts are comprehensive, bottom-up migrations and they are the cleanest, fastest way to take an application portfolio to .NET:

- You gather up your VB codebase,
- You translate it to .NET from the bottom-up (i.e. dependencies first),
- You build the .NET code,
- You test the .NET applications,

- You deploy the .NET application to production, and then
- You celebrate.

This sounds feasible in theory, but you will probably run into the following problem with your project timeline: testing your new .NET codebase will tie up your test team for a long time. Also during this lengthy testing effort, making functional changes to the .NET code may be discouraged because they will complicate regression testing. If you decide to continue maintaining and deploying the VB codebase, while you are manually fixing and testing the .NET version, you will have to contend with merging those changes into the .NET code, doing them at least twice. This approach can quickly become very expensive or very disruptive to ongoing maintenance, both of which are most unsavory to the users of the system.

Seems like quite a quandary: you can't PiecePort the projects separately because that gives you Interop translations you do not want; you can't SmartPort the projects together because you cannot regression the entire system in a timely manner. What you need is a clean, incremental approach to migration – an approach that avoids the rework of a PiecePort, and also avoids the all-or-nothing proposition of a SmartPort. The answer to this is what we call a **CleanPort,** which is the topic of the next section.

A CleanPort Migration

In a CleanPort, the entire codebase is migrated as a unit, but some legacy components are accessed through wrappers until they can be migrated and tested in .NET. The wrappers, also known as .NET External Wrapper Components, or **NEWCs**, are .NET components generated by gmBasic. NEWCs provide a cleaner, more final interface than the wrappers generated by Visual Studio tools. In essence, a NEWC is the migrated interface of a COM component. Because the NEWC is a very thin wrapper around the original COM component, it requires less testing than the migrated implementation would. When the legacy COM component is translated to .NET, it replaces the NEWC, but it does this without changing the interface exposed to clients so there is much less impact.

The CleanPort approach is a variation on the SmartPort: In a SmartPort, gmBasic assumes the COM component will be replaced by a managed component, whereas in a CleanPort, gmBasic assumes the COM component will be replaced by a NEWC. In both cases, clients are translated to use clean, managed references, not Interop. The CleanPort approach allows an organization to migrate their code incrementally with minimal rework. This may have benefits from a schedule flexibility and risk management perspective. Beware however, that introducing NEWCs adds complexity to your build and deployment processes because they represent additional components that must be built and deployed. In general we encourage people to go the SmartPort route if they can commit to invest in *tuning* the translation process in advance.

The Problem with SmartPorts Revisited

Recall that the problem with SmartPorts is that doing the bottom-up migration of an entire codebase may demand a lot of testing before you can begin deploying your .NET application because the foundation of the system is changing as well as the client applications. However, we still find that this is an efficient approach and more time spent tuning means less work and risk in the long run.

Side Bar: Translation Tuning	Translation Tuning is the iterative process of making translations with gmBasic, testing/inspecting the results, and refining the gmBasic configuration so that, in the next iteration, gmBasic produces code that is defect free and compliant with target architecture standards.

Because gmBasic translations are fast, robust, and repeatable, functional changes to the source code can continue in parallel with the tuning process. Once the translation configuration is tuned to a client's requirements, the entire codebase can be quickly translated, built, and deployed with minimal re-testing.

Choice of .NET Language

One of the principles of .NET is that developers should have their choice of language; we also believe in this, so gmBasic currently offers translation to both C# and VB.NET. The target .NET language, or dialect as we call it, may be specified as a switch on the gmBasic command line.

```
gmBasic ScanToolLib.vbp csh – produces a C# translation (default)
gmBasic ScanToolLib.vbp vbn – produces a VB.NET translation
```

The gmBasic configuration files control many of the dialect differences. For example, the settings and structure of project files are controlled by template files. Also mappings for language and interface constructs for the different dialects can be controlled by dialect-specific entries in the language and interface description files.

