



Agile Business Suite External Classes

Calling out of AB Suite 2.0 for Windows (.NET)

White Paper

Applications generated and deployed with Agile Business Suite (AB Suite) 2.0 for Windows platform do not operate in isolation. They are required to communicate with other systems in order to meet various business requirements. Diverse systems and system interfaces are available in AB Suite. They might need to use or process data from other services or might provide data to downstream processes. To support this diversity, AB Suite 2.0 applications need the capability to communicate with other systems using different mechanisms.

There are various protocols and services available in AB Suite to communicate directly with other AB Suite or EAE systems. However, this whitepaper focuses on the External Class infrastructure that is used primarily to communicate with non-AB Suite systems and services.

Table of Contents

Calling out of AB Suite 2.0 for Windows (.NET)	4
Introduction	4
External Classes and External Libraries	4
Component Types of External Libraries	5
Configuring an External Class	6
.NET Assembly component configuration	6
COM component configuration	6
DLL library component configuration	7
EXE component configuration	7
Java Component Configuration	7
Shell Component Configuration	7
Configuring Methods and Parameters	8
Writing External Libraries for use with AB Suite	9
Writing a C# Assembly for use with AB Suite	9
Writing a VB.NET Assembly for use with AB Suite	13
Writing a C++\CLI DLL for use with AB Suite	16
Writing a C++ DLL for use with AB Suite	17
Writing a C++ Executable for use with AB Suite	19
Example 1: Calling a WebService using a C# assembly	20
External Library – a C# wrapper to a Web service	20
External Class - Configuring AB Suite to call the C# assembly	22
Putting it together – calling our Web Service from AB Suite	25
Example 2: Calling AB Suite as an External Library	25
External Class interface	25
External Library Definition	28
Debugging Tips	28
Debugging the External Libraries	28
Troubleshooting Tips	29

Calling out of AB Suite 2.0 for Windows (.NET)

Introduction

Applications generated and deployed with AB Suite 2.0 for Windows typically do not operate in isolation and need to communicate with other systems in order to meet business requirements. Systems may need to consume data from other services or they may themselves provide data to downstream processes. Given the diversity of systems and system interfaces available, AB Suite 2.0 applications need the capability to communicate with other systems using a variety of mechanisms.

Various protocols and services are available in AB Suite to communicate directly with other AB Suite or EAE systems. However, this document focuses on the **External Class** infrastructure that is used primarily to call out to non-AB Suite systems and services.

AB Suite external classes provide an interface that can be used to call out to external programs. External programs are programs, which are written in other languages that provide specific functionality to other programs including AB Suite. For example,

- A DLL might be created to handle domain authentication for user names and passwords.
- An executable might be used to perform file cleanup operations.
- A C# program might call web services to provide data from external sources

External programs improve the capability of AB Suite system significantly by allowing it to perform native operations and use data from external sources. External programs also help the AB Suite system to orchestrate various external systems to act with a single coordinated purpose.

External Classes and External Libraries

AB Suite allows classes to be defined where the implementation of that class exists outside of AB Suite.

These **External Classes** are normal AB Suite classes whose IsExternal property is set to true.

Interface type	None
IsConstant	False
IsExternal	True
Kind	Attribute

External Classes are proxies for the **External Libraries** that the AB Suite systems need to call. External Libraries are binaries that exist external to the AB Suite system, but typically reside on the same machine as the AB Suite system. Examples of External Libraries include DLLs, batch programs, executables, and other .NET assemblies.

External Libraries can be used to extend the functionality of an AB Suite system, or they can be used to make calls to other systems or services. One External Class is required for each External Library that is to be called. In some cases, multiple External Classes may be created to expose different interfaces of an External Library to AB Suite.

External Classes must be configured to match the External Library interface that is to be called. For example, if there is a function in an External Library DLL that will sort numbers, then the corresponding External Class must also have a method defined that matches the functional signature of the function in the DLL passing the sort method parameters to the External Library.

In general, AB Suite can pass simple and primitive type parameters to an external program through the methods defined, and can receive some returned data when the call is completed. However, the mechanism for passing data between AB Suite and external programs varies between external program types.

