Component Replacement Using gmStudio

Great Migrations LLC
June 2014

Glossary

GM	Great Migrations LLC
gmStudio	GM's Migration Development Platform
gmBasic	gmStudio's Language Translation Engine (VB6 to C#, VB6 to VB.NET, ASP to ASP.NET)
IDF	Interface Description File
CIDF	Custom Interface Description File modified to direct COM replacement
SRP	Semantic Reference Report
TCF	Translation Configuration File (xml file containing translation rules for gmBasic)
Translation Script	Synonym for TCF
FileExplorer	One of the VB6 sample codes distributed with gmStudio (single-project code)
ScanTool	One of the VB6 sample codes distributed with gmStudio (two-project code)
FMstocks	One of the VB6/ASP sample codes distributed with gmStudio (multi-project code)

Planning and Estimating Component Replacement

A common and important objective in VB6/ASP/COM to .NET reengineering projects is to reduce or eliminate system dependence on external COM components (API libraries and ActiveX controls). The Great Migrations Methodology accomplishes this by applying the following four-step process:

1) Understand the services legacy components provide to your application
2) Define your COM replacement strategy
3) Design and implement the rules to migrate from COM components to .NET components
4) Apply, refine and extend rules to additional source codes as needed

These four tasks are described in more detail below.

1) Understand the services legacy components provide to your application

COM components provide many types of services to the applications that use them. A few examples are listed here:

- User interface controls
- Data access
- File, directory, and data management
- Communications – wire protocols, email, faxing, FTP
- Document automation – Excel, Word, PDF
- Graphics and Multimedia
- Document generation and reporting
- Interaction with peripheral devices

COM API, file	Service(s)	Possible .NET Replacement(s)
Scripting, scrrun.dll	Dictionary Basic File/Directory Manipulation	Migrate to System.Collections (or Generics) System.IO
ADODB, msado27.tlb	Data Access	System.Data
MSXML2, msxml6.dll	DOM Access XMLHTTP Messaging	System.Xml System.Web.HttpRequest

In practice, external services are consumed in terms of fine-grained operations initiated by setting properties, invoking methods, and responding to events from API classes. In order to flesh out a replacement strategy, you need a detailed inventory of the COM dependencies at the type-member level. Fortunately, gmStudio can generate this very detailed COM dependency information for each translation unit in its "**Semantic References Report**". Each record in this report describes a reference to a symbol defined either in your code or in a COM component referenced by your code. The fields of this report are described in Table 1 and Figure 1 below.

Table 1: Semantic Reference Report Record Structure	
Field ' Description	**Sample (Comments)**
SrcName ' Migration Unit Name	CCONV (the internal name of the migration unit)
RecType ' Report Type	'REF' (constant)
MemName ' Referenced Member Name	Enabled (name of element being referenced)
MemClas ' Referenced Member Parent	ISSBase (class of element being referenced)
MemLibr ' Referenced Member Library	Threed (library of element being referenced)
MemType ' Referenced Member Type	Lib_Property (type of element being referenced)
Pos_Cnt ' Referring Location (line #)	971 (approximate)
LocText ' Referring Source Code	If cmdSave.Enabled Then cmdSave.SetFocus
LocMemb ' Referring Member	accountid_KeyPress (source sub or function)
LocFile ' Referring File Path	C:\gmSrc\APP\R5.0\SOURCE\CCONV\ACCOUNTL.FRM
LocName ' Referring Type Name	AccountList (source class)
LocType ' Referring File Type	FRM (source file type)

Figure 1: Consolidated and Cross-Tabulated Reference Report Data

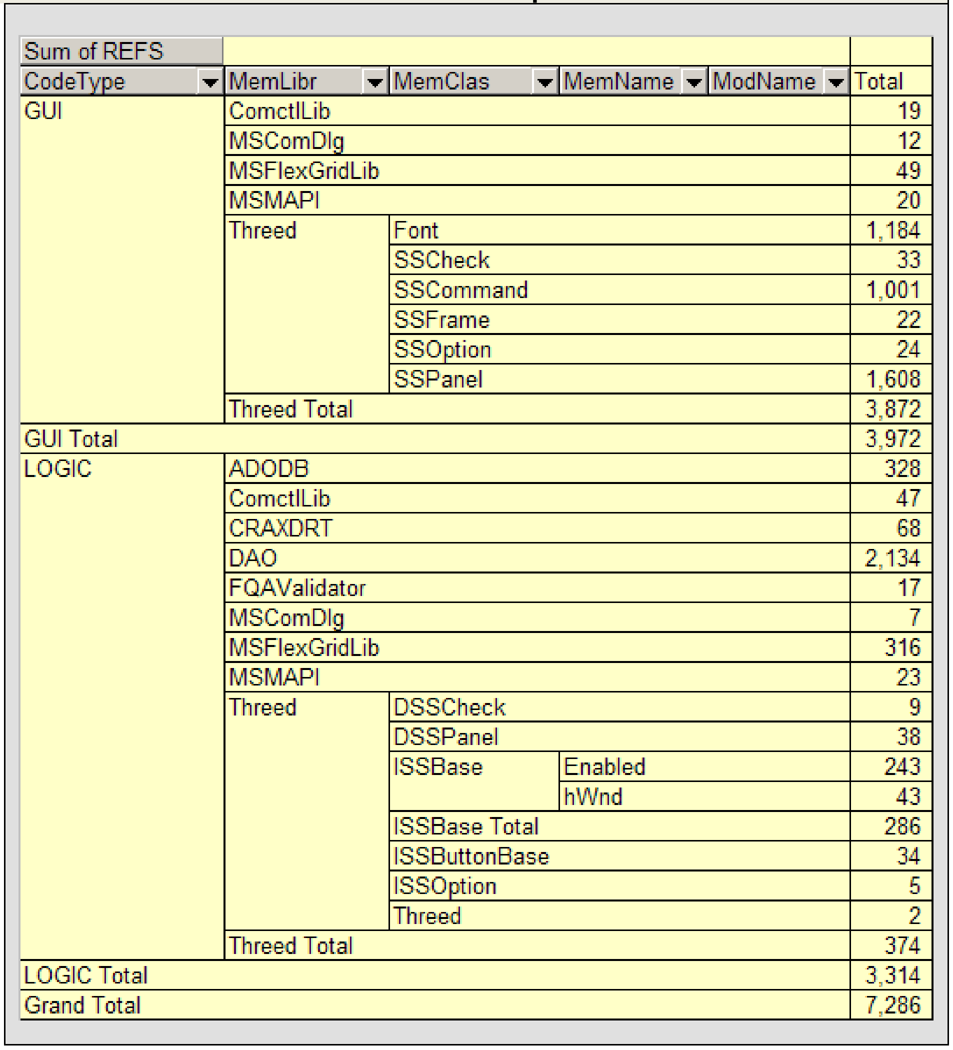
A Semantic Reference Report (SRP) is a tab-delimited file. For a large application, this report will contain hundreds of thousands, sometimes millions, of records. When the analytics report is very large, you may wish to load it into a relational database, consolidate and filter the results, and export it to a spreadsheet program for analysis.

The SRP also describes internal references that a codebase makes to its own symbols (variables, subprograms, properties, enum entries, etc.). Self-references have MemLibr=LocName.

The detailed quantitative data on component usage helps you fully measure, understand, and plan the component replacement efforts.

A pivot table created from typical SPR data is shown in Figure 1.

Note: Control references in the designer code can be gathered using the “Source GUI Scan Report” and merged with the Semantic References Report.



Sum of REFS					
CodeType	MemLibr	MemClas	MemName	ModName	Total
GUI	ComctlLib				19
	MSComDlg				12
	MSFlexGridLib				49
	MSMAPI				20
	Threed	Font			1,184
		SSCheck			33
		SSCommand			1,001
		SSFrame			22
		SSOption			24
		SSPanel			1,608
	Threed Total				3,872
GUI Total					3,972
LOGIC	ADODB				328
	ComctlLib				47
	CRAXDRT				68
	DAO				2,134
	FQAValidator				17
	MSComDlg				7
	MSFlexGridLib				316
	MSMAPI				23
	Threed	DSSCheck			9
		DSSPanel			38
		ISSBase	Enabled		243
			hWnd		43
		ISSBase Total			286
		ISSButtonBase			34
		ISSOption			5
		Threed			2
	Threed Total				374
LOGIC Total					3,314
Grand Total					7,286

2) Define your COM replacement strategy

The second step in the COM replacement process is selecting new components that will replace the legacy ones. The three most common strategies for component replacement are listed here:

- Use a New .NET component
- Use the Legacy Component via Interop
- Use a Custom Wrapper Assembly

A fourth option is to completely remove the functionality that depends on the legacy component. While this is an important option to consider in special cases, it is not discussed further here.

Use a new .NET component

If you are already using .NET, you may already have defined coding and component usage standards and you will probably want to use those in the migrated code as well. .NET components may be built, bought from third parties, or utilized from Open Source. In addition, there is an extensive array of components available in the .NET Framework. Although the out-of-the-box .NET framework components and controls are very rich, some UI controls must be extended to match certain capabilities of third-party ActiveX controls.

If you can find a new .NET component that is very similar to the legacy component, it will be fairly straightforward to pick up the new component and start using it in your .NET code. If you choose a new .NET component that is very different from the component it replaces, you will face a steep learning curve as well as significant challenges in redesigning legacy code.

Key Point	Custom, in-house components almost always fall under this scenario. They will be upgraded to .NET essentially replacing the VB6 component with a newly migrated .NET replacement, or they will be replaced with a new .NET component.

Use the legacy component via a generic Interop assembly

Another strategy for dealing with COM dependencies is to use Interop to access the legacy component. COM Interop is **not** a replacement strategy: your .NET application will still depend on the original COM component which must still be properly deployed and registered in your development, build, and runtime environments. Interop also means you must "import" the COM library into .NET to create a Runtime Callable Wrapper (RCW) that encapsulates the COM library and exposes its members to the .NET application. Your .NET assembly will reference the RCW instead of the original COM component, so the RCW must also be deployed to your development, build, and runtime environments.

Beware, a few legacy components that offer specialized functionality not provided in .NET can only be accessed through Interop. At the time of this writing, Microsoft Office is a notable example of this.

Key Point	In some cases you cannot use an Interop component – for example, many ActiveX controls that interface aggressively with the VB6 forms engine (i.e., containers, resizers, elastics, etc.) will cause the .NET forms engine to abort with an exception. In these cases, the control **must** be migrated to a .NET replacement. In addition, sometimes COM components cannot be completely imported for Interop usage.

Although COM Interop may be useful as an intermediate state during the translation tuning effort, it is almost never the desired end state. In general, migration teams want to minimize use of Interop because it creates challenges in terms of security, features, ease of development, debugging, integration, performance, deprecation, and community support.

Use a custom wrapper

If you are doing a migration with gmStudio, there is a third option. At your option, gmStudio will replace the legacy component with a Prototype Assembly. Prototype Assemblies are 100% .NET components that expose the same interface as the legacy components. They help bridge the gap between your application logic and new components. Prototype Assemblies allow you to get the migrated application to a build-complete state quickly and provide a runtime framework that is consistent with the legacy application. Another advantage of Prototype Assemblies is that they insulate your application from third party components by creating an architectural layer to manage and extend those interfaces. Insulating your application code from third-party component details is universally recognized as a useful technique for managing the impact of technology change over time.

Key Point	Prototype Assemblies begin as stubs – they have the same interface as the original component, but they lack an implementation. They are only a starting point. Extending a prototype through inheritance from or encapsulation of one or more appropriate .NET classes or by leveraging the Windows APIs via P/Invoke can accelerate the implementation of a full-featured custom wrapper. P/Invoke (Platform Invocation Services) allows managed code to call unmanaged functions that are implemented in an entry-point DLL.

3) Design and implement the rules to migrate from COM to .NET

Designing the rules for using .NET components is a manual task -- regardless of whether you plan to use tools to help you rewrite the code or not. The rules must be considered and implemented at a detailed level based on the specific COM classes and members that are used by your application. In our methodology, the rules that direct COM replacement are directed by two types of files: Custom Interface Description Files, gmSL scripts, and Migration DLLs.

Custom Interface Description Files (CIDFs) are XML documents that describe how COM object models map to one or more replacement classes in .NET. Recall that an IDF (Interface Description File) is an XML document that describes a COM component and helps gmStudio interpret source code that uses it. gmStudio automatically generates standard IDFs for the COM components referenced by your source code. The easiest way to create a custom IDF is to copy the standard IDF and then modify it by hand: adding component replacement rules incrementally for the API elements you want to replace. The file naming convention for an IDF is: <COM File Name >.xml; for a custom IDF the notation is: GM.<COM File Name >.xml. A number of Custom IDF samples may be found in the [installation_directory]\support\rules folder of the gmStudio distribution.

gmSL scripts are text files containing Great Migrations Scripting Language (gmSL) routines that implement advanced code analysis and transformation features. These routines can be attached to specific COM API elements (e.g. Properties, Methods, Events). When gmBasic is about to analyze or rewrite application code that uses the specific COM API element, it will invoke the gmSL routine and pass in arguments that facilitate inspection, analysis, and transformation at the right place in the code. gmSL also has a collection of powerful service classes that simplify interacting with gmBasic and with the upgrade information model. gmSL routines allow you to make your transformation process very sophisticated and dynamic. You can add code, remove code, and manipulate the code stream with great precision.

Migration DLLs are very similar to gmSL scripts, but they are implemented in C, Managed C++, C#, or VB.NET using Great Migrations Native Interface (gmNI). Migration DLLs allow you to implement advanced, custom migration like gmSL, they must be precompiled and linked with the gmBasic service library. Migration DLLs can operate faster than gmSL and they are distributed as binary files.

More details on how to implement COM migration rules are discussed in the appendices to this document and also in the gmStudio Samples and in the gmStudio User Guide.