Key Point: **gmBasic does NOT use the VB.NET code from the VB Upgrade Wizard.**	The VB Upgrade Wizard (VBUW) does a pretty good job of PiecePorting simple VB6 codes to VB.NET, but it has a number of limitations and flaws. These have been discussed elsewhere. Despite its weaknesses, we frequently hear about people trying to translate VBUW output to C# as a means of upgrading VB6 to C#. This approach suffers the same limitations and flaws as the VBUW and we do not consider it a productive option. Note: gmBasic does not run the VB Upgrade Wizard to produce VB.NET then translate the VB.NET to C#. gmBasic does a global, semantic VB6-to-.NET translation and then writes its output in either C# or VB.NET syntax.

Migrating to .NET Components

When you have the VB source code of a component, SmartPorting allows you to bypass Interoping it. In this case, gmBasic controls the translation of the component, so it controls every detail of its .NET interface, and therefore it is able to generate a managed interface description file for the component. Using this managed interface description file, gmBasic then produces client translations that access the managed component.

Things are very different with COM components for which you do not have source code. In this situation, the .NET interface is a runtime callable wrapper (RWC) contained in an Interop assembly file. RWCs work, but they present some challenges. These challenges have been discussed elsewhere and they will not be covered here. Furthermore, it is commonly accepted that if there is a managed replacement available for a COM component, you will almost always want to use it rather than Interoping to the COM component. This presents its own set of challenges because most .NET components have very different designs from the COM components they replace. Restructuring client code requires knowledge of the client application architecture as well as a mastery of the source and target API's syntax *and* semantics. However, with careful research one can map the coding patterns for one API to those for another. If you document this mapping in gmBasic interface description format, gmBasic can then use it to automatically restructure your client code to use the new API.

Steps To Migrate A Third-Party COM Component To A .NET Component

For the purposes of this demo, we will show that gmBasic can translate two popular COM APIs to their managed replacements:

- Windows Scripting Host to System.IO
- MSXML to System.XML

As already discussed, the output generated by gmBasic is to a great degree controlled by what we call interface description files. These are XML files that can specify rules for mapping one COM API to another. By default, these files direct the translator to generate Interop code, but they can be easily modified to direct the translator to use completely different classes, properties, and methods. A full discussion of how this is done is beyond the scope of this document, but the basics are described below.

1) Prepare a RefactorLibrary.	A RefactorLibrary is an XML file that specifies in detail how the classes, enums, properties, methods, and events of a source API map to a target API(s). The amount of effort needed to implement a RefactorLibrary on the differences between the source and target APIs' syntax and semantics. If the two APIs are one-to-one, creating the interface description will be fairly mechanical and simple. If the two APIs have very have different semantics (e.g. different object models, different design patterns, different exception profiles) then the work is more complex. In necessary deep migration operations may by implementing in a standard Win32 DLL and assigned to the RefactorLibrary. Creating a managed interface description for a COM component requires knowledge of the gmBasic interface description language as well as a mastery of the source and target API's syntax and semantics. The GM staff has already authored many of these for the most popular COM to .NET replacements, including Scripting and MSXML, which are used in the ScanTool demo.

2) Deploy the RefactorLibrary	Deploying a RefactorLibrary means putting the RefactorLibrary file into a migration project folder.
3) Retranslate, deploy, and build.	At this point you have a functionally equivalent SmartPort of ScanTool that has been restructured to use System.Xml and System.IO instead of MSXML and Scripting, respectively.

The FMStocks Sample (ASP to ASP.NET)

The migration concepts introduced in the context of the ScanTool demo also apply to ASP-to-ASP.NET translations. For the most part these concepts are all handled the same way for ASP as they are for VB6. In fact, gmBasic compiles VB6 and ASP to the same internal, intermediate format before analyzing and restructuring it to the target architecture. However, in addition to a programming language change, in this case VB Script to C# or VB.Net, ASP brings several new kinds of migration problems:

- Subtle changes in the semantics of core ASP classes (Request, Response)
- Scoping matters relating to nested and stacked ASP include files
- Very weak typing and other difference between VB Script and Visual Basic,
- Different rules for structuring markup and various styles of script tags
- Different format and semantics of ASP directives
- Looser coupling to COM, and
- Architectural decisions regarding where and how to use the many powerful new features of ASP.NET

The FMStocks sample illustrates how gmBasic handles many of these matters for you. FMStocks also provides another example of VB to .NET migration because there is a set of VB6 COM objects behind the pages.