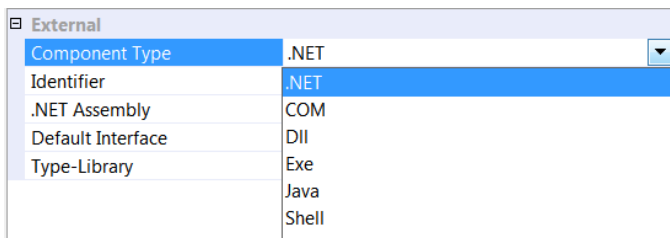
Note that external programs that pop-up on screens, or wait for user input, generally do not work well with AB Suite. This is mainly due to a change in behavior of newer Windows operating systems with respect to batch processes initiated from services interacting with the desktop. Since, this behavior varies between operating systems and can lead to inconsistent behavior, it is not

recommended to use pop-ups in an external program to request user input or display results in AB Suite.

In general, it is preferable to define and implement a working version of the External Library first, before attempting to build the External Class. This is because AB Suite's builder process often needs to refer to the External Library itself when generating the AB Suite application. However, placeholder or stubbed versions of an External Library will also suffice, allowing that actual implementation to be completed at a later stage. In practice, there are likely to be several iterations of External Class / External Library interfaces requiring changes on both sides, as the design evolves to meet changing requirements.

Component Types of External Libraries

External Classes have a 'Component Type' property to broadly define the component type of the External Library that implements its functionality. For a Windows platform configuration, the types that are available in AB Suite 2.0 are .NET, COM, DLL, Exe, Java. and Shell.



The screenshot shows a window titled 'External' with a table of properties. The 'Component Type' property is highlighted, and its dropdown menu is open, showing the following options: .NET, COM, Dll, Exe, Java, and Shell. The 'Identifier' property is also highlighted and set to '.NET'.

Property	Value
Component Type	.NET
Identifier	.NET
.NET Assembly	COM
Default Interface	Dll
Type-Library	Exe
	Java
	Shell

For example, the Component Type configuration property allows you to define whether you are trying to call a .NET assembly or an executable. This makes a difference, because AB Suite needs to generate wrappers internally to match the specific kind of interface that each of these component types present. For example, if the component type is an executable, then AB Suite will need to generate the infrastructure to pass in appropriate command line parameters and retrieve results from stdout. If you specify the component type to be a COM component, then AB Suite will need to generate the infrastructure to locate the component from the registry and use the methods exposed by the interface specified.

Following are the component types that are available in AB Suite 2.0.

.NET component type is used for calls to any .NET assembly. Assemblies may be written in any of the CLI supported languages including C#, C++\CLI and VB .NET. Assemblies needed to be constructed in a certain way (see below) so that the AB Suite Builder can locate and bind to them during the generate phase and also configured for the AB Suite Runtime to be able to locate and call them during runtime.

COM components are typically unmanaged components, although it is possible through the use of COM callable wrappers for .NET assemblies to be called as a COM component. Similarly, a COM component could use the Runtime Callable Wrapper to present itself as a .NET object (and therefore use the .NET component external class type), but as the AB Suite application is a COM+ application, it is usually more efficient to call COM components using the COM component external class type.

DLLs are unmanaged libraries hosted by another executable. AB Suite accesses code and data within a DLL through the exported functions provided by the DLL. DLLs are dynamically linked at runtime into the External Call Helper process, so the DLL needs to be designed with this in mind. Typically written in C++ or Visual Basic, many legacy libraries are still available as unmanaged DLLs.

EXEs are applications that can be called from the command line. AB Suite will need to locate the application at runtime, and will call it as a separate process, including 64 bit executables. Parameters are passed into the executable's command line, and results can be retrieved from the stdout via GLB.PARAM. It is possible to specify other command line callable applications such as batch files, by specifying the full name of the application, like Start.bat.

Java component type is not implemented. It was intended to allow AB Suite to call native java objects by generating the necessary wrappers, but this type is not available.

Shell component type is similar to the EXE type, but is a call to the command process of the Windows OS. Although no results are captured, it is useful for calling batch processes.

Configuring an External Class

For each external class you define, you also need to provide extra information to AB Suite, so that it can find the component (and in some cases the correct interface) for the external call. So, in addition to selecting the Component Type, you need to provide further details in the External Class' properties section.

Keep in mind that the External Classes in AB Suite can contain methods, variables, and method parameters like any other class in AB Suite. The methods that you add to an External Class must correspond to methods or functions in the External Library that implements the External Class. So, in most cases, the additional configuration is required to define exactly where those methods are located. Much like finding a street address, we have to first define some scope information. For a street address, we usually specify the city, suburb or locality where we might find a particular street. For methods defined in an external library, we need to specify the location/path of the component, and then where inside a component we would be able to locate those methods.