4) Apply, refine and extend rules for additional source codes as appropriate

A typical VB6/COM source code uses only portions of the COM components they reference. Fortunately, the migration rules for a given COM component only need to address the parts that your source code actually uses. If you are migrating a large codebase you can expect to incrementally modify your custom IDFs as you work on each successive source code unit (VBP/ASP file). However, the number of additions should diminish to zero once you have processed a representative sampling of your code.

Balancing manual and tool-assisted techniques

Of course there are limitations to what should be automated in a software reengineering effort. For example, it may not make sense to automate a particularly complex set of reengineering operations that only impact one application function. In these cases, the migration team should consider the following balanced approach:

1) Leveraging gmBasic’s analytic capabilities, find out where and how the COM component is used in order to scope the redesign work and to plan component replacement tasks. gmBasic is the translation engine of gmStudio.

2) If appropriate, use gmStudio to implement part of the migration

3) Finish and refine the transformation work manually

4) If desired, configure gmStudio to integrate hand-written code into your migration script solution to make the manual reengineering a repeatable operation.

Appendix A: Upgrading Msxml6.dll to System.Xml

The following section offers more detail on how COM replacement is approached with gmStudio. This discussion applies to ScanTool, one of the sample codes distributed with gmStudio. ScanTool uses a number of COM APIs. One of them is Microsoft's XML API (MSXML2) typically found in one of the following files: **MSXML2.dll, MSXML4.dll,** or **MSXML6.dll**.

1) Understand the services legacy components provide to your application

A portion of the Semantic References Report for the ScanTool sample and the MSXML2 API is shown in Table A1. It shows that ScanTool references 11 different elements of the MSXML2 library. A cursory analysis shows that MSXML2 is used for loading and saving XML files and manipulating DOM objects.

Table A1: Cross-Tabulated MSXML2 Library Reference Report for ScanTool Sample

MemLibr	MSXML2			
Count of MemName				
MemClas	**MemType**	**MemName**	**LocText**	**Total**
IXMLDOMDocument	**Lib_Method**	createElement	Set elmNew = elmRoot.ownerDocument.createElement(name)	1
			Set elmRoot = domConfig.createElement(CFG_XMLDOC)	1
		createTextNode	elmNew.appendChild elmNew.ownerDocument.createTextNode(value)	1
		load	If domConfig.Load(filename) = False Then	1
		save	Call domConfig.save(filename)	1
IXMLDOMElement	**Lib_Method**	setAttribute	elmParam.setAttribute CFG_IGNORE_CASE, IIf(chkNoCase, CFG_IGNORE	1
IXMLDOMNode	**Lib_Method**	appendChild	domConfig.appendChild elmRoot	1
			elmNew.appendChild elmNew.ownerDocument.createTextNode(value)	1
			elmRoot.appendChild elmNew	1
		selectSingleNode	Set nod = dom.selectSingleNode(xpath)	1
	Lib_Property	firstChild	If nod.firstChild Is Nothing Then	1
		nodeValue	readNode = nod.firstChild.nodeValue	1
		xml	Debug.Print domConfig.xml	2
MSXML2	**Coclass**	DOMDocument	Set domConfig = New DOMDocument	2
Grand Total				**16**

2) Define your COM replacement strategy

In .NET, all XML file and DOM object manipulation done in the ScanTool sample can be handled by classes and functions in the System.Xml namespace. Table A2 presents some examples of the services provided by MSXML2 and the strategies for using System.Xml as an alternative means of getting these services.

Table A2: High-level Redesign Strategies

Legacy API Service	**Replacement Strategy**
Reference to MSXML6.dll	Replace with System.Xml namespace. This namespace is in the mscorlib assembly and is available by default in .NET project files generated by gmStudio.
Load a DOM	The MSXML2.IXMLDOMDocument has Load and LoadXML methods that can initialize a DOMDocument object from a file or XML string respectively. These APIs return a boolean to indicate success (true) or error (false), and if an error occurs, there is a DOMError object that contains the details about the error. The System.Xml API offers similar methods, but they throw exceptions to indicate an error. In .NET the resulting exception object contains the details about the error.
Save a DOM	Saving a DOM to a text file is accomplished with a save method in both APIs.
Accessing a DOM	In general the most common Xml operations are done by invoking like-named members, although the casing of class and member names has changed. The .NET members and their arguments also use stronger typing.

Key Point	The rules that direct the transformation of ScanTool code to use System.Xml instead of MSXML2 are, for the most part, relatively simple because the two APIs are very similar. Consequently, all the rules can be specified in a custom IDF; neither a gmSL script nor a Migration DLL is needed.

3) Design and implement the rules to migrate from COM to .NET

The rules for replacing MSXML2 elements with System.Xml are specified in a custom IDF, which you write by hand by simply editing the MSXML2 IDF as generated by gmStudio. In practice, you do the following:

- Copy the standard IDF (a text file, “msxml6.dll.xml”, generated by gmStudio from the binary file “msxml6.dll”) from the .\IDF\FromIDL folder into the .\usr folder and rename it “GM.msxml6.dll.xml”. Note, this is just a naming convention, but it helps trace the Custom IDF back to its original source IDF.
- Add a registry-libname statement to the Translation Script to tell gmStudio that you want to use the Custom IDF instead of the standard one.

```
<Registry type="libname" source="msxml6.dll" target="GM.msxml6.dll" />
```

The following sections explain how to customize the Custom IDF to specify the rules that will migrate code using MSXML2 to code using System.Xml.

Removing reference to msxml6.dll and replacing MSXML2 with System.Xml

Every IDF contains a library node that specifies how to author references to the API assembly in your .NET project. For standard (prototype) and refactoring translations, this is specified as follows.

Standard Interface Description File, msxml6.dll.xml	Custom Interface Description File, GM.msxml6.dll.xml

`<DescriptionFile>`
`<library id="msxml6.dll"`
 	
`location="%library%\Interop.MSXML2.dll"`
`         migName="MSXML2"`
`         type="LocalImport"`
`         ...`
`>`
`</DescriptionFile>`
	`<DescriptionFile>`
	`<library id="GM.msxml6.dll"`
	`         location="DoNotDeclare"`
	`         migName="System.Xml"`
	`         type="Native"`
	`         ...`
	`>`
	`</DescriptionFile>`

The library node tells gmBasic that .NET projects that use the MSXML2 object model should use a namespace called “MSXML2”. If you use the Interop strategy, your code will find the implementation of this namespace in an Interop assembly called Interop.MSXML2.dll. The meta variable %library% is set in the translation script and defaults to the Interop folder in the migration workspace. If you use the prototype strategy, the implementation of this namespace will be a stub class generated by gmStudio	The id is changed to match the containing file name without the .xml extension. location=“DoNotDeclare” tells gmBasic not to author a reference in the .NET project file. System.Xml is part of the System.XML.dll assembly that is in the default .NET project file template used by gmStudio, so no other declaration is needed. In the library element, migName=”System.Xml” tells gmBasic that the namespace MSXML2 is replaced with System.Xml. type="Native" tells gmBasic to author the reference in .NET style; this is moot here since DoNotDeclare means no reference will be authored, but it serves to provide documentation.

Replacing MSXML types with System.Xml types

A standard IDF file contains class and coclass elements that describe the API exposed by the COM component. Adding a migName attribute to any of these nodes tells gmBasic that the .NET replacement element has a different name, as shown in the examples below.

`...`
`<class id="IXMLDOMNode" parent="IDispatch"`
`       creatable="off">`
`...`
`<coclass id="DOMDocument">`
	`...`
	`<class id="IXMLDOMNode" parent="IDispatch"`
	`       creatable="off" migName="XmlNode">`
	`...`
	`<coclass id="DOMDocument" migName="XmlDocument">`

```
...
<coclass id="XMLHTTP">
```

```
...
<coclass id="XMLHTTP"
migName="System.Net.HttpWebRequest"
         migStatus="external">
```

Note that in some situations, the replacement class is in a different namespace from the one specified in the library node. In these cases you must fully qualify the replacement class name and add migStatus="external" as in the XMLHTTP example above.

Replacing an Enum

An IDF contains information about the enums and their entries. Both enum type names and entry names can be migrated using the migName attribute.

```
...
<enumeration id="tagDOMNodeType">
 <entry id="NODE_INVALID" value="0"/>
 <entry id="NODE_ELEMENT" value="1"/>
 <entry id="NODE_ATTRIBUTE" value="2"/>
 <entry id="NODE_TEXT" value="3"/>
 <entry id="NODE_CDATA_SECTION" value="4"/>
 <entry id="NODE_ENTITY_REFERENCE" value="5"/>
 <entry id="NODE_ENTITY" value="6"/>
 ...
</enumeration>
```

```
...
<enumeration id="tagDOMNodeType"
migName="XmlNodeType">
 <entry id="NODE_INVALID" value="0"/>
 <entry id="NODE_ELEMENT" value="1"
migName="Element"/>
 <entry id="NODE_ATTRIBUTE" value="2"
migName="Attribute"/>
 <entry id="NODE_TEXT" value="3" migName="Text"/>
 <entry id="NODE_CDATA_SECTION" value="4"
migName="CDATA"/>
 <entry id="NODE_ENTITY_REFERENCE" value="5"/>
 <entry id="NODE_ENTITY" value="6"/>
...
</enumeration>
```

Notice that not all of the enum entries have been mapped to replacements; references to the ones that have not been mapped will be unchanged in the resulting codes, but if the source code does not reference those entries then this is of no consequence.

Replacing Members

An IDF contains information about the properties, methods, accessors (parameterized properties), and events exposed by the classes of the COM type. These elements can be migrated by adding migration attributes and making other modifications to the member nodes in the Custom IDF. Some of these modifications are described below.

Renaming a Member

Adding a migName attribute to a member changes the name used to refer to that operation in translated codes.

```
...
<class id="IXMLDOMDocument"
parent="IXMLDOMNode">
...
   <property id="documentElement"
            type="IXMLDOMElement"
            status="InOut"/>

  ...
```

```
...
<class id="IXMLDOMDocument" parent="IXMLDOMNode">
...
  <property id="documentElement"
                   type="IXMLDOMElement"
                  status="InOut"
                 migName="DocumentElement"/>
  ...
```

Removing a member

Marking a member with migStatus="delete" means that the resulting translations will not contain any statements that reference the member. This can be handy if the default, implicit behavior of the .NET replacement is sufficient for the needs of the code being upgraded.

```
...
<class id="IXMLDOMDocument"
parent="IXMLDOMNode">
...
  <property id="async" type="Boolean"
```

```
...
<class id="IXMLDOMDocument" parent="IXMLDOMNode">
...
  <property id="async" type="Boolean"
          status="InOut"
```

```
             status="InOut"/>
  ...
```

```
             migStatus="delete"/>
  ...
```

Changing the type of a member

The type attribute of each member affects how it is handled in various operations such as assignments and parameter passing. Changing the type impacts how conversions are applied to expressions involving the member. In the case of nodeValue, both the type and name are changed.

```
...
<class id="IXMLDOMNode" parent="IDispatch"
       creatable="off">
  ...
  <property id="nodeValue" type="Variant"
             status="InOut"/>
...
```

```
...
<class id="IXMLDOMNode" parent="IDispatch"
       creatable="off" migName="XmlNode">
...
   <property id="nodeValue" type="String"
             status="InOut" migName="Value"/>
```

Changing the reference type of a member

Sometimes a member that was a method in COM becomes a property in .NET. For example, in MSXML2, IMXLDOMNode.hasChildNodes is a method. The replacement in .NET is a property. This change is indicated by changing the element type in the Custom IDF.

```
...
<class id="IXMLDOMNode" parent="IDispatch"
       creatable="off">
  ...
  <method id="hasChildNodes"
type="Boolean"/>
  ...
```

```
...
<class id="IXMLDOMNode" parent="IDispatch"
       creatable="off" migName="XmlNode">
  ...
  <Property id="hasChildNodes" type="Boolean"
             migName="HasChildNodes"/>
  ...
```

Changing the type of a member argument

An IDF contains information about the names, types, marshaling status, and optionality of member arguments. These elements may also be changed by adding or modifying the argument attributes. The example below changes the type of the second argument to the insertBefore method from Variant to IXMLDOMNode.

Standard Interface Description File, msxml6.dll.xml

```
...
<class id="IXMLDOMNode" parent="IDispatch"
       creatable="off">
  ...
  <method id="insertBefore"
type="IXMLDOMNode">

    <argument id="newChild"
type="IXMLDOMNode"
              status="ByVal"/>
    <argument id="refChild" type="Variant"
              status="ByVal"/>
  </method>
  ...
```

Custom Interface Description File, GM.msxml6.dll.xml

```
...
<class id="IXMLDOMNode" parent="IDispatch"
       creatable="off" migName="XmlNode">
  ...
  <method id="insertBefore" type="IXMLDOMNode"
          migName="InsertBefore">
    <argument id="newChild" type="IXMLDOMNode"
              status="ByVal"/>
    <argument id="refChild" type="IXMLDOMNode"
              status="ByVal"/>
  </method>
  ...
```

Changing the calling pattern for a member

The default calling pattern for a class member is object.member(arguments). This can be modified to reflect changes in the ordering of parameters and even the structure of the call by adding a pattern specification to the member.

For example, the MSXML2.IXMLDOMDocument has a Load method that can initialize a DOMDocument object from a file. These APIs return a boolean to indicate success (true) or error (false), and if an error occurs, there is a DOMError object that contains the details about the error. The System.Xml API offers a similar method, but it throws an exception to indicate an error. In .NET the resulting exception object contains the details about the error.