FMStocks, the Fitch and Mather Stocks 2000 sample, was created by Vertigo software and distributed through MSDN to demonstrate some of the best practices for building scaleable web applications using ASP and COM+. FMStocks was also featured in the Patterns and Practices guide as an example of how to upgrade VB6 to .NET. It consists of the following:

- Six VBPs (2500 LOC)
  - 2 Data Access
  - 4 Business Objects
- 33 ASP pages (2300 LOC)
  - 5 include files (nested and stacked)
  - 28 script pages
- Poster Child DNA Web App
  - Portfolio Management
  - Research/Buy/Sell Stocks
  - On-line Store
  - Forms-based Login

Web Pages (ASP/VBS)
Business Objects
Data Access Objects (ADODB)
Stored Procs SQL Server

Tool-Assisted ASP Migration

ASP.NET offers a very different programming model based on server-side controls, separation of HTML from business logic in code-behind files, and many other new web application framework classes. From the programmer's perspective, ASP.NET application logic will typically look and behave very different from ASP classic. Despite these differences, migration tools can still facilitate your adoption of .NET so you can begin taking advantage of the benefits of building everything with new developer tools and running everything on the new platform. Migrating an ASP application to ASP.Net also provides you with C# or VB.NET code that is inherently more robust than the VBScript being replaced. This output from the tool can be run more or less as it was in ASP classic, or it can be restructured further, by hand or by customizing gmBasic, to fit new architectural patterns such as server-controls or web-services.

gmBasic currently does the following tasks:

Analysis

- Reporting the include order of the system. This is used in planning the order of migration and making design decisions regarding creation of web controls.

- Reporting the definitions of subroutines and variables across the entire codebase. This is used in estimating the size and complexity to the logic in the ASP pages.

- Determine the dependencies between items across the codebase and on external COM components. This is also used in estimating the complexity of the ASP pages and in identifying needs and opportunities to eliminate COM Interop.

Migration

- Restructure ASP to ASPX/ASCX and CodeBehind
    - Page Directives
    - Consolidate Script code into a single script block or a code-behind file per page
    - Translate Render Functions to Response. Write
    - Standardize HTML
    - Convert #Includes to Web User Controls
    - Implement page directives and logic to connect pages and includes
    - Create a Web Application Project File

- Translate VBScript to C# or VB.NET
    - Strong Typing everywhere
    - Integrate with translations of VB6 components
    - Deal with different semantics of ASP intrinsic collections (Response, Request)
    - COM replacements

There are many examples of how we do these tasks in the sample translations.

Other Translation Topics

Error Handling to Exception Handling

VB6 and C# have radically different error handling models, and one of the most difficult aspects of upgrading VB6 code to C# is restructuring VB6's On Error GoTo coding patterns to C#'s try-catch-finally coding patterns. Migrating to .NET APIs with different exception profiles than the original components compounds the problem. Fortunately, dealing with perplexing migration problems is precisely what gmBasic is designed to do.

In general, gmBasic follows the principle of simpler is better: removing VB error handling structures that do not add anything to what is now done by the .NET exception system while still leaving the logic that did clean up or did something significant with the error.

For example, in VB a programmer may catch an error just to call re-raise it and add the location of the error. This is not necessary in .NET because the exception object maintains an internal stack trace for you. When gmBasic sees an error handling do nothing more than a re-raise, it removes it completely. However, if there was logic in the error handler that did cleanup, that logic will be moved to a finally clause. Of course, gmBasic also knows to restructure variable declarations to account for changes in local scope that result from inserting a try block.

Perhaps the most notorious VB6 statement is On Error Resume Next; this tells VB6 to trap and ignore all errors without raising them. This statement has no clean analog in C#, and naïve approaches like just put a "try-catch around every statement" quickly become a mess with anything other than a simple sequential series of statements. gmBasic has algorithms specifically for detecting and converting On Error Resume Next to try-catch. An example is shown below.