.NET Assembly component configuration

.NET component types require an additional Identifier, .NET Assembly path, the Default Interface, and optionally the Type-Library.

External	
Component Type	.NET
Identifier	CurrencyWs.Currency
.NET Assembly	C:\Demo\ECSolution\CurrencyWs\bin\Release\
Default Interface	ICurrency
Type-Library	

The **Identifier**, is the class identifier specified as <Namespace>.<Class>. We will discuss this in details below, but it is essential to supply the name of the class

that you are going to call in the assembly. It is used in both Build phase, and in Runtime.

The **.NET Assembly** path is the path where the Builder will be able to locate the assembly that you are trying to call. Builder needs to know where the assembly is in order to generate necessary wrappers. For the purposes of building your application, this assembly does not need to be registered with the Global Assembly Cache (GAC), but needs to be simply placed in an accessible folder location for the Builder.

The **Default Interface** is the name of the Interface, which contains the method that you are trying to invoke. If you have multiple methods in different interfaces, then you need to define a separate External Class for each interface. Currently, the Default Interface field is limited to interface names without '.' characters, so interfaces can only be used if they are in the same namespace as the class.

The **Type-Library** field is optional. You can supply a specific type library and AB Suite will extract information from the type library to find the necessary GUIDs to locate the class that implements the methods. If you do not supply a type library, AB Suite will generate one (through tibexp and regasm calls) and extract the details from that instead. This has implications on how you specify methods in your .NET assembly. Since, AB Suite uses the type library information, only classes and methods that are defined in a type library, will be found. If certain classes are not com-visible, then they will not be automatically made available in a type library, and thus will be invisible to AB Suite.

COM component configuration

COM components do not require a .NET assembly path, as the COM objects are registered with the system's COM admin catalog and are available via its ProgID. The COM object does not need to be registered on the Build machine, but will need to be registered on the Runtime server for it to be available to AB Suite. Similar to the .NET Assembly external class type, the Identifier and Default Interface configuration properties are required while the Type-Library property is optional.

External	
Component Type	COM
Identifier	MarketPrices.Price.1
.NET Assembly	
Default Interface	IPrice
Type-Library	

The Identifier contains the ProgID of the COM component. Make sure that it is exactly specified. If you change the external library COM component's Prog ID for example, if there's been a version change, you need to also make the same change in the External Class' Identifier property, or you may end up calling the old version (or it may no longer be a valid Prog ID on the machine).

The Default Interface property specifies the dispatch interface that contains the methods to be called. Similar to the .NET Component Type, if you are using methods from different interfaces on your component, then you need to define a different External Class for each interface.

The Type-Library field is optional. If it is supplied, then AB Suite attempts to use it when generating the external class wrapper. This gives you the opportunity to define GUIDs or to customize the type library with various attributes. If this property is left blank, AB Suite will generate its own type library (through tlbexp and regasm calls).

DLL library component configuration

DLLs are libraries that are hosted by the External Call Helper component running on the Runtime host server. External classes that represent a DLL simply need to have their Identifier set.

External	
Component Type	Dll
Identifier	DeliveryPeople
.NET Assembly	
Default Interface	IUnknown
Type-Library	

The Identifier property specifies the name of a particular DLL. This DLL needs to be available on the host server's PATH environment variable, and is usually placed in the AB Suite generated application's bin directory for ease of access. Note that the Identifier does not need to have its extension specified, as AB Suite will assume it is a .DLL extension.

EXE component configuration

Standalone executables are configured in a similar way to DLLs.

External	
Component Type	Exe
Identifier	Discounter.exe
.NET Assembly	
Default Interface	IUnknown
Type-Library	

The Identifier configuration property specifies the name of the executable that AB Suite will call. The name of the executable should include the extension file extension as well.

Java Component Configuration

This component type is not fully implemented and will be removed in subsequent releases.

Shell Component Configuration

Shell components are almost identical to executable types and are specified the same way as EXE and DLL types.