The following modifications to the IDF direct the translator to rewrite the calls to the load method:

```
...
<method id="load" type="Boolean">
```

```
...
<method id="load" type="Boolean"
```

`<argument id="xmlSource" type="Variant"`
`          status="ByVal"/>`
`</method>`
`...`
	`   nPram="2"`
	
	`cshPattern="try{\p%1d.Load(%2d);\q}catch{}\n%1d.HasChildNodes"`
	`   vbnPattern="%1d.Load(%2d)\n%1d.HasChildNodes">`
	`   <argument id="xmlSource" type="Variant"`
	`             status="ByVal"/>`
	`</method>`
	`...`

This tells the translator to replace calls to IXMLDOMDocument.load with a calls to System.Xml.XmlDocument.Load and also rework the call to use try-catch error handling.

nPram	is the number of parameters to pull off the 'string stack' when authoring the .NET code. The 'string stack' contains the pCode stream that represents the operations of your code. The top of the stack contains the expressions for the objects and arguments associated with the current operation.
cshPattern	is the desired 'surface form' for expressing the operation in C#. Notice that the cshPattern is parameterized. The notation %1d means author the expression associated with the object that will receive the call. %2d is the first and only argument to the call. If one pattern is valid for both C# and VB.NET, you can use migPattern instead.
vbnPattern	is the desired 'surface form' for expressing the operation in VB.NET.

Note that this is a special pattern that assumes the call was in an If statement to check for and handle the case when the call fails.

```
  If domConfig.Load(filename) = False Then
         Err.Raise 17, Source:=TypeName(Me), Description:="Unable to load XML"
  End If
```

The result of the new pattern on the translation is shown below:

```
  try { domConfig.Load(filename);} catch{ }
  if (!domConfig.HasChildNodes)
  {
       VBNET.Information.Err().Raise(17,this.GetType().Name,"Unable to load XML",null,null);
  }
```

The assumption that load is called within an If statement is not universally true, but it is true for ScanTool -- and it produced the desired result here. Other patterns would be more applicable to other situations, and these can be specified as needed.

Appendix B: Upgrading Scrrun.dll to System.IO

The following section offers more detail on how COM replacement is approached with gmStudio. This discussion applies to the ScanTool sample code. ScanTool uses a number of COM APIs. One of them is the popular Microsoft Scripting Runtime (**scrrun.dll)**. This appendix shows how code that uses Scripting can be migrated to System.IO by gmStudio

1) Understand the services legacy components provide to your application

A portion of the Semantic References report for the ScanTool sample and the Scripting API is shown in Table B1 It shows ScanTool references 23 different elements of the Scripting library a total of 92 times. A cursory analysis shows that Scripting is used for file IO and for inspecting the file system.

Table B1: Cross-Tabulated Scripting Library Reference Report for ScanTool Sample

MemLibr	Scripting			
Count of MemName				
MemClas	**MemType**	**MemName**	**LocText**	**Total**
IFile	**Lib_Property**	DateCreated	Call writeRpt(rpt, _	1
		DateLastModified	Call writeRpt(rpt, _	2
			LastModDate = fil.DateLastModified	2
		Name	Call writeRpt(rpt, _	4
			Dim filname As String: filname = fil.name	7
			Dim filname As String: filname = LCase$(fil.name)	1
			filextn = Mid$(fil.name, i + 1)	1
		Path	Call setStatus(fil.Path)	1
			If Err Then _	7
			Set tlinfo = tliApp.TypeLibInfoFromFile(fil.Path)	1
			Set txtfil = openInputText(fil.Path, fileSys)	6
		Size	Call writeRpt(rpt, _	1
			filSiz = filSiz + fil.Size	1
IFileSystem	**Lib_Method**	GetFile	Set fil = fileSys.GetFile(modpath)	2
		GetFolder	Set fld = fso.GetFolder(txtRootPath)	1
		OpenTextFile	Set openInputText = fileSys.OpenTextFile(sinpFileName, _	1
			Set openOutputText = fileSys.OpenTextFile(soutFileName, _	1
			Set txt = fso.OpenTextFile(logFile, IOMode:=ForAppending, Create:=True)	1
IFolder	**Lib_Property**	Files	For Each fil In fld.Files	1
		Path	Call writeRpt(rpt, _	1
			Dim filpath As String: filpath = fil.ParentFolder.Path	7
			Dim fldpath As String: fldpath = LCase$(fld.Path) & "\"	1
		SubFolders	For Each subfld In fld.SubFolders	1
IFolderCollection	**Lib_Property**	Count	Call writeRpt(rpt, _	1
IOMode	**Lib_EnumEntry**	ForAppending	Set txt = fso.OpenTextFile(logFile, IOMode:=ForAppending, Create:=True)	1
		ForReading	Set openInputText = fileSys.OpenTextFile(sinpFileName, _	1
		ForWriting	Set openOutputText = fileSys.OpenTextFile(soutFileName, _	1
ITextStream	**Lib_Method**	Close	If Not txtfil Is Nothing Then txtfil.Close	8
			rpt.Close	1
			txt.Close	1
		ReadAll	s = txtfil.ReadAll()	7
		ReadLine	s = txtfil.ReadLine	1
		Write	Call rpt.Write(str)	1
		WriteLine	Call rpt.WriteLine("PATH" & vbTab & "SUBFOLDERS" & vbTab & "FILES" & vbTab & "SIZE")	1
			Call rpt.WriteLine(filHandler.header)	1
			Call txt.WriteLine(rpt)	1
	Lib_Property	AtEndOfStream	Do While Not txtfil.AtEndOfStream	1
Scripting	**Coclass**	FileSystemObject	Set fileSys = New FileSystemObject	8
			Set fso = New Scripting.FileSystemObject	2
Tristate	**Lib_EnumEntry**	TristateFalse	Set openInputText = fileSys.OpenTextFile(sinpFileName, _	1
			Set openOutputText = fileSys.OpenTextFile(soutFileName, _	1
			Set txt = fso.OpenTextFile(logFile, IOMode:=ForAppending, Create:=True)	1
Grand Total				**92**

Note: Scripting also contains a Dictionary class that is not used by ScanTool. This class does not migrate to any class in System.IO; rather it becomes System.Collections.Specialized.OrderedDictionary; this will be shown for completeness even though it is not needed by ScanTool.

2) Define your COM replacement strategy

The reference report showed us that the Scripting library is being used for file IO and inspecting the file system. There are several ways to do this in .NET:

1) Use scrrun.dll through a runtime callable wrapper (aka., COM Interop)
2) Use My.VisualBasic.FileSystem and related classes
3) Use the classes in the System.IO namespace
4) Use a custom wrapper that you buy from a vendor, build yourself, or “borrow” from an open source.

In this example, we will be showing how to migrate to classes in the System.IO namespace.

Table B2 presents some examples of the redesign solutions needed to replace Scripting with System.IO as done for the ScanTool sample:

Table B2: High-level Redesign Strategies

Legacy API Service	**Replacement Strategy**
Reference to scrrun.dll	Replace with System.IO namespace. This namespace is in the mscorlib assembly and is typically available by default.
Manage The File System	The Scripting API has the FileSystemObject to inspect and manipulate the file system and open text files for reading and writing. The System.IO object model has no such class. These services are provided by a variety of different classes and objects. Logic that used the FileSystemObject instance will be modified to use these other objects as needed and the variable that held the FileSystemObject will be removed.
Accessing a text file	The Scripting API has the IFileSystem.OpenTextFile function for opening a text file. The arguments to this function specify how the file should be opened (for reading, writing, appending, create if not found, etc.). The function returns an TextStream that keeps track of its position and can be read or written depending on how it was opened. The System.IO design pattern is quite different: different types of streams are used for reading than writing. The design pattern requires that you instantiate an appropriate StreamReader or StreamWriter class based on what you plan to do with it. Determining the correct type of stream and setting up the constructor requires some analysis of the OpenTextFile arguments in the source code. If you want to do reading, then you instantiate a System.IO.File.StreamReader then use a variety of simple methods: Read, ReadLine, ReadToEnd to read content. If you want to do writing, then you instantiate a System.IO.File.StreamWriter then use use a variety of simple methods: Write, WriteLine to write content.
Close a text file	The Scripting and System.IO libraries both simply call the Close method of a given stream to close it.
Get the attributes of a file	The FileInfo object offers several properties that provide information about a file.
Get the attributes of a folder	The DirectoryInfo object offers several properties that provide information about a file.

Key Point	The rules that direct the transformation of ScanTool code to use System.IO are instead of Scripting include a number of cases that require inspection of the code stream to decide how to replace instances of TextStream based on usage. At present, a Migration DLL is needed to extend the rules with this logic. A detailed discussion of Migration DLL implementation is not in scope for this document. Please contact Great Migrations if you would like additional information.

3) Design and implement the rules to migrate from COM to .NET

The rules for replacing Scripting elements with System.IO are specified in a Custom IDF. In order to get started with a Custom IDF you do the following two steps:

- Copy the standard IDF (scrrun.dll.xml) from the IDF\FromIDL folder into the usr folder and rename it to GM.scrrun.dll.xml. Note this naming is just a convention, but it helps trace the Custom IDF back to its source.
- Add a registry-libname statement to the translation script to tell gmStudio that you want to use the Custom IDF instead of the standard one.

  <Registry type="libname" source="scrrun.dll" target="GM.scrrun.dll" />

The following sections explain how to customize the Custom IDF to specify the rules that will migrate code that used Scripting to System.IO.

Remove Reference to scrrun.dll and replace Scripting with System.IO

Every IDF contains a library node that specifies how to author references to the API assembly in your .NET project. The table below shows how to do this specification for the scrrun.dll (Scripting) library – for both standard (prototype) and refactoring translations.

Standard Interface Description File, scrrun.dll.xml	Custom Interface Description File, GM.scrrun.dll.xml

`< DescriptionFile >`
`<library id="scrrun.dll"`
`      location="%library%\Interop.Scripting.dll"`
`      migName="Scripting"`
`      type="LocalImport"`
`...`
`>`
`...`
`</DescriptionFile >`
	`<DescriptionFile>`
	`<library id="GM.scrrun.dll"`
	`      location="DoNotDeclare"`
	`       migName="System.IO"`
	`       type="Native"`
	`...`
	`>`
	`...`
	`</DescriptionFile >`

The library node tells gmBasic that .NET projects that use the Scripting object model should use a namespace called "Scripting". If you use the Interop strategy your code will find the implementation of this namespace in an Interop assembly called Interop.Scripting.dll. The meta variable %library% is set in the translation script and defaults to the Interop folder in the migration workspace. If you use the prototype strategy, the implementation of this namespace will be a stub class generated by gmStudio.	The id is changed to match the containing file name without the .xml extension. location="DoNotDeclare" tells gmBasic not to author a reference in the .NET project file. System.IO is part of the System.dll assembly that, by default, referenced in .NET projects generated by gmStudio, so no other declaration is needed. In the library element, migName="System.IO" tells gmBasic that the namespace Scripting is replaced with System.IO. type="Native" tells gmBasic to author the reference in .NET style; this is moot here since DoNotDeclare means no reference will be authored, but it serves to provide documentation.

Replacing Scripting types with System.IO types

A standard IDF file contains class and coclass elements that describe the API exposed by the COM component. Adding a migName attribute to any of these elements tells gmBasic that the .NET replacement element has a different name. Several examples are listed below.

The Scripting API uses an instance of the FileSystemObject class to inspect and manipulate the file system and open text files. The System.IO object model has no single class, rather these services are provided by a number of different static classes and objects in a way that makes a FileSystemObject instance unnecessary. Adding the migStatus="static" attribute for the FileSystemObject tells gmBasic to remove instances of FileSystemObject from the code.

<coclass id="FileSystemObject">	<coclass id="FileSystemObject" migStatus="static">

The examples below show how to specify simple type name changes:

<coclass id="Drive" creatable="off">	<coclass id="Drive" creatable="off" migName="DirectoryInfo">
<coclass id="Folder" creatable="off">	<coclass id="Folder" creatable="off" migName="DirectoryInfo">
<coclass id="File" creatable="off">	<coclass id="File" creatable="off" migName="FileInfo">

The following two examples show how different patterns for C# and VB.NET may be specified, using cshPattern and vbnPattern, respectively:

<coclass id="Folders" creatable="off">	<coclass id="Folders" creatable="off"
	cshPattern="DirectoryInfo[]"
	vbnPattern="DirectoryInfo()">

<coclass id="Files" creatable="off">	<coclass id="Files" creatable="off"
	cshPattern="FileInfo[]"
	vbnPattern="FileInfo()" >

The following example shows how to rename a type and also tell the tool that it should apply casts or conversions when operations that use it are authored. This is because TextStream will get special handling (see Scripting Migration DLL below.)

<coclass id="TextStream" creatable="off">	<coclass id="TextStream" creatable="off" migName="Stream" CastType="off">

In the mapping for Dictionary below, note the use of a fully qualified .NET type and migStatus="external". This is so, because the replacement does not exist in default replacement namespace (System.IO)

<coclass id="Dictionary">	<coclass id="Dictionary"
	migName="System.Collections.Specialized.OrderedDictionary"
	migStatus="External" >

Replacing an Enum

An IDF contains information about the enums and their entries. Both enum type names and entry names can be migrated using the migName attribute.

The first example shows simply renaming the enum type and its entries. Notice that the replacement enum is not part of replacement namespace (System.IO), so it is qualified with the namespace alias VBNET and has migStatus="external". Note that a using VBNET=Microsoft.VisualBasic statement is part of the default code file template used by gmStudio (See the file textblock.xml in your gmStudio installation.