<table>
<tr><th>VB6 Using On Error Resume Next</th><th>C# Restructured to use try-catch</th></tr>
<tr><td>

```
On Error Resume Next

Dim tliApp As TLI.TLIApplication
Set tliApp = New TLI.TLIApplication
Dim tlinfo  As TLI.TypeLibInfo
Set tlinfo = tliApp.TypeLibInfoFromFile(fil.Path)

If Err = TLI.tliErrCantLoadLibrary Then

    Err.Clear
    libname = "-"
    libguid = "-"

Else

    On Error GoTo ErrorHandler
    libname = tlinfo.name
    libguid = tlinfo.Guid

End If
```

</td><td>

```
try
{
   GmBasic_Application.ClearErrorObject();
   tliApp = new TLI.TLIApplication();
   tlinfo = tliApp.TypeLibInfoFromFile(fil.Path);
}
catch
{
    VBNET.Information.Err().Number =
    (int)TLI.TliErrors.tliErrCantLoadLibrary;
}
if (VBNET.Information.Err().Number ==
   (int)TLI.TliErrors.tliErrCantLoadLibrary)
{
   VBNET.Information.Err().Clear();
   libname = "-";
   libguid = "-";
}
else
{
   // OnErrorGoto(ErrorHandler);
   libname = tlinfo.Name;
   libguid = tlinfo.GUID;
}
```

</td></tr>
<tr><td colspan="2">This translation occurs in ScanToolLib.clsCOMReporter.cs</td></tr>
</table>

VB.NET is much more backward compatible with VB6, and gmBasic's VB.NET translations take advantage of this. For example, VB.NET supports the notorious On Error Resume Next, so gmBasic's treatment of error handling for VB.NET translations is much more direct.

Control Arrays to Arrays of Controls

In VB6 you can layout a form with multiple instances of a same-named control. This is referred to as a control array and it is a very popular VB language feature. The .NET framework has generalized the concept of control arrays with a new event model. You will also find ControlArray classes in the .NET framework that provide backward compatibility, and the VB Upgrade Wizard uses these ControlArray classes. However, gmBasic translates control arrays into (strongly typed) Generic Collections as we believe this provides more flexible and maintainable code. See the translation of the radio buttons on ScanToolUI.frmScanTool.cs for an example of this.

Late Binding

VB6 supported late binding implicitly, but in .NET late bound calls require a CallByName. gmBasic is able to correctly author CallByName statements.

<table>
<tr><td>

```
VB6
Dim filHandler As Object

filHandler.processFile(fil, rpt)
```

</td><td>

```
VB.NET
Dim filHandler As Object

CallByName(filHandler, "processFile",
       CallType.Method, New Object() {fil, rpt})
```

</td></tr>
<tr><td></td><td>

```
C#
object filHandler = null;

VBNET.Interaction.CallByName(filHandler,"processFile",
       VBNET.CallType.Method,new object[]{fil,rpt});
```

</td></tr>
</table>

Weak to Strong Typing

VB6 has an extremely flexible type system; it can pretty much convert any type into any other type implicitly. In addition, it offers a Variant type that are frequently used and sometimes abused. Incorrectly typed variables are also common. On the other hand, the .NET languages are designed for very explicit typing and they are much stricter in terms of which conversions are allowed. Dealing with the change from weak to strong typing can make manually migrating VB code a nightmare. Once again, this is a problem that gmBasic is specifically designed to solve and it can almost always determine the type of a variable by how it is used, and it uses this information to author the correct declaration and type casts automatically.

Other Features

Source Code Analytics

One of the more daunting tasks in a migration is mapping legacy APIs to .NET APIs. For example, both ADODB and ADO.NET have hundreds of members. Knowing how you use a legacy API will let you narrow your focus to the API members that you actually use. This is much more efficient than trying to map the entire API.

gmBasic's Symbol Reports help you understand how you actually use APIs as well as helping to reveal the detailed structure of your programs. Symbol Reports show you, in extreme detail, the types and methods you are defining and using in your codebase. This facility goes well beyond simply looking at explicit Reference and Object statements in the VBP files. Symbol reports show which specific members are used, where they are used, and how they are used. In addition, Symbol reports show you how many times a symbol is actually referenced.