External	
Component Type	Shell
Identifier	ThisCall.bat
.NET Assembly	
Default Interface	IUnknown
Type-Library	

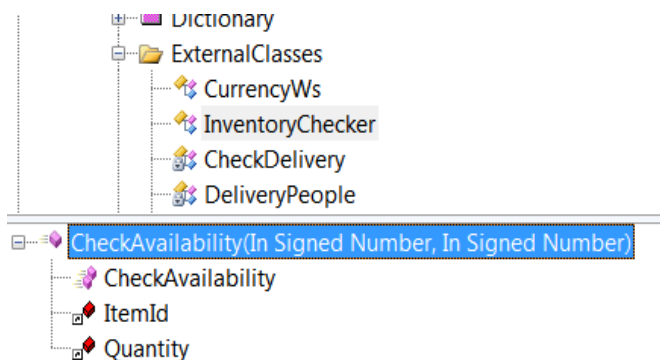
The identifier property specifies the name of the command line processor command to invoke. Since, it's possible to invoke executables on the command line, it's also possible to call executables using the Shell component type.

Note that at this point, there is still further configuration required to specify the method and parameters that an AB Suite application will use. However, it is recommended that you read the subsequent section about how methods and parameters are defined in the language of your External Library first.

Configuring Methods and Parameters

External Classes use methods to represent the methods, functions and command in the matching External Library. For example, if there is a method on the external .NET assembly that allows you to 'CheckAvailability' on particular stock items, you will need a matching 'CheckAvailability' defined in the External Class. This method must be defined with a matching functional signature as the one on the External Library. For example, if there are two 'int' parameters to the method in the External Library, then the method in the External Class, must also have two numeric parameters.

A method is added to an External Class in the same way methods are added to regular AB Suite classes. Right-click on the External Class, select Add >Method and provide it with an appropriate matching name.



In the example above, a method called 'CheckAvailability' is added to the 'InventoryChecker' External Class. This method has two Signed Number parameters. One is called ItemId and the other is called Quantity. Parameters are added by Right-Clicking on the method, selecting Add > Parameter, then providing the parameter with an appropriate name and type properties.

Where there are multiple parameters in a method, the Sequence property will determine their order in which they are passed to the corresponding method. So, in the above example, if the 'ItemId' is expected to be the first parameter, it will have a Sequence of 1. The second parameter 'Quantity' will have a Sequence of 2. While these Sequence numbers may be correct already if you added the parameters in their correct order to the method, it is a good idea to check that the Sequence numbers are correct, to avoid passing the data to the wrong parameter.

Note that methods of External Classes can also have a return variable. You can add a return variable to any External Class method by Right-Clicking on the method and selecting Add...Return Variable. The return variable will appear as having the same name as the method to which it belongs. In the example above, the return variable is also called CheckAvailability, and is listed first under the method. The return variable does not appear in the list of parameters (which only contains two numeric parameters in the example), but is still part of the method's functional signature.

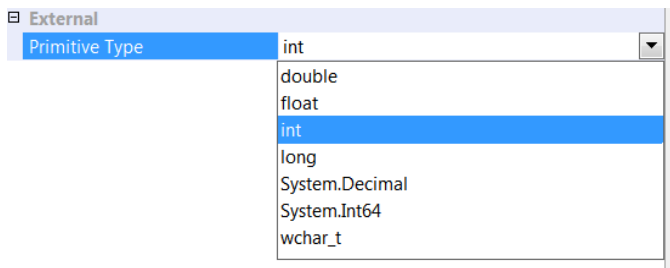
The matching External Library, will be expected have a method called 'CheckAvailability' which will expect 2 numeric parameters, ItemId and Quantity, and it will also be expected to have a return parameter.

It is important to ensure that the functional signature that you define in the External Class exactly matches that of the method/function or command that you are trying to get AB Suite to invoke in the External Library.

To help AB Suite match the parameter and return types more accurately, External Class method parameters and return variables have additional configuration properties that tell AB Suite the kind of external data type it should use when trying to pass information to the External Library.

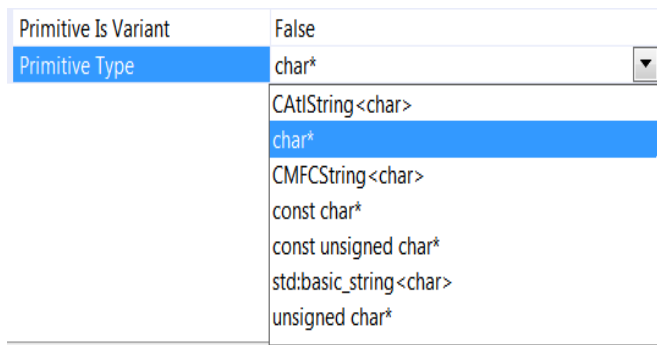
Number parameter types can be associated with one of a small number of common external numeric types. The primitive types available will vary depending on the type of External Class that you have specified. For example, a Signed Number in a .NET External Class, can be

associated with double, float, int, long, System.Decimal, System.Int64 and wchar_t types.