<enumeration id="CompareMethod">
   <entry id="BinaryCompare" value="0"/>
   <entry id="TextCompare" value="1"/>
   <entry id="DatabaseCompare" value="2"/>
</enumeration>
	<enumeration id="CompareMethod"
	migName="VBNET.CompareMethod"
	migStatus="External">
	   <entry id="BinaryCompare" value="0" migName="Binary"/>
	   <entry id="TextCompare" value="1" migName="Text"/>
	   <entry id="DatabaseCompare" value="2" migName="Database"/>
	</enumeration>

The second example shows that more complex replacement patterns are possible. Here, the enum entries are changed to static methods that return strings containing the special folder paths. The enum type is mapped to the namespace that exposes these static methods.

```
<enumeration id="SpecialFolderConst">


  <entry id="WindowsFolder" value="0"/>

  <entry id="SystemFolder" value="1"/>

  <entry id="TemporaryFolder" value="2"/>

</enumeration>
```

```
<enumeration id="SpecialFolderConst"
             migName="System.Environment"
             migStatus="External">
<entry id="WindowsFolder" value="0"

migName="GetFolderPath(Environment.SpecialFolder.Windows)"/>
<entry id="SystemFolder" value="1"

migName="GetFolderPath(System.Environment.SpecialFolder.System)"/>
<entry id="TemporaryFolder" value="2"
            migName="System.IO.Path.GetTempPath()"/>
</enumeration>
```

This non-intuitive rule is needed because the SpecialFolderConst enum was intended for use as an argument to the GetSpecialFolder method as shows in the standard IDF (scrrun.dll.xml). The replacement operation in .NET is done by calling DirectoryInfo() and passing the path rather than by calling GetSpecialFolder and passing an enum entry.

```
<method id="GetSpecialFolder"
type="Folder">

  <argument id="SpecialFolder"

type="SpecialFolderConst"
                       status="ByVal"/>
</method>
```

```
<method id="GetSpecialFolder" type="Folder"
                 nPram="2" migPattern="new
DirectoryInfo(%2d)">
  <argument id="SpecialFolder"
                     type="SpecialFolderConst"
                    status="ByVal"/>
</method>
```

Replacing Members

An IDF contains information about the properties, methods, accessors (parameterized properties), and events exposed by the classes of the COM type. These elements can be migrated by adding migration attributes and making other modifications to the member nodes in the Custom IDF. Some of these modifications are described below.

Renaming a Member

Adding a migName attribute to a member changes the name used to refer to that operation in translated codes.
The first set of example shows that simple property names can be replaced with simple names or more complex patterns:

```
<class id="IDrive" parent="IDispatch"
default="Path">
  <property id="Path" type="String"
status="Out"/>

  <property id="DriveLetter" type="String"
          status="Out"/>
  <property id="RootFolder" type="Folder"
          status="Out"/>
  <property id="IsReady" type="Boolean"
status="Out" />
```

```
<class id="IDrive" parent="IDispatch"
default="Path">
  <property id="Path" type="String" status="Out"
                  migName="Root.FullName" />
  <property id="DriveLetter" type="String"
status="Out"

migName="Root.FullName.Substring(0,1)" />
  <property id="RootFolder" type="Folder"
status="Out"
                     migName="Root" />
  <property id="IsReady" type="Boolean"
status="Out"
                     migName="Exists" />
```

Changing the type of a member

The type attribute of each member affects how it is handled in various operations such as assignments and parameter passing. Changing the type impacts how conversions are applied to expressions involving the member.

An example is the Dictionary.HashVal. In COM this was an accessor (a parameterized property) that returned a variant and took a ByRef variant argument: In .NET, the member has a different name and return type as well as different marshaling of the argument:

```
<accessor id="HashVal" type="Variant">
     <argument id="Key" type="Variant"
status="ByRef"/>
</accessor>
```

```
<accessor id="HashVal" type="Long"
migName="GetHash" >
     <argument id="Key" type="Variant"
status="ByVal"/>
</accessor>
```

Changing the reference type of a member

Sometimes a member that was a property in COM becomes a method in .NET. For example, in Scripting, IFolder.IsRootFolder is a property. In .NET it is authored as a method.

`<property id="IsRootFolder" type="Boolean"`
`status="Out"`
 	
`/>`
	`<method id="IsRootFolder" type="Boolean"`
	`status="Out"`
	`       nPram="1"`
	`        cshPattern="(%1d.Parent == null)"`
	`        vbnPattern="(%1d.Parent = Nothing)" />`

Changing the calling pattern for a member

The default calling pattern for a class member is object.member(arguments). This can be modified to reflect changes in the ordering or parameters and even the structure of the call by adding a pattern specification to the member.

A couple of example rules that direct gmStudio to replace FileSystemObject methods with calls to static methods are shown below:

`<class id="IFileSystem"`
`parent="IDispatch">`
`<method id="BuildPath" type="String">`
 	
`<method id="GetDriveName" type="String">`
	`<class id="IFileSystem" parent="IDispatch">`
	`      <method id="BuildPath" type="String"`
	`              nPram="3"`
	`              migPattern="System.IO.Path.Combine(%2d,%3d)" >`
	`      ...`
	`      <method id="GetDriveName" type="String"`
	`   nPram="2"`
	`   cshPattern="System.IO.Path.GetPathRoot(%2d).TrimEnd('\\\')"`
	`   vbnPattern='System.IO.Path.GetPathRoot(%2d).TrimEnd("\\")'>`

nPram	is the number of parameters to pull off the ‘string stack’ when authoring the .NET code. The ‘string stack’ contain the pCode stream that represents the operations of your code. The top of the stack contains the expressions for t objects and arguments associated with the current operation.
cshPattern	is the desired ‘surface form’ for expressing the operation in C#. Notice that the cshPattern is parameterized. The notation %1d means author the expression associated with the object that will receive the call. %2d is the first ar only argument to the call.
vbnPattern	is the desired ‘surface form’ for expressing the operation in VB.NET.
migPattern	is a shorthand notation that can be used with the desired surface form is the same for C# and VB.NET

Changing a default Item property to an indexer

Many collections have a default parameterized property (typically named Item) that should be replaced by an indexer in .NET. An example for setting up the rule to make this transformation is shown below. The modifications needed to replace this with indexer notation are shown below. Notice both C# and VB.NET patterns are needed and that the argument to the indexer is now passed by value.

`<class id="IDictionary" parent="IDispatch"`
`default="Item">`
`       ...`
`      <accessor id="Item" type="Variant">`
 	
`         <argument id="Key" type="Variant"`
`status="ByRef"/>`
`      </accessor>`
	`<class id="IDictionary" parent="IDispatch"`
	`default="Item">`
	`       ...`
	`      <accessor id="Item" type="Variant"`
	`                nPram="2"`
	`                cshPattern="%1d[%2d]"`
	`                vbnPattern="%1d(%2d)" >`
	`         <argument id="Key" type="Variant"`
	`status="ByVal"/>`
	`      </accessor>`

Using a Migration DLL

The migration of Scripting to System.IO requires a few rules that cannot be completely specified solely by the declarative notation of a Custom IDF. In order to accommodate these rules in a flexible and extensible manner, gmStudio uses Migration DLLS. A migration DLL is a library of functions that extend the behavior of gmStudio. A Migration DLL is associated with a COM API in the Refactor section of the Custom IDF. The Refactor section may also include various code patterns to be used for authoring the .NET code. Although the details of creating and using Migration DLLs is beyond the scope of this document, the curious reader will find a portion of the Refactor element for Scripting below.

```
<Refactor id="[GM.scrrun.dll]" dllName="ScrrunMigration.dll" event="scrrun" >
   <migclass id="DotNet">
     <method id="ForEachDrive" type="void" nPram="3"
             cshPattern="foreach(string drvTemp in
               System.IO.Directory.GetLogicalDrives())\n{\p%3d %1d = new %3d(drvTemp)\c"
             vbnPattern="Dim drvTemp As String\nFor Each drvTemp in
               System.IO.Directory.GetLogicalDrives()\pDim %1d As %3d = New
%3d(drvTemp)\c">
         <argument id="Drives" type="variant" status="In"/>
         <argument id="numb" type="Integer" status="In"/>
         <argument id="type" type="variant" status="In"/>
     </method>
   </migclass>

   <migClass id="TextReader" netName="System.IO.StreamReader" Creatable="off">
      <method id="OpenAsciiFile" type="TextReader"
               migPattern="(new System.IO.FileInfo(%2d)).OpenText()">
         <argument id="filename" type="String" status="In"/>
         <argument id="IOMode" type="IOMode" status="In" optional="ForReading"/>
         <argument id="Create" type="Boolean" status="In" optional="false"/>
         <argument id="Format" type="Tristate" status="In" optional="TristateFalse"/>
      </method>
```

Furthermore, if you inspect the Custom IDF for Scripting, you will find a number of API members have migStatus attributes with special values; for example:

```
   <class id="IFileSystem" parent="IDispatch">
      <property id="Drives" type="Drives" status="Out"

      ...
      <method id="OpenTextFile" type="TextStream"

      ...
      </method>

   <class id="ITextStream" parent="IDispatch">
       ...
      <method id="ReadLine" type="String" />

      <method id="ReadAll" type="String" />

      <method id="Write" type="Void" >
         <argument id="Text" type="String"
status="ByVal"/>
      </method>
      <method id="WriteLine" type="Void"  >
         <argument id="Text" type="String"
status="ByVal"
                              optional="Default"/>
      </method>
       ...
```

```
   <class id="IFileSystem" parent="IDispatch">
      <property id="Drives" type="Drives" status="Out"
                        migStatus="scrrun" />
      ...
      <method id="OpenTextFile" type="TextStream"
                      migStatus="scrrun" >
      ...
      </method>

   <class id="ITextStream" parent="IDispatch">
       ...
      <method id="ReadLine" type="String"
                      migStatus="scrrun.read" />
      <method id="ReadAll" type="String"
                      migStatus="scrrun.read"
migName="ReadToEnd" />
      <method id="Write" type="Void"
migStatus="scrrun.write" >
         <argument id="Text" type="String"
status="ByVal"/>
      </method>
      <method id="WriteLine" type="Void"
migStatus="scrrun.write" >
         <argument id="Text" type="String"
status="ByVal"
                             optional="Default"/>
      </method>
       ...
```

These migStatus values represent translation events context codes that are passed into the Migration DLL event handlers and may be used to direct processing in the body of the handler when it responds to events raised by the translation engine. By convention, if a migStatus attribute value is not one of the predefined migStatus enumeration entries (defined in the system configuration file enumerations.xml) then it is treated as a user-defined translation event context code.

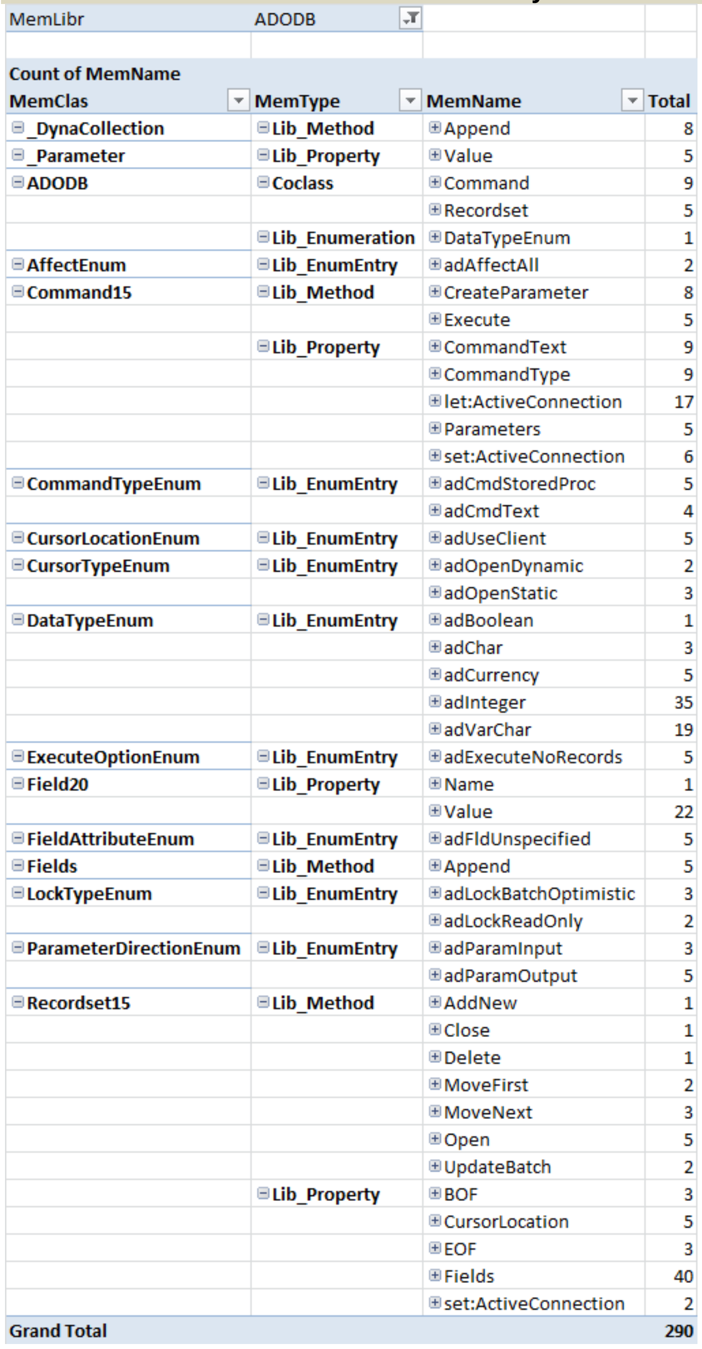
Appendix C: Upgrading ADODB to System.Data and MigrationSupport.DataLib

The following section offers more detail on how COM replacement is approached with gmStudio. This discussion applies to the FMStocks sample code. FMStocks uses a number of COM APIs, one of which is Microsoft ActiveX Data Objects (**msado27.tlb)**. This appendix shows how code that uses ADODB can be migrated to System.Data and MigrationSupport.DataLib by gmStudio.

1) Understand the services legacy components provide to your application

A portion of the Semantic Reference Report for the FMStocks sample and the ADODB API is shown in table C1. It shows FMStocks references 44 different elements of the ADODB library in the FMSLib source code.