Each record in a symbol report includes the following fields:

Field	Description
Record Type	Indicates if the record describes a definition of, or referenced to, a symbol (DEF or REF)
Member Name	The name of the symbol being described
Member Class	The class of the type being described
Member Library	The library of the type being described
Member Type	The Type of the symbol (Class, Enum, Property, Sub, Function, Event)
Loc Line	If Record Type=DEF, the number of references to the symbol If Record Type=REF, the line number in the source where this reference occurred
Loc Member	If Record Type=DEF, the name of the member defined If Record Type=REF, the name of the subprogram where this reference occurred
Loc Text	If Record Type=DEF, details about the definition of the symbol If Record Type=REF, the text of the source line where this reference occurred
Loc Path	If Record Type=DEF, path to the file that contains the definition of the symbol If Record Type=REF, path to the file that contains this reference of the symbol
Loc Name	If Record Type=DEF, name of the type that contains the definition of the symbol If Record Type=REF, name to the type that contains this reference of the symbol
Loc Type	If Record Type=DEF, type of the file that contains the definition If Record Type=REF, type of the file contains this reference of the symbol

These reports are useful for analyzing your codebase and planning a migration.

Build Order Report

In order to do a SmartPort or a CleanPort, it is necessary to translate the system in build order – this means translating lower level components before their clients. Determining build order can be a challenge for some organizations, so gmBasic provides a facility for determining the build order of an arbitrary collection of VBPs.

Translation Control Scripts

In addition to the simple command-line translations shown so far, gmBasic offers a rich job control language that can be used to provide special handling for your translations. The job control commands can be specified at a global level or at the translation job level.

Dealing with "Bad Code"

VB6 and ASP are extremely forgiving development platforms and coding errors can go undetected. In addition, the extremely weak typing of VBScript used in ASP means there is a shortage of information in the code itself. The bottom line is "bad code happens", so few VB6/ASP projects translate perfectly the first time through. Fortunately, gmBasic was designed to help you deal with this in a systematic and automated way. When we identify problems in the source code, we can use our "corrective" job control commands to provide additional information to gmBasic so it can produce clean translations. Some of the corrective commands include:

Command	Description
Replace	In rare cases, a block of VB6 code that is a bit "too creative" or archaic needs to be changed in

<table>
<tr><td>(Pre-Edit)</td><td>order to facilitate a clean translation. The Pre-Edit facility is design to support making changes to the source code as it is translated. Pre-edits can search and replace code, delete statements, or comment things out. A Pre-Edit does not change the existing VB code; rather it changes the VB6 code in memory after it is read from disk. This approach simplifies VB source control, it is automated and repeatable, and it also ensures that there is a record of the change.</td></tr>
<tr><td>FixType</td><td>VBScript has no intrinsic typing and VB6 has an extremely flexible type system; they both can pretty much convert any type into any other type implicitly. In addition, it offers a Variant type that programmers frequently use and sometimes abuse. Incorrectly typed variables are fairly common. On the other hand, the .NET languages require the programmer to be explicit in request typing conversions and it is much stricter in terms of which conversions are allowed.

gmBasic can almost always determine the correct type of a variable by how it is used and author the correct declaration and type casts automatically. In the rare case where it cannot determine the correct type, we can select a specific type or specify ByRef/ByVal using the fixtype command.</td></tr>
<tr><td>Reauthor</td><td>If you want to "completely rewrite" a specific function you can use the Reauthor command. This command causes gmBasic to insert your handcrafted code when it authors the specified function.

```
<Reauthor subprogram="PromisScan.GetFileVersionInformation" ><![CDATA[
        public static VersionReturnValue GetFileVersionInformation(string pstrFileName,
               FILEINFO tFileInfo)
        {
           try
           {
              System.Diagnostics.FileVersionInfo fvi =
               System.Diagnostics.FileVersionInfo.GetVersionInfo(pstrFileName);

              tFileInfo.CompanyName = fvi.CompanyName;
              tFileInfo.FileDescription = fvi.FileDescription;
              tFileInfo.FileVersion = fvi.FileVersion;
              tFileInfo.InternalName = fvi.InternalName;
              tFileInfo.LegalCopyright = fvi.LegalCopyright;
              tFileInfo.OriginalFileName = fvi.OriginalFilename;
              tFileInfo.ProductName = fvi.ProductName;
              tFileInfo.ProductVersion = fvi.ProductVersion;

              return VersionReturnValue.eOK;
           }
           catch
           {
              return VersionReturnValue.eNoVersion;
           }
        }
]]></Reauthor>
```