For a Number parameter on an unmanaged External Class type (for example DLL), additional types that can be used, include CURRENCY, DECIMAL and various unsigned datatypes. AB Suite tries to provide appropriate primitive type options based on the context of the External Class type.

Strings in .NET External Class types are usually mapped only to System.String type, but String parameters in unmanaged External Class can be mapped to a wider range of external string types. These include CATLString<char>, char*, std::basic_string and others (see below).



The variety of external primitive types that are available makes it easier to match the External Library's method signatures exactly and ensure that the data being passed to and from that library are not inadvertently truncated or converted incorrectly.

It is important to ensure that you choose the correct primitive types when defining the method and parameters for your External Class. This is easiest when you have access to the actual source code of the

External Class, but this information can usually be obtained from the vendor of the External Class, or from metadata supplied with the class.

Writing External Libraries for use with AB Suite

As noted in the previous sections, External Libraries implement the functionality that is defined by the External Class in AB Suite. There are, however, some limitations to the kinds of External Libraries that can be used, as well as some rules that should be followed when installing or configuring the External Libraries. There are significant differences in these rules between different External Library component types, for example, whether it is a .NET assembly, or a DLL or EXE.

Writing a C# Assembly for use with AB Suite

This section describes the process for writing a C# assembly. It presumes a working knowledge of C#, and covers some of the rules that must be followed, for the assembly to be available for use with AB Suite. While this is mostly applicable for developers who are also writing a C# external library for their application, it can be useful information for getting 3rd party provided assemblies to work.

The Interface

AB Suite views .NET assemblies through the type library (.tlb) that can be obtained from calls to the Type Library Exporter utility (tlbexp.exe). Type libraries can be specifically provided (by specifying it in the configuration properties), or they will be generated automatically for you. The type library defines the interface that the AB Suite will use at build time to generate the underlying C# caller code, and also at runtime to locate the actual assembly and method.

C# assemblies therefore need to specify an interface for the classes and methods that they wish to expose to AB Suite. *Correctly defining the interface in C# is central to ensuring your AB Suite application can call the assembly.*

The assembly's interface is defined using the interface keyword in C#, for example:

```
public interface IMathsClass
{
    void Add(int Operand1, int
Operand2, out int Result);
    int Subtract(int Operand1,
int Operand2);
}
```

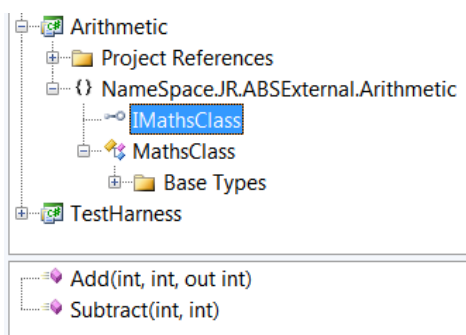
Note that the interface is marked as public. This allows the Type Library Exporter to expose the interface for AB Suite to use.

The interface also contains the Methods that will be exposed. Since the interface is public, it is advisable to specify only methods that are required by AB Suite, and that these methods are designed with public access in mind. We will discuss the functional signatures of the methods in more detail later.

The name of the Interface is the value that should be specified in the External Class' Default Interface configuration property, as mentioned:

External	
Component Type	.NET
Identifier	NameSpace.JR.ABSExternal.Arithmetic.MathsClass
.NET Assembly	C:\TEMP\ABSCallOutEx\CsharpExternal\bin\Deb
Default Interface	IMathsClass
Type-Library	

The interface and its methods can be more clearly seen in the Class View, as shown below.



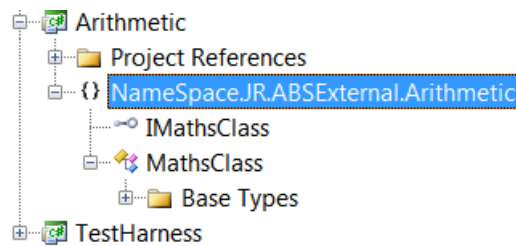
While C# allows the interface to belong to another Namespace, AB Suite currently can only use interfaces that are in the same namespace as the class.