Table C1: Cross-Tabulated ADODB Library Reference Report for FMStocks FMSLib Sample



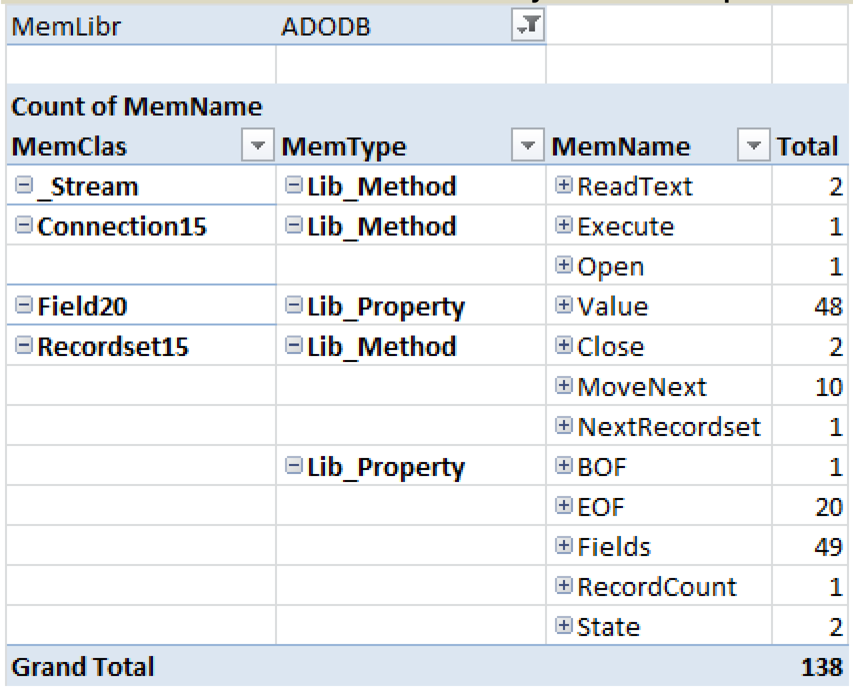
MemLibr	ADODB		
Count of MemName			
MemClas	**MemType**	**MemName**	**Total**
_DynaCollection	**Lib_Method**	Append	8
_Parameter	**Lib_Property**	Value	5
ADODB	**Coclass**	Command	9
		Recordset	5
	Lib_Enumeration	DataTypeEnum	1
AffectEnum	**Lib_EnumEntry**	adAffectAll	2
Command15	**Lib_Method**	CreateParameter	8
		Execute	5
	Lib_Property	CommandText	9
		CommandType	9
		let:ActiveConnection	17
		Parameters	5
		set:ActiveConnection	6
CommandTypeEnum	**Lib_EnumEntry**	adCmdStoredProc	5
		adCmdText	4
CursorLocationEnum	**Lib_EnumEntry**	adUseClient	5
CursorTypeEnum	**Lib_EnumEntry**	adOpenDynamic	2
		adOpenStatic	3
DataTypeEnum	**Lib_EnumEntry**	adBoolean	1
		adChar	3
		adCurrency	5
		adInteger	35
		adVarChar	19
ExecuteOptionEnum	**Lib_EnumEntry**	adExecuteNoRecords	5
Field20	**Lib_Property**	Name	1
		Value	22
FieldAttributeEnum	**Lib_EnumEntry**	adFldUnspecified	5
Fields	**Lib_Method**	Append	5
LockTypeEnum	**Lib_EnumEntry**	adLockBatchOptimistic	3
		adLockReadOnly	2
ParameterDirectionEnum	**Lib_EnumEntry**	adParamInput	3
		adParamOutput	5
Recordset15	**Lib_Method**	AddNew	1
		Close	1
		Delete	1
		MoveFirst	2
		MoveNext	3
		Open	5
		UpdateBatch	2
	Lib_Property	BOF	3
		CursorLocation	5
		EOF	3
		Fields	40
		set:ActiveConnection	2
Grand Total			**290**

It appears that FMStocks is executing Commands against the database (both stored procedures and inline SQL), and working with Recordsets.

It also appears that some Recordset instances are being created by hand based on the use of Fields.Append() and Recordset15.AddNew().

Now here is the same report run against the FMSWeb source code from the FMStocks sample:

Table C2: Cross-Tabulated ADODB Library Reference Report for FMStocks FMSWeb Sample



MemLibr	ADODB		
Count of MemName			
MemClas	MemType	MemName	Total
_Stream	Lib_Method	ReadText	2
Connection15	Lib_Method	Execute	1
		Open	1
Field20	Lib_Property	Value	48
Recordset15	Lib_Method	Close	2
		MoveNext	10
		NextRecordset	1
	Lib_Property	BOF	1
		EOF	20
		Fields	49
		RecordCount	1
		State	2
Grand Total			138

The FMSWeb code is using only 12 different elements from the ADODB API. There is some usage overlap between the two source codes, and the use cases are very similar.

2) Define your COM replacement strategy

Table C3: High-level Redesign Strategies

Legacy API Service	Replacement Strategy
Reference to msaado27.tlb	Replace with MigrationSupport.DataLib namespace. This namespace is in the MigrationSupport.dll assembly distributed with gmStudio, and it is available by default in .NET project files generated by gmStudio.
Establish a connection to the database	The ADODB.Connection15 is being used to Open a connection to the database.
Execute a Command against the database	The ADODB.Command CoClass in a chain of interfaces inherits from Command15 which provides methods for adding parameters and executing CommandText, and a series of proeprties exposing the Parameters collection, the CommandText, the ActiveConnection, and the CommandType.
Iterate over a Recordset	The ADODB.Recordset CoClass in a chain of interfaces inherits from Recordset15 which provides the Open method for retrieving data from a Command object, and MoveNext() and MoveFirst() methods for iterating over that retrieved data. Some properties specify the CursorLocation as well as the type of cursor.
Create a Recordset	ADODB exposes a Fields collection through the Recordset15 interface specifying column metadata as well as the value in a Recordset. The AddNew() method on Recordset15 adds a new row to the Recordset, and the Append() method on Fields adds a new Field to the Fields collection.

3) Design and implement the rules to migrate from COM to .NET

The rules for replacing ADODB elements are specified in a Custom IDF. In order to get started with a Custom IDF, perform the following two steps:

- Copy the standard IDF (msado27.tlb.xml) from the IDF\FromIDL folder into the usr folder and rename it to GM.msado27.tlb.xml. Note this naming is just a convention, but it helps trace the Custom IDF back to its original source.

- Add a registry-libname statement to the translation script to tell gmStudio that you want to use the Custom IDF instead of the standard one.

  ```
  <Registry type="libname" source="msado27.tlb" target="GM.msado27.tlb" />
  ```

The following sections explain how to customize the Custom IDF to specify the rules that will migrate code using ADODB to System.Data and MigrationSupport.DataLib.

Remove Reference to msado27.tlb and replace ADODB with MigrationSupport.DataLib

Every IDF contains a library node that specifies how to author references to the API assembly in you .NET project. The library node of the Standard and Custom IDFs for the ADODB library is shown below.

Standard IDF, msado27.tlb.xml	Custom IDF, GM.msado27.tlb.xml
`<DescriptionFile>` `<library id="msado27.tlb"` `name="ADODB"` `...` `location="%library%\Interop.ADODB.dll"` `migName="ADODB"` `type="LocalImport"` `>`	`< DescriptionFile >` `<library id="GM.msado27.tlb" name="ADODB"` `...` `location="DoNotDeclare"` `migName="MigrationSupport.DataLib"` `type="Native"` `>`
This library node tells gmBasic that .NET projects that use the ADODB object model should use a namespace called ADODB in an Interop assembly called Interop.ADODB.dll. The meta variable %library% is set in the translation script and defaults to the Interop folder in the migration workspace. If you use the prototype strategy, the implementation of this namespace will be a stub class generated by gmStudio.	The id is changed to match the containing file name without the .XML extension. The attribute location="DoNotDeclare" tells gmBasic not to author a reference in the .NET project file. MigrationSupport.DataLib is part of the MigrationSupport.dll assembly that is in the default .NET project file template used by gmStudio, so no other declaration is needed. In the library element, migName="MigrationSupport.DataLib" tells gmBasic that the namespace ADODB is replaced with MigrationSupport.Datalib. The attribute type="Native" tells gmBasic to author the project reference in .NET style; this is moot here since DoNotDeclare means no reference will be authored, but it serves to provide documentation. The closing tag for the root node must match the element name of the root node.

Replacing Command and Command15

The report in Table C1 shows that Command is referenced directly and members of Command15 are referenced. Some research would turn up that System.Data.Common.DbCommand has similar members that cover all of those referenced in Command15. Migrating Command15 and the associated members would look like the following:

`<class id="Command15" parent="_ADO"` `default="Parameters">`	`<class id="Command15" parent="_ADO"` `default="Parameters"` `migName="System.Data.Common.DbCommand"` `migStatus="External">`

The migName attribute tells gmBasic to migrate references to Command15 to System.Data.Common.DbCommand. The attribute migStatus="External" tells gmBasic that the type is not part of the MigrationSupport.DataLib namespace. Otherwise, gmBasic will attempt to prefix the type specified in migName with MigrationSupport.DataLib.

DbCommand was chosen over OleDbCommand since no instances of Command15 are created directly in the application, and DbCommand allows for a more abstract approach to the types migration.

There are no variables declared as Command15, nor are any instances created of that type. Declaring the new type serves more as documentation.

The ActiveConnection property needs to be updated to reflect the name for the Connection on the DbCommand, which is Connection. There is no concept of a Let method for properties in .NET. So when the Let method is required for the ActiveConnection property, it is migrated to either the C# pattern defined by cshPattern or the VB.NET pattern as defined by vbnPattern.

```
<property id="ActiveConnection"
type="Connection"
                  status="GetSetLet">
<Get npram="1"
         migPattern="%1d.ActiveConnection"/>
<Set npram="2"
         migPattern="%1d.ActiveConnection =
%2d\c"/>
<Let npram="2"

migPattern="%2d.let_ActiveConnection(%1d)\c"/>




</property>
```

```
<property id="ActiveConnection"
type="Connection"
                  status="GetSetLet">
         <Get npram="1"
          migPattern="%1d.Connection"/>
         <Set npram="2"
         migPattern="%1d.Connection = %2d\c"/>
         <Let npram="2"
         cshPattern="%2d.Connection = new

System.Data.OleDb.OleDbConnection(%1d)\c"
         vbnPattern="%2d.Connection = New

System.Data.OleDb.OleDbConnection(%1d)\c"/>
 </property>
```

It's not shown in Table C1, but the use of Execute on the Command15 objects is set up to not return any data, the number of records affected is ignored, and there are no parameters being passed in. The System.Data.Common.DbCommand class offers the ExecuteNonQuery function as an equivalent.

```
<method id="Execute" type="Recordset">

     <argument id="RecordsAffected"
type="Variant"
                         status="Out"
optional="Default"/>
     <argument id="Parameters" type="Variant"
status="ByRef"
                         optional="Default"/>
     <argument id="Options" type="Integer"
status="ByVal"
                         optional="-1"/>
</method>
```

```
<method id="Execute" type="Recordset" nPram="4"

migPattern="%1d.ExecuteNonQuery()\c">
     <argument id="RecordsAffected"
type="Integer"
                          status="ByRef"
optional="Default"/>
     <argument id="Parameters" type="Variant"
status="ByRef"
                         optional="Default"/>
     <argument id="Options" type="Integer"
status="ByVal"
                          optional="Default"/>
</method>
```

No changes are necessary for the parameters since they're unused.

```
<method id="CreateParameter" type="Parameter" nPram="6"

migPattern="MigrationSupport.DataLib.Utils.CreateParameter(%1d,%2d,%3d,%4d,%5d,%6d)">
    <argument id="Name" type="String" status="ByVal" optional="Default"/>
    <argument id="Type" type="DataTypeEnum" status="ByVal" optional="adEmpty"/>
    <argument id="Direction" type="ParameterDirectionEnum" status="ByVal"
optional="adParamInput"/>
    <argument id="Size" type="Integer" status="ByVal" optional="0"/>
    <argument id="Value" type="Object" status="ByVal" optional="Default"/>
</method>
```

The Connection and Recorrdset CoClasses are migrated to the equivalent OleDb implementation in the MigrationSupport.DataLib namespace. Because the migStatus is not specified, gmBasic will prefix "OleDb.Recordset" with the namespace “MigrationSupport.DataLib” defined in the migName attribute for the DescriptionFile element. An alternative would have been to specify “MigrationSupport.DataLib.OleDb” as the migName for the DescriptionFile element. In doing so though, any classes that exist in the parent namespace would have to be specified with the full type and namespace and set as migStatus=”External”. Either method will work, it’s simply a matter of preference.

The Command CoClass is migrated to System.Data.OleDb.OleDbCommand. Like Command15, the migStatus is set to External so that gmBasic doesn’t attempt to prefix the type with MigrationSupport.Datalib.

```
   <coclass id="Connection" >

      <subclass id="_Connection"/>
      <subclass id="ConnectionEvents"/>
   </coclass>
...
   <coclass id="Command">


      <subclass id="_Command"/>
   </coclass>
   <coclass id="Recordset" >
      <subclass id="_Recordset"/>
      <subclass id="RecordsetEvents"/>
   </coclass>
```

```
   <coclass id="Connection"
                   migName="OleDb.Connection">
      <subclass id="_Connection"/>
      <subclass id="ConnectionEvents"/>
   </coclass>
...
   <coclass id="Command"

migName="System.Data.OleDb.OleDbCommand"
                   migStatus="External">
      <subclass id="_Command"/>
   </coclass>
   <coclass id="Recordset"
migName="OleDb.Recordset">
      <subclass id="_Recordset"/>
      <subclass id="RecordsetEvents"/>
   </coclass>
```

Replacing _DynaCollection

The report in Table C1 shows a reference to the _DynaCollection type. More specifically, the report shows a reference to the Append method of the _DynaCollection type. The _DynaCollection type isn’t a type that’s typically consumable by applications using ADODB. To see what the _DynaCollection is, perform a search in the custom IDF.