Using Reauthor ensures that the fix occurs every time the translation is done.</td></tr>
<tr><td>Replace
(Post-Edit)</td><td>Before authoring the final .NET code, you have the opportunity to apply search and replace type edits. This facility has several benefits:

A simple way to document and resolve open translation issues.
A way to solve one-off problems
A temporary workaround to handle issues until a permanent solution can be incorporated into the formal tool configuration or the system tools.
</td></tr>
</table>

Appendix A: Great Migrations Methodology

The ultimate objective of this migration project is to produce a correct and maintainable version of the codebase on the .NET platform -- within a certain timeframe and a reasonable budget and subject to the migration team's code quality requirements. Figure A1 shows that in our methodology, the process of producing a clean, correct, and standards conformant code is a highly iterative one with four major cycles:

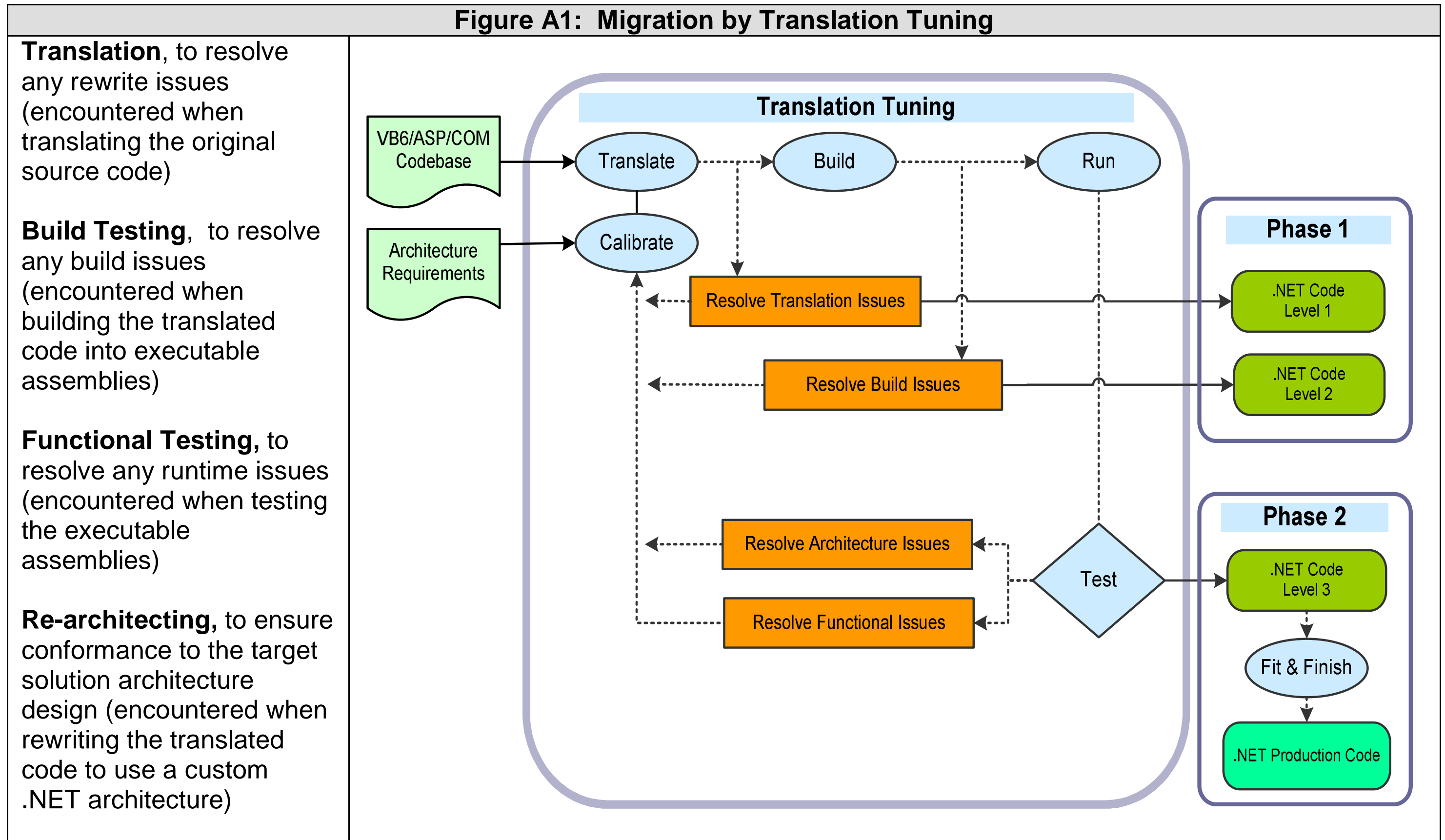
Figure A1: Migration by Translation Tuning