Note: The Default Interface property cannot currently accept the '.' character, which makes it impossible to refer to an interface in a nested namespace.

The Namespace

A quick look at the example assembly in Class View will reveal that a namespace is used. The namespace is important because AB Suite will use it for name resolution purposes when locating the interface. It also forms part of the Identifier configuration property, so it is also used to properly scope the Class that implements the methods. Declaring your interface and classes within a globally and uniquely identifiable namespace is highly recommended (although classes and interfaces that belong to the default global namespace can also be used). Without a unique namespace declaration, it is possible for another class or interface of the same name to clash with the one you are providing in your assembly.

Note the Namespace of the example assembly below,



And the corresponding namespace used in specifying the Identifier property:

External	
Component Type	.NET
Identifier	NameSpace.JR.ABSExternal.Arithmetic.MathsClass
.NET Assembly	C:\TEMP\ABSCallOutEx\CsharpExternal\bin\Deb
Default Interface	IMathsClass

A namespace is defined in C# using the namespace keyword, usually at the very beginning of the .cs file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices; //
Needed for ABS EXT classes
using System.Text;

namespace
NameSpace.JR.ABSExternal.Arithmetic
{
```

With the Namespace and the Interface defined, we can turn our attention to the code that implements the functionality.

Note: The System.Runtime.InteropServices namespace is also required for managed External Libraries.

The Class

The methods that actually implement the functionality are contained within a Class. The class needs to:

- be Public
- implement the interface defined above
- belong to the same namespace as the interface

These rules are required to allow AB Suite to correctly locate the class in runtime. The public access modifier is needed as the AB Suite generated application is essentially an external application. The interface implementation tells AB Suite that this class indeed implements the interface that was defined in the type library. The namespace rule is required because AB Suite builder assumes the same namespace for the interface and the implementing class.

The class can be included in the same C# file, and included in the same namespace, as below.

```
namespace
Namespace.JR.ABSExternal.Arithmetic
{
    public interface IMathsClass
    {
        void Add(int Operand1, int
        Operand2, out int Result);
        int Subtract(int Operand1, int
        Operand2);
    }
    public class MathsClass : IMathsClass
    {
    }
}
```

The name of the class is used as the suffix in the Identifier property:

External	
Component Type	.NET
Identifier	ameSpace.JR.ABSExternal.ArithmeticMathsClass
.NET Assembly	C:\TEMP\ABSCallOutEx\CsharpExternal\bin\Del
Default Interface	IMathsClass
Type-Library	

So the correct Identifier for a .NET external class type is <Namespace>.<Class Name>. Compare this with how the Namespace property is used in the Identifier property on the previous page.

It is important to note that the Identifier is the class that will be instantiated by AB Suite when attempting the external call. If the class has no namespace and belongs to the global default namespace instead, then the identifier should simply be <Class Name>. Conversely, if the class belongs to a nested namespace, then the identifier will be <Namespace>.<Nested Namespace>.<Class>.

Once the class is defined, you can add the implementation of the methods into the body of the class.

The Method

To recap, methods that are exposed to AB Suite (via the interface), are implemented inside the Class, all of which live inside a namespace.

It is important to note that in AB Suite 1.2 and 2.0, methods can only contain primitive parameters. This is because AB Suite does not have the ability to import and use custom or complex types. In practice, this limits method parameters to numbers, strings, and boolean types. The only way to use custom or complex types is to write code that marshals primitive types to the complex types manually.

Similarly, the return type of methods should only be primitive types (void return types are permissible).

A simple example of methods that use primitive types was defined in the Interface section above, and is shown in fully implemented form within the class, as shown below:

```
namespace
Namespace.JR.ABSExternal.Arithmetic
{
    public interface IMathsClass
    {
        void Add(int Operand1, int
        Operand2, out int Result);
        int Subtract(int Operand1, int
        Operand2);
    }
}
```

```

public class MathsClass : IMathsClass
{
    public void Add(int Operand1, int
Operand2, out int Result)
    {
        Result = Operand1 + Operand2;
    }
    public int Subtract(int Operand1,
int Operand2)
    {
        return Operand1 - Operand2;
    }
}

```

Note that both the methods are public. This allows external programs (such as AB