A search will show that the Parameters class inherits from the _DynaCollection type. That means the Append function is likely adding a parameter to the Parameters collection. Next, more investigation needs to be done to see how the Parameters collection is being used.

A search shows that the Command15 class is using Parameters as the type for the Parameters property. Command15 is being migrated to System.Data.Common.DbCommand. DbCommand laso has a Parameters property. The property is of DbParameterCollection. DbParameterCollection doesn’t have an Append method, but there is a method named Add. Therefore, the Append method of _DynaCollection can be migrated with a simple change to the method name as follows:

```
<class id="_DynaCollection" parent="_Collection"
               creatable="off">
      <method id="Append" type="Void">
              <argument id="Object" type="Object"
status="ByVal"/>
      </method>
...
</class>
```

```
 <class id="_DynaCollection"
parent="_Collection"
               creatable="off">
      <method id="Append" type="Void"
migName="Add">
              <argument id="Object"
type="Object" status="ByVal"/>
      </method>
...
</class>
```

Replacing Connection15

The report in Table C1 shows a reference to the Connection15 type. The connection object is used to establish a connection to the database. The type isn’t instantiated directly, and there are only a couple of members that are referenced. The Connection15 type can be migrated as follows:

```
<class id="Connection15" parent="_ADO"
           default="ConnectionString">
...
      <method id="Execute" type="Recordset" >

           <argument id="CommandText"
type="String"
                              status="ByVal"/>
```

```
<class id="Connection15" parent="_ADO"
           default="ConnectionString">
...
      <method id="Execute" type="Recordset"
nPram="4"

migPattern="%1d.Execute(%2d,%3o,%4o)">
           <argument id="CommandText"
```

`<argument id="RecordsAffected" type="Variant" status="Out" optional="Default"/> <argument id="Options" type="Integer" status="ByVal" optional="-1"/> </method>`	`type="String" status="ByVal"/> <argument id="RecordsAffected" type="Integer" status="ByRef" optional="DEF.Overload"/> <argument id="Options" type="Integer" status="ByVal" optional="DEF.Overload"/> </method>`

The MigrationSupport.DataLib namespace has a custom Connection type specified that has several overloads for Execute. To match the new overloads, the pattern had to be set up as specified in the migPattern attribute. Of the four parameters, %1d is the Connection object that Execute is being called on. %2d is the CommandText, and is required. Parameters %3o and %4o are optional, which is the "o" portion of the parameter specification.

The parameters themselves were updated to be used as part of an overload instead of using default values, as indicated by the updated "optional" attribute in the argument elements.

RecordsAffected was updated with a strong type as opposed to variant. It was also set up as a ByRef parameter instead of an output parameter to match the signature of the method on the compatibility class.

`<method id="Open" type="Void"> <argument id="ConnectionString" type="String" status="ByVal" optional="Default"/> <argument id="UserID" type="String" status="ByVal" optional="Default"/> <argument id="Password" type="String" status="ByVal" optional="Default"/> <argument id="Options" type="Integer" status="ByVal" optional="-1"/> </method>`	`<method id="Open" type="Void" nPram="5" migPattern="%1d.Open(%2o,%3o,%4o,%5o)\c"> <argument id="ConnectionString" type="String" status="ByVal" optional="DEF.Overload"/> <argument id="UserID" type="String" status="ByVal" optional="DEF.Overload"/> <argument id="Password" type="String" status="ByVal" optional="DEF.Overload"/> <argument id="Options" type="Integer" status="ByVal" optional="DEF.Overload"/> </method>`

The Open method was updated changing the signature to a series of optional parameters. The parameters, instead of requiring the default values were updated to use the method signature overloads.

Updating Recordset and Fields Members

The Recordset, Fields, and Field types in the MigrationSupport.DataLib namespace were created to provide a nearly 1:1 replacement of the equivalent ADODB types. There are some exceptions in the members in an attempt to move more toward native ADO.NET types. The members are migrated as follows. The type for State was changed from Integer to ObjectStateEnum.

`<class id="Recordset15" parent="_ADO" default="Fields"> ... <property id="State" type="Integer" status="Out"/>`	`<class id="Recordset15" parent="_ADO" default="Fields"> ... <property id="State" type="ObjectStateEnum" status="Out"/>`

The Delete method was changed to use a different pattern specified by migPattern. The function takes two parameters as specified by the nPram attribute. The first parameter is the instance that the method is being called on. The second parameter is an optional parameter for AffectRecords. The notation %2o indicates parameter 2 is optional. In addition, the optional attribute of the AffectRecords parameter was updated to DEF.Overload telling gmBasic to use an overload of the method as opposed to passing in a default value.

```
<method id="Delete" type="Void">

    <argument id="AffectRecords"
type="AffectEnum"
                         status="ByVal"
optional="adAffectCurrent"/>
</method>
```

```
<method id="Delete" type="Void" nPram="2"
                 migPattern="%1d.Delete(%2o)\c">
     <argument id="AffectRecords"
type="AffectEnum"
                          status="ByVal"
optional="DEF.Overload"/>
</method
```

The MoveNext method is migrated to the Read method, as indicated by the migName attribute.

```
<method id="MoveNext" type="Void" />
```

```
      <method id="MoveNext" type="Void"
migName="Read"/>
```

The Open method is migrated to only use the first 3 of the 6 parameters: the Recordset instance, the source, and the ActiveConnection. The remaining arguments are ignored for now, so they aren't specified in the migPattern.

```
...
<method id="Open" type="Void">

 <argument id="Source" type="Variant"
                      status="ByVal"
optional="Default"/>
 <argument id="ActiveConnection" type="Variant"
                     status="ByVal"
optional="Default"/>
<argument id="CursorType" type="CursorTypeEnum"
                     status="ByVal"
optional="Default"/>
<argument id="LockType" type="LockTypeEnum"
                     status="ByVal"
optional="Default"/>
<argument id="Options" type="Integer"
                    status="ByVal" optional="-
1"/>
</method>
```

```
...
<method id="Open" type="Void" nPram="6"

migPattern="%1d.Open(%2o,%3o)\c">
 <argument id="Source" type="Variant"
                     status="ByVal"
optional="DEF.Overload"/>
 <argument id="ActiveConnection" type="Variant"
                     status="ByVal"
optional="DEF.Overload"/>
<argument id="CursorType" type="CursorTypeEnum"
                    status="ByVal"
optional="DEF.Overload"/>
<argument id="LockType" type="LockTypeEnum"
                   status="ByVal"
optional="DEF.Overload"/>
<argument id="Options" type="Integer"
status="ByVal"
                    optional="DEF.Overload"/>
</method>
```

The UpdateBatch method is migrated to use a migPattern. The migPattern doesn't take in the optional parameter. The parameter is always ignored so no updates are necessary for the parameter.

```
<method id="UpdateBatch" type="Void">

     <argument id="AffectRecords"
type="AffectEnum"
                         status="ByVal"
optional="adAffectAll"/>
</method>
```

```
<method id="UpdateBatch" type="Void"
                 nPram="2"
migPattern="%1d.Update()\c">
     <argument id="AffectRecords"
type="AffectEnum"
                         status="ByVal"
optional="adAffectAll"/>
</method>
```

The type for the RecordsAffected parameter of the NextRecordset method is updated from Variant to Integer.

```
<method id="NextRecordset" type="Recordset">
   <argument id="RecordsAffected"
                  type="Variant"
                  status="Out"
                  optional="Default"/>
</method>
```

```
<method id="NextRecordset" type="Recordset">
   <argument id="RecordsAffected"
             type="Integer"
             status="Out"
             optional=" DEF.Overload "/>
</method>
```

In the Fields class below, the Append method was migrated to use a specific migPattern. The migPattern ignores parameter 5, the FieldAttributeEnum, since there's no equivalent in .NET, and specifies that parameters 4 and 6 are optional. Overloads of the Append method will be called for the optional parameters.

```
<class id="Fields" parent="Fields20"
creatable="off">
      <method id="Append" type="Void"

         <argument id="Name" type="String"
status="ByVal"/>
```

```
<class id="Fields" parent="Fields20"
creatable="off">
      <method id="Append" type="Void" nPram="6"

migPattern="%1d.Append(%2d,%3d,%4o,%6o)\c">
         <argument id="Name" type="String"
```

```
          <argument id="Type" type="DataTypeEnum"
                               status="ByVal"/>
          <argument id="DefinedSize"
type="Integer"
                               status="ByVal"
optional="0"/>
          <argument id="Attrib"
type="FieldAttributeEnum"
                               status="ByVal"
optional="adFldUnspecified"/>
          <argument id="FieldValue"
type="Variant"
                              status="ByVal"
optional="Default"/>
      </method>
...
</class>
```

```
status="ByVal"/>
         <argument id="Type" type="DataTypeEnum"
                              status="ByVal"/>
         <argument id="DefinedSize"
type="Integer"
                              status="ByVal"
optional="DEF.Overload"/>
         <argument id="Attrib"
type="FieldAttributeEnum"
                              status="ByVal"
optional="adFldUnspecified"/>
         <argument id="FieldValue" type="Variant"
                             status="ByVal"
optional="DEF.Overload"/>
      </method>
...
</class>
```

Replacing an Enum

There are several enumerations that the report in Table C1 shows are referenced in the application. Only one example is given here.

The CommandTypeEnum is used to indicate what the purpose of a Command instance is used for. That helps to optimize how the Command instance executes against the database. System.Data.Common.DbCommand still has a CommandType property, but the type needs to be migrated to the new type of System.Data.CommandType, and the enumeration values need to be migrated to the new names. That can be done as follows:

```
<enumeration id="CommandTypeEnum">



      <entry id="adCmdUnspecified"
value="0xffffffff"/>
      <entry id="adCmdUnknown" value="8"/>
      <entry id="adCmdText" value="1"
migName="Text"/>
      <entry id="adCmdTable" value="2"/>
      <entry id="adCmdStoredProc" value="4">

      <entry id="adCmdFile" value="256"/>
      <entry id="adCmdTableDirect" value="512"/>
</enumeration>
```

```
<enumeration id="CommandTypeEnum"

migName="System.Data.CommandType"
                          migStatus="External">
      <entry id="adCmdUnspecified"
value="0xffffffff"/>
      <entry id="adCmdUnknown" value="8"/>
      <entry id="adCmdText" value="1"
migName="Text"/>
      <entry id="adCmdTable" value="2"/>
      <entry id="adCmdStoredProc" value="4"
                 migName="StoredProcedure"/>
      <entry id="adCmdFile" value="256"/>
      <entry id="adCmdTableDirect" value="512"/>
</enumeration>
```

The type is migrated to System.Data.CommandType as indicated by the migName attribute. The attribute "migStatus" indicates that the type is outside of the MigrationSupport.DataLib namespace, and it tells gmBasic not to prefix the type with the namespace.

The two enumeration values in use, adCmdText and adCmdStoredProc, are then migrated to the equivalent enumeration values in the new type, as indicated by the migName attributes on each.

Appendix D: Upgrading MSComCtlLib to Windows.Forms Controls

The following section offers more detail on how COM replacement is approached with gmStudio. This discussion applies to the FileExplorer sample code. FileExplorer uses Microsoft Common Controls (**MSComCtl.ocx)** to implement a lightweight windows folder explorer. This appendix shows how FileExplorer can be migrated to use System.Windows.Forms (**WinForms**) Controls with gmStudio.

1) Understand the services legacy components provide to your application

A portion of the Semantic References report for the FileExplorer sample and the MSComCtlLib API is shown in Table D1. It shows FileExplorer references 15 different types from the MSComCtlLib library a total of 139 times. A cursory analysis shows that MSComCtlLib uses the ListView, TreeView, ToolBar, StatusBar, and ImageList controls.

Table D1: Cross-Tabulated MSComCtlLib Library Reference Report for FileExplorer Sample

Count of MemName	**RecType**		
MemClas	**GUI**	**LOGIC**	**Grand Total**
IButton	18	5	23
IButtonMenu	3		3
IColumnHeader	16	4	20
IImage	11		11
IImageList	8		8
IListItem		3	3
IListItems		1	1
IListSubItems		3	3
IListView	13	5	18
INode		10	10
INodes		1	1
IPanel	11	1	12
IStatusBar		1	1
IToolbar	9		9
IToolbarEvents		1	1
ITreeView	8	5	13
ITreeViewEvents		1	1
TreeRelationshipConstants		1	1
Grand Total	**97**	**42**	**139**

Notice that there are two columns: GUI and LOGIC. The GUI column lists the references to MSComCtlLib from the designer section of the Form files (aka, the PropertyBag). The LOGIC column lists the references to MSComCtlLib from the logic sections of the code files. The LOGIC references may be from event handlers or from application sub programs. Designer code, event handlers, and application sub programs each require slightly different migration tactics.

2) Define your COM replacement strategy

The Reference Report shows that the MSComCtlLib library is being used for a number of UI controls. Most of these controls have obvious replacements in WinForms; using these WinForms classes is the replacement strategy.

Table D2 presents some examples of the redesign solutions needed to replace the MSComCtlLib controls with System.Windows.Forms (WinForms) for the FileExplorer sample.

Table D2: High-level Redesign Strategies	
Legacy API Service	**Replacement Strategy**
Reference to MSComCtl.ocx	Replace with the System.Windows.Forms namespace. This namespace is in the Sytem.Windows.Forms.dll assembly and is typically available by default.
ListView	WinForms.ListView and supporting classes
TreeView	WinForms.TreeView and supporting classes
Toolbar	WinForms.ToolStrip and supporting classes
Statusbar	WinForms.StatusStrip and supporting classes
ImageList	WinForms.ImageList and supporting classes

Key Point	Many of the controls in the MSComCtlLib API are composite controls: they have a top-level control as well as a number of sub-controls of some sort. For example, a ListView has a Columns, Items, and SubItems collections; a TreeView has a Nodes collection; a StatusBar has a Panels collection. The replacement WinForms controls have similar structures. Each top-level control and its sub-controls must be migrated as a coherent entity.