Translation, to resolve any rewrite issues (encountered when translating the original source code)

Build Testing, to resolve any build issues (encountered when building the translated code into executable assemblies)

Functional Testing, to resolve any runtime issues (encountered when testing the executable assemblies)

Re-architecting, to ensure conformance to the target solution architecture design (encountered when rewriting the translated code to use a custom .NET architecture)



Tool-Assisted

To facilitate this approach we leverage **gmBasic,** our proprietary software re-engineering technology. gmBasic is a proven toolset which facilitates a systematic, repeatable and improvable transformation of VB6 code to one of the .NET languages (either VB.NET or C#.NET). gmBasic also supports the re-structuring of VB6/COM applications to the .NET Framework and it provides extremely detailed information about the entire codebase that can guide the implementation of improved and/or additional functionality. gmBasic also translates ASP classic code to ASP.NET code. gmBasic is extremely fast: in just a few minutes, it can rewrite hundreds of thousands of lines of VB6/ASP code to thousands of lines of correct .NET code (C#.NET, VB.NET or ASP.NET) -- a result which could take an expert .NET developer years to achieve. Furthermore, these automated transformations can be systematically improved and customized based on the feedback and requirements of the migration team – this includes automating, either fully or partially, the resolution of both functional and architectural issues.

Iterative

In our methodology, the migration of an application to the .NET platform is done via a sequence of tuning cycles – leading to a final Fit & Finish task to finish the code to production. Each tuning cycle produces code of increasing quality by resolving any issues encountered during the cycle – leading up to a "Cut-Over" point where the only issues left are those that were identified as being easier (i.e., more economical) to do by hand. The end result is a highly efficient conversion process that balances manual work with automated re-engineering.

As we move through the tuning cycles of the migration diagram in Figure A1, we are producing code of increasing quality for the target platform. The three levels of code quality that we expect to achieve in this process are shown in Table A1.

Table A1: Code Quality Index	
Level 1: Translation Complete	All statements/constructs of the source code translate into target code statements/constructs with no rewrite errors and the resulting code meets generally accepted standards of well-formed syntax on the target platform. The resulting target projects are well formed and load cleanly in the Visual Studio .NET platform for further development on that platform.
Level 2: Build Complete	All statements/constructs of the target code compile with no compiler errors. In addition, specialized project elements (e.g., Forms) are accessible through their specialized editors provided by the target development environment.
Level 3: Verification & Upgrade Complete	The target application yields correct results on the target platform. The functionality of the target application matches that of the source application. The application is ready for deployment and further upgrade on the .NET platform. The functionality of the target application has been enhanced with new features and upgrades of existing features. The application is ready for production use on the .NET platform.

Test-Driven

The approach is iterative and test-driven and the level of effort required to do a migration depends on the number of issues found and resolved during each cycle of the tuning process. Most of our work is in customization ("tool calibration") subject to the migration team's re-architecting requirements and acceptance criteria. Most of the the migration team effort will be in specifying a target architecture and in validating the .NET codebase by manual and/or automated code review and functional testing.