3) Design and implement the rules to migrate from COM to .NET

The rules for replacing MSComCtlLib elements with System.Windows.Forms are specified in a Custom IDF and a Migration DLL. In order to get started with the Custom IDF you do the following:

- Copy the standard IDF (MSComCtl.ocx.xml) from the IDF\FromIDL folder into the usr folder and rename it to GM.MSComCtl.ocx.xml. Note this naming is just a convention, but it helps trace the Custom IDF back to its source.
- Add a registry-libname statement to the translation script to tell gmStudio that you want to use the Custom IDF instead of the standard one.

  <Registry type="libname" source="MSComCtl.ocx" target="GM.MSComCtl.ocx" />

The following sections explain how to customize the Custom IDF to specify the rules that will rewrite code that used MSComCtlLib to System.Windows.Forms.

Remove Reference to MSComCtl.ocx and replace MSComCtlLib with System.Windows.Forms

Every IDF contains a library node that specifies how to author references to the API assembly in your .NET project. The library node in the MSComCtlLib IDF (MSComCtl.ocx.xml) looks like this:

Standard IDF, MSComCtl.ocx.xml	Custom IDF, GM. MSComCtl.ocx.xml

`<DescriptionFile>`
`<library id="MSComCtl.ocx"`
`...`
`location="%library%\Interop.MSComCtlLib.dll"`
`migName="MSComCtlLib"`
`type="LocalImport"`
`>`
`...`
`</DescriptionFile>`
	`<DescriptionFile>`
	`<library id="GM.MSComCtl.ocx"`
	`...`
	`location="DoNotDeclare"`
	`migName="System.Windows.Forms"`
	`type="Internal"`
	`>`
	`...`
	`</DescriptionFile>`

This library node tells gmBasic that .NET projects that use the MSComCtlLib object model should use a namespace called "MSComCtlLib". If you use the Interop strategy your code will find the implementation of this namespace in an Interop assembly called Interop.MSComCtlLib.dll. The meta variable %library% is set in the translation script and defaults to the Interop folder in the migration workspace. If you use the prototype strategy, the implementation of this namespace will be a stub class generated by gmStudio	We want to change the external assembly reference in the project file and also change the namespace. The modifications commands are made in GM.MSComCtl.ocx.xml to do this: The id is changed to match the containing file name without the .xml extension. location="DoNotDeclare" tells gmBasic not to author a reference in the .NET project file. System.Windows.Forms is part of the System.dll assembly that, by default, referenced in .NET projects generated by gmStudio, so no other declaration is

	needed. In the library element, migName=”System.Windows.Forms” tells gmBasic that the namespace MSComCtlLib is replaced with System.Windows.Forms. type="Internal" tells gmBasic to author the reference in .NET style; this is moot here since DoNotDeclare means no reference will be authored, but it serves to provide documentation.

Replacing MSComCtlLib types with System.Windows.Forms types

A standard IDF file contains class and coclass elements that describe the types exposed by the COM component. Adding a migName attribute to any of these elements tells gmBasic that the .NET replacement type has a different name. Several examples are listed below.

```
<coclass id="Toolbar" migName="ToolStrip" role="control"
migStatus="Add1:Items,Add2:System.Windows.Forms.ToolStripItem">

<coclass id="Button" creatable="off" migName="ToolStripButton" role="control" >

<coclass id="StatusBar" migName="StatusStrip" role="control"
                migStatus="Add1:Items,Add2:System.Windows.Forms.ToolStripItem">

<coclass id="Panel" creatable="off" migName="ToolStripStatusLabel" >

<coclass id="Node" creatable="off" migName="TreeNode" >
```

Replacing Members

An IDF contains information about the properties, methods, accessors (parameterized properties), and events exposed by the classes of the COM type. These elements can be migrated by adding migration attributes and making other modifications to the member nodes in the Custom IDF. A variety of these modifications are described below.

Renaming a Member

Adding a migName attribute to a member changes the name used to refer to that operation in translated codes.

The first set of example shows that simple property names can be replaced with simple names or more complex patterns:

```
   <class id="IStatusBar" parent="IDispatch">
      <property id="SimpleText" type="String" status="InOut" migName="Text" />
      <property id="Style" type="SbarStyleConstants" status="InOut"/>
      <property id="Panels" type="Panels" status="InOut" migName="Items" />
```

Changing the type of a member

The type attribute of each member affects how it is handled in various operations such as assignments and parameter passing. Changing the type impacts how conversions are applied to expressions involving the member.

```
   <class id="IToolbar" parent="IDispatch">
...
      <property id="ButtonHeight" status="InOut" migName="ButtonSize.Height" type="TwipsY" />
      <property id="ButtonWidth" status="InOut" migName="ButtonSize.Width" type="TwipsX" />
```

Changing the reference type of a member

Sometimes a member that was a method in COM becomes a property in .NET.

`<class id="ITreeView" parent="IDispatch">`
`       ....`
`      <method id="GetVisibleCount"`
`type="Integer"/>`
	`<class id="ITreeView" parent="IDispatch">`
	`      ...`
	`      <property id="GetVisibleCount"`
	`type="Integer"`
	`                        migName="VisibleCount"`

	/>

Changing the calling pattern for a member

The default calling pattern for a class member is object.member(arguments). This can be modified to reflect changes in the ordering of parameters and even the structure of the call by adding a pattern specification to the member.

`<class id="IListSubItems" parent="IDispatch"`
`default="ControlDefault"`
`creatable="off">`
`...`
`<method id="Add" type="ListSubItem">`
 	
`<argument id="Index" type="Variant"`
`status="ByRef"`
 	
`optional="Default"/>`
`<argument id="Key" type="Variant"`
`status="ByRef"`
 	
`optional="Default"/>`
`<argument id="Text" type="Variant"`
`status="ByRef"`
 	
`optional="Default"/>`
`<argument id="ReportIcon"`
`type="Variant" status="ByRef"`
 	
`optional="Default"/>`
`<argument id="ToolTipText"`
`type="Variant" status="ByRef"`
 	
`optional="Default"/>`
`</method>`
	`<class id="IListSubItems" parent="IDispatch"`
	`default="ControlDefault"`
	`creatable="off">`
	`...`
	`<method id="Add" type="ListSubItem"`
	`nPram="6"`
	`migPattern="%1d.Add(%3o,%4o,%5o,%6o)" >`
	`<argument id="Index"`
	`type="Variant" status="ByVal"`
	
	`optional="Default"/>`
	`<argument id="Key"`
	`type="String" status="ByVal"`
	
	`optional="DEF.Overload"/>`
	`<argument id="Text"`
	`type="String" status="ByVal"`
	
	`optional="DEF.Overload"/>`
	`<argument id="ReportIcon"`
	`type="Variant" status="ByVal"`
	
	`optional="DEF.Overload"/>`
	`<argument id="ToolTipText"`
	`type="Variant" status="ByVal"`
	
	`optional="DEF.Overload"/>`
	`</method`

	nPram is the number of parameters to pull off the 'string stack' when authoring the .NET code. The 'string stack' contains the pCode stream that represents the operations of your code. The top of the stack contains the expressions for the objects and arguments associated with the current operation. cshPattern is the desired 'surface form' for expressing the operation in C#. Notice that the cshPattern is parameterized. The notation %1d means "author the expression associated with the object that will receive the call". %2d is the first and only argument to the call. vbnPattern is the desired 'surface form' for expressing the operation in VB.NET. migPattern is a shorthand notation that can be used with the desired surface form is the same for C# and VB.NET

Changing a default Item property to an indexer

Many collections have a default parameterized property (typically named Item) that should be replaced by an indexer in .NET. An example for setting up the rule to make this transformation is shown below. Both the Standard and the Custom IDF are shown. The modifications needed to replace this with indexer notation are shown. Notice both C# and VB.NET patterns are needed and that the argument to the indexer is now passed by value.

```
 <class id="IButtons" parent="IDispatch"
            default="ControlDefault"
creatable="off">
       ...
      <accessor id="Item" type="Button">


            <argument id="Index" type="Variant"
status="ByRef"/>
      </accessor>
```

```
 <class id="IButtons" parent="IDispatch"
            default="ControlDefault"
creatable="off">
       ...
      <method id="Item" type="Button"
                       nPram="2"
migStatus="ZeroBased"
                       cshPattern="%1d[%2d]"
vbnPattern="%1d(%2d)" >
           <argument id="Index" type="Variant"
status="ByVal"/>
      </method>
```

Replacing an Event Handler

Replacing an even involved specifying the event name (netName), specifying the event handler type (netHandler) and specifying the event arguments type (netArgs).

In addition, you will frequently have to specify rules on the legacy arguments that will gmStudio how to replace the legacy arguments by declaring them as local variables and initializing them from the EventArgs instance.

```
Standard IDF

   <class id="ITreeViewEvents" parent="None" creatable="off">
...
      <event id="NodeClick"  netName="NodeClick"
                  netHandler="AxMSComctlLib.ITreeViewEvents_NodeClickEventHandler"
                  netArgs="AxMSComctlLib.ITreeViewEvents_NodeClickEvent">
         <argument id="node" type="Node" status="ByVal"/>
      </event>
```

```
Custom IDF

   <class id="ITreeViewEvents" parent="None" creatable="off">
      . . .
      <event id="NodeClick" role="event"
             netArgs="TreeNodeMouseClickEventArgs"
             netHandler="System.Windows.Forms.TreeNodeMouseClickEventHandler"
             migName="NodeMouseClick">
         <argument id="node" type="Node" status="ByVal" migPattern="%1d = e.Node" />
      </event>
```

Replacing an Enum

An IDF contains information about the enums and their entries. Both enum type names and entry names can be migrated using the migName attribute.

The first example simply replaces an enum and the names of its entries:

```
   <enumeration id="ToolbarStyleConstants" migName="ToolBarAppearance" >
      <entry id="tbrStandard" value="0" migName="Normal" />
      <entry id="tbrFlat" value="1" migName="Flat" />
   </enumeration>
```

More complex transformations of enums are needed for MSComCtl. These will be discussed in the context of authoring designer code.

Adding a Property

This is done as follows:

```
<class id="IPanels" parent="IDispatch"
            default="ControlDefault">
      ...
```

```
 <class id="IPanels" parent="IDispatch"
            default="ControlDefault">
      ...
```

	`<Property id="NumPanels" type="integer" />`

Authoring Designer Code

The rules below in a Custom IDF are instructions for gmBasic to author .NET Designer Code:

```
<Refactor id="[GM.mscomctl.ocx]" dllName="MscomctlMigration.dll" event="mscomctl">
<!--
**********************************************************
* ImageList Designer Code
**********************************************************
-->
   <migClass id="NetControl.ImageList"
             migName="System.Windows.Forms.ImageList"
             parent="ImageList"
   >
      <enumeration id="DesignPhase" >
         <Entry id="NewInstance"
                nPram="1"
                cshPattern="this.%1d = new System.Windows.Forms.ImageList(this.components);"
                vbnPattern="Me.%1d = New System.Windows.Forms.ImageList(Me.components)"
         />
      </enumeration>
      <property id="ImageSize"
                value="(ImageWidth,ImageHeight)"
                nPram="2"
                vbnPattern="New System.Drawing.Size(%1d, %2d)"
                cshPattern="new System.Drawing.Size(%1d, %2d)"
      />
      <property id="ImageStream" type="object"
         migStatus="external"
         value="SYM.name"
         nPram="3"
         cshPattern='this.%1d.%2d =
((System.Windows.Forms.ImageListStreamer)(resources.GetObject("%1d.%2d")));'
         vbnPattern='Me.%1d.%2d = CType(resources.GetObject("%1d.%2d"),
System.Windows.Forms.ImageListStreamer)'
      />
      <property id="Images" type="Collection"
         migStatus="external"
         value="ListImage.Key"
         nPram="5"
         cshPattern='this.%1d.%2d.SetKeyName(%4d, %5d);'
         vbnPattern='Me.%1d.%2d.SetKeyName(%4d, %5d)'
      />
      <property id="TransparentColor" type="Integer"
             default="System.Drawing.Color.Transparent"
      />
   </migClass>
```

The rules above direct gmBasic to author the following code block:

```
this.ilSmall.ImageSize = new System.Drawing.Size(16, 16);
this.ilSmall.ImageStream = ((System.Windows.Forms.ImageListStreamer)
(resources.GetObject("ilSmall.ImageStream")));
this.ilSmall.Images.SetKeyName(0, "genericSmall");
this.ilSmall.Images.SetKeyName(1, "fldrClosed");
this.ilSmall.TransparentColor = System.Drawing.Color.Transparent;
```

Sometimes, designer code depends on information other than what was available in the legacy code files. In these cases, it is possible to extend the designer code generation logic with custom code using the Great Migrations Scripting Language (gmSL). For example, certain StatusBar Panel attributes require special handling, as shown in the following examples.

1) Panel.Style indicates what should be displayed in the Panel, the most common being Text, but other commonly used indicate that a panel should show the state of the keyboard, current time, and current date. Only the Text style made the cut for WinForms StatusBar.
2) Panel.AutoSize indicates if a Panel should stretch to fill available space.

We add migStatus="external" to the enumeration so that the tool will retain the fully qualified name.