No-Code Freeze

We assume a working source codebase, but we do not require a legacy code-freeze until the migration team is satisfied with the conversion process and ready to do a final translation. Up to that point, the legacy application is allowed to be maintained and enhanced for ongoing maintenance. We take the updated versions of the VB6/COM codebase into the translation refinement process as they become available – typically on the same release cycle as the legacy application.

Measurable

The use of semi-automated translation, build, and code-review procedures along with test-driven refinement of the rewrite process facilitates an ongoing collection of metrics relating to code quality and work effort. Migration progress is scalable, incremental, and easy to track.

Repeatable and Documented

Driven by a detailed XML-based configuration, gmBasic is a tool that can be calibrated to produce a variety of .NET outputs. Furthermore, the configuration files and translation scripts provide a precisely documented record of the rewrite rules and other information that make the system transformation repeatable.

Balanced Application of Automated and Manual Development

The up-front investment in the rewrite process ensures a predictable, efficient, and effective transition from the old to the new platform; however, we certainly do not expect the toolset to do everything. The migration process illustrated in Figure 2 includes tasks performed automatically by tool and tasks performed by the migration team – a group of skilled software engineers from GM and migration team working in tandem to run the tools, analyze the results, identify issues, resolve issues, run tests, verify results, and deploy the final .NET code to production on the target platform. As a migration partner, our goal is to help the migration team strike an optimal balance between project scope, automated rewrite and manual development. Our approach frees up funds to focus on non-automatable tasks such as design, development process improvement, quality assurance, and training.

Appendix B: Navigating the Sample Files

The GM Technology Sample files are organized into to a subdirectory with the following folder structure:

```
PROMIS>
└───demo
    ├───FMStocks
    │   ├───FMSlib
    │   │   ├───FMSStore_Bus_std_csh
    │   │   ├───FMSStore_Bus_std_vbn
    │   │   ├───FMSStore_Bus_VB6
    │   │   ├───FMSStore_DB_std_csh
    │   │   ├───FMSStore_DB_std_vbn
    │   │   ├───FMSStore_DB_VB6
    │   │   ├───FMSStore_Events_std_csh
    │   │   ├───FMSStore_Events_std_vbn
    │   │   ├───FMSStore_Events_VB6
    │   │   ├───FMSTest_std_csh
    │   │   ├───FMSTest_std_vbn
    │   │   ├───FMSTest_VB6
    │   │   ├───FMStocks_Bus_std_csh
    │   │   ├───FMStocks_Bus_std_vbn
    │   │   ├───FMStocks_Bus_VB6
    │   │   ├───FMStocks_DB_std_csh
    │   │   ├───FMStocks_DB_std_vbn
    │   │   ├───FMStocks_DB_VB6
    │   │   ├───FMStocks_Ext_std_csh
    │   │   ├───FMStocks_Ext_std_vbn
    │   │   └───FMStocks_Ext_VB6
    │   └───FMSweb
    │       ├───ASP
    │       ├───CSH
    │       └───VBN
    │
    └───ScanTool
        ├───ScanToolLib_managed_csh
        ├───ScanToolLib_managed_vbn
        ├───ScanToolLib_std_csh
        ├───ScanToolLib_std_vbn
        ├───ScanToolLib_VB6
        ├───ScanToolUI_managed_csh
        ├───ScanToolUI_managed_vbn
        ├───ScanToolUI_std_csh
        ├───ScanToolUI_std_vbn
        └───ScanToolUI_VB6
```

`demo\FMStocks` contains the translations for the FMStocks Sample

- `FMSLib` contains the data access and business logic components used by the FMStocks web site. There are folders for each VBP.
  - `VB6` – original VB6 codes
  - `std_csh` – C# translations / Interop
  - `std_vbn` – VB.NET translations / Interop
- `FMSWeb` is the ASP sample. There is a folder for each version of the web site.
  - `ASP` – original ASP/VBScript files
  - `CSH` – C# translations
  - `VBN` – VB.NET translations

`demo\ScanTool` contains the translations for the ScanTool sample.

- `std_csh` – C# translations / Interop
- `std_vbn` – VB.NET translations / Interop
- `managed_csh` – C# translations / managed
- `managed_vbn` – VB.Net translations / managed