This is used to help the tool retain the original enum value during processing later.

```
<enumeration id="PanelStyleConstants"
                               migStatus="external">
      <entry id="sbrText" value="0"/>
      <entry id="sbrCaps" value="1"/>
      <entry id="sbrNum" value="2"/>
      <entry id="sbrIns" value="3"/>
      <entry id="sbrScrl" value="4"/>
      <entry id="sbrTime" value="5"/>
      <entry id="sbrDate" value="6"/>
      <entry id="sbrKana" value="7"/>
   </enumeration>
```

In the FileExplorer Example, "date" and "time" types are used. In the Custom IDF, we specify rules that will cause these types to become text panels and show the word "DATE" and "TIME" respectively. The next step is to implement logic to decide what string to display. This is done in a gmSL (GM Scripting Language) block at the bottom of the custom IDF.

```
<gmSL NameSpace="mscomctl" Class="MigCode" ><![CDATA[
   Sub Panel_BorderSides(ByVal Name As String)
      If LangIdent = "vbn" Then
         #TextStart
         Me.(%= Name %).BorderSides = DirectCast((( _
            (System.Windows.Forms.ToolStripStatusLabelBorderSides.Left Or _
             System.Windows.Forms.ToolStripStatusLabelBorderSides.Top) Or _
             System.Windows.Forms.ToolStripStatusLabelBorderSides.Right) Or _
            System.Windows.Forms.ToolStripStatusLabelBorderSides.Bottom),   _
            System.Windows.Forms.ToolStripStatusLabelBorderSides)
         #TextEnd
      Else
         #TextStart
         this.(%= Name %).BorderSides = ((System.Windows.Forms.ToolStripStatusLabelBorderSides)
            ((((System.Windows.Forms.ToolStripStatusLabelBorderSides.Left
               | System.Windows.Forms.ToolStripStatusLabelBorderSides.Top)
               | System.Windows.Forms.ToolStripStatusLabelBorderSides.Right)
               | System.Windows.Forms.ToolStripStatusLabelBorderSides.Bottom)));
         #TextEnd
         DecreaseMargin
      End If
      DecreaseMargin
   End Sub
   Sub Panel_Spring(ByVal Name As String, ByVal Value As String)
      If Value = "PanelAutoSizeConstants.sbrSpring" Then
         If LangIdent = "vbn" Then
            WriteLine "Me." & Name & ".Spring = True"
         Else
            WriteLine "this." & Name & ".Spring = true;"
         End If
      End If
   End Sub
   Sub Panel_Text(ByVal Name As String, ByVal Value As String)
      If LangIdent = "vbn" Then
         WriteText "Me." & Name & ".Text = "
         If Value = "PanelStyleConstants.sbrTime" Then
            WriteLine """TIME"""
         ElseIf Value = "PanelStyleConstants.sbrDate" Then
            WriteLine """DATE"""
         Else
            WriteLine """" & Name & """"
         End If
      Else
         WriteText "this." & Name & ".Text = "
         If Value = "PanelStyleConstants.sbrTime" Then
```

```
            WriteLine """TIME"";"
         ElseIf Value = "PanelStyleConstants.sbrDate" Then
            WriteLine """DATE"";"
         Else
            WriteLine """" & Name & """;"
         End If
      End If
   End Sub
]]></gmSL>
```

Preparing Resx File Contents

In some cases a control will store property values as binary data in an FRX file associated with the containing form. This binary data must be converted into an appropriate form for storage and ultimate consumption by the .NET replacement component: typically a base64-encoded data block in a RESX file. gmStudio has a facility for converting FRX data to RESX data. This facility is designed to be executed as a preparation step prior to running the translation process as described here: http://www.greatmigrations.com/pubs/gmStudio/Preparation_Resx.htm.

In order to tell gmBasic to use the RESX data generated by gmStudio you must add the Resx attribute to the compile command in your translation script:

```
   <Compile Project="%SrcPath%" Resx="%ResxFolder%">
```

This FRX-to-RESX facility is used by the FileExplorer sample to convert the graphics data (icons, bitmaps, etc.) associated with ImageList controls.

Adding a Control Type

In MSComCtlLib, there is only one Toolbar class, but it can present itself differently depending on its Style Setting. In WinForms, there are different types.

The FileExplorer sample uses three button styles. The Standard IDF includes a Toolbar Button coclass that corresponds to the default picture button (Style=0) albeit with a name change.

FileExplorer also uses Toolbar separator (Style=3) and MenuButton (Style=5). In COM, these different button objects were indicated using the Style property; in .NET, however, they are represented by different classes. In order to introduce and use these styles we add two new coclasses with subclass IButton. Adding these coclasses allows us to specify the rules for authoring them in .NET designer code. gmBasic handles selecting one of these based on the style setting in the code.

`<coclass id="Button" creatable="off">`
 	
`<subclass id="IButton"/>`
 	
`</coclass>`
	`<coclass id="Button" creatable="off"`
	`                migName="ToolStripButton" role="control" >`
	`      <subclass id="IButton"/>`
	`</coclass>`
	`<coclass id="Separator" creatable="off"`
	`                migName="ToolStripSeparator" role="control" >`
	`      <subclass id="IButton"/>`
	` </coclass>`
	`<coclass id="MenuButton" creatable="off"`
	`            migName="ToolStripDropDownButton" role="control"`
	
	`migStatus="Add1:DropDownItems,Add2:System.Windows.Forms.ToolStripItem">`
	`      <subclass id="IButton"/>`
	`   </coclass>`

Associating a sub-control collection with their parent control

Many of the controls in the MSComCtlLib API are composite controls: they have a top-level control as well as a number of sub-controls of some sort. For example a ListView has a Columns, Items, and SubItems collections; a TreeView has a Nodes collection; a StatusBar has a Panels collection. The replacement WinForms controls have similar structures. Each top-level control and its sub-controls must be migrated as a coherent entity. In addition, the sub-controls, must be programmatically added to their parent control later in the form initialization process. gmBasic uses the AddRange method to make this association. The name of the sub-control collection property and type of element in this control are specified using a migStatus attribute.

```
   <coclass id="StatusBar"
           migName="StatusStrip" role="control"
           migStatus="Add1:Items,Add2:System.Windows.Forms.ToolStripItem" >
```

This directs the tool to add the panels to the status strip using the following command

```
         this.sbStatusBar.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
            this.sbStatusBar_Panel1,
            this.sbStatusBar_Panel2,
            this.sbStatusBar_Panel3
         });
```

Using a Migration DLL

The migration of MSComCtlLib to System.Windows.Forms requires a few rules that cannot be completely specified solely by the declarative notation of a Custom IDF. In order to accommodate these rules in a flexible and extensible manner, gmStudio uses Migration DLLS. A migration DLL is a library of functions that extend the behavior of gmStudio. A Migration DLL is associated with a COM API in the Refactor section of the Custom IDF. The Refactor section may also include various code patterns to be used for authoring the .NET code. Although the details of creating and using Migration DLLs is beyond the scope of this document, the curious reader will find a portion of the Refactor element for MSComCtlLib below.

```
   <Refactor id="[GM.mscomctl.ocx]" dllName="MscomctlMigration.dll" event="mscomctl">
```

Furthermore, if you inspect the Custom IDF for MSComCtlLib, you will find a number of API members have migStatus attributes with special values; for example:

```
   <class id="IListView" parent="IDispatch">
      ...
      <property id="SortOrder" type="ListSortOrderConstants" status="InOut"
                migName="Sorting" migStatus="mscomctl" />
```

These migStatus values represent translation event context codes that are passed into the Migration DLL event handlers and may be used to direct processing in the body of the handler when it responds to events raised by the translation engine. By convention, if a migStatus attribute value is not one of the predefined migStatus enumeration entries (defined in the system configuration file enumerations.xml) then it is treated as a user-defined translation event context code.

Appendix X: Miscellaneous Topics

Multiple DLL versions

Large VB6 and ASP applications sometimes reference multiple versions of the same COM API. For example, there are several versions of the MSXML2 API in circulation – msxml2.dll, msxml3.dll, msxml4.dll, and msxml6.dll.

The registry-libname command in a Translation Configuration File allows multiple versions of a given DLL file to target a single standard as part of the migration process. For example, the following two commands tell gmBasic that references to msxml4.dll and msxml6.dll should be processed as if they were references to GM.msxml6.dll. This means that only the custom msxml6 IDF will be the only used in the upgrade process. The following directives

```
   <Registry type="libname" source="msxml4.dll" target="GM.msxml6.dll" />
   <Registry type="libname" source="msxml6.dll" target="GM.msxml6.dll" />
```

Progid aliases

Some COM APIs are accessed through classes that use non-standard ProgIDs. For example, the following three different ProgIDs all resolve to MSXML2.DomDocument:

- Microsoft.XMLDOM
- Msxml2.DOMDocument.3.0
- Msxml2.DOMDocument.4.0

These non-standard ProgIDs allow vendors to document their APIs using custom names and also allow developers to specify a specific version of the desired class. Some codebases may reference specific versions of a DOM class to meet stringent SCM and build procedures.

The registry-progid command allows non-standard or versioned classnames to be recognized as a specific class in the IDF. For example, the following commands will allow the translator to reference all three of the specified source ProgIDs as an alias for the specified target.

```
<Registry type="progid" source="Microsoft.XMLDOM"       target="MSXML2.DOMDocument" />
<Registry type="progid" source="Msxml2.DOMDocument.3.0" target="MSXML2.DOMDocument" />
<Registry type="progid" source="Msxml2.DOMDocument.4.0" target="MSXML2.DOMDocument" />
```

Late Libraries

In many cases you may find VB6 codes that use classes from libraries that are not explicitly referenced in the VBP file. These classes are loaded by name in the VB6 source code via a CreateObject() call:

```
       CreateObject("libname.classname")
```

In order to be able to migrate classes in these libraries, you need to tell gmBasic to which library is needed library. This can be done with registry-libname command that associates the library with the IDF. For example, the following command tells the tool that ProgIDs beginning with IWshRuntimeLibrary are described in wshom.ocx.xml.

```
<Registry type="libname" source="IWshRuntimeLibrary" target="wshom.ocx" />
```

Note that IWshRuntimeLibrary has an alias: Wscript. So you will also need a registry-progid command:

```
<Registry type="progid" source="WScript.Shell" target="IWshRuntimeLibrary.WshShell" />
```

Incremental COM Replacement

In the Tool-Assisted Rewrite Methodology, your migration process, and the code it produces, will proceed through a sequence of stages:

- Level 0: Source Code Complete, Consistent, and "Clean"
- Level 1: No translation warnings or errors (Single VBP/Single Page, Prototype Externals)
- Level 2: build complete (Single VBP/Single Page, Prototype Externals)
- Level 2+: build complete with reengineering (multi-VBP/multi-Page, .NET externals)
- Level 3: build complete with reengineering, verified functionally correct (multi-VBP/multi-Page, .NET externals)
- Level 4: ready for production (standard build and deployment processes)
- Level 5: production tested (rolled out and stable)

When you begin, you will be using the prototype strategy for externals (i.e., select BuildFile=on): all external COM APIs will be replaced by stub classes. This strategy significantly simplifies verifying the internal consistency and syntactic accuracy of translated code, however it does not produce a runnable code. In order to become runnable, stub classes must be replaced with a functional implementation. This replacement will rarely happen all at once, so you will have to move incrementally, developing and activating your migration rules for the libraries when you are ready. In order to facilitate an incremental transition from using prototype externals to .NET replacements, you must be able to indicate which libraries are ready to migrate and which one are not.

By default, select BuildFile=on suppresses the execution of migration rules. You can override this behavior by adding a Registry-idfstatus command to your script. For example, the following command tells gmBasic that it should perform migrations according to the specification in GM.msxml.dll.xml.

```
   <Registry type="idfstatus" source="GM.msxml6.dll" target="migrate" />
```

gmSL Scripts

gmSL scripts extend and alter the behavior of the gmBasic translator. gmSL scripts can manipulate the information about the source and target code at the lowest level: symbol tables and operation streams. gmSL scripts can also be used to extend the gmBasic translator to develop specialized analysis, reporting, and code generation tools. gmSL allows migration teams to make gmBasic do things that cannot be easily specified using the declarative refactoring statements.

gmSL scripts contain "handlers". These are subroutines invoked by the translator when various "migration events" occur during processing. There is a large set of predefined migration events as well as a facility for attaching migration events to the specific types and members in COM libraries and to specific application types and variables.

gmSL scripts are written using the Great Migrations Scripting Language (gmSL). gmSL is a C-like language with a few special features to facilitate template-based code generation. gmSL is also an extensive service-oriented, API that facilitates interacting with the translator and the system model in migration event handlers. The meta-programming concepts needed to develop gmSL scripts are somewhat advanced. We offer a gmSL Training Package for teams who want to develop gmSL in house.

gmSL is documented here: https://portal.greatmigrations.com/display/GMG/gmSLIntroduction
Over a dozen gmSL scripts are distributed with gmStudio.

Migration Dlls

Migration DLLs extend and alter the behavior of the gmBasic translator. Migration DLLs can manipulate the information about the system at the lowest level: symbol tables and operation streams. Migration DLLs can also be used to extend the gmBasic translator, for example, to develop specialized analysis, reporting, and code generation tools. Migration DLLs allow migration teams to make gmBasic do things that cannot be easily specified using the declarative refactoring statements or the gmSL.

Migration DLLs contain "handlers". These are subroutines invoked by the translator when various "migration events" occur during processing. There is a large set of predefined migration events as well as a facility for attaching migration events to the specific types and members in COM libraries and to specific application types and variables.

Migration DLLs can be coded with Visual Studio using C, Managed C++, C#, or VB.NET. The system programming techniques and meta-programming concepts needed to develop migration DLLs are somewhat advanced. We typically develop the Migration DLLs for our customers; but we also offer an SDK and Training Package for teams who want to develop Migration DLLs in house.

Migration DLLs make use of the Great Migrations Native Interface (gmNI). gmNI is a C-based API that supports and parallels the gmSL API.

gmNI is documented here: https://portal.greatmigrations.com/display/GMG/gmNIIntroduction
A few migration DLLs are distributed with gmStudio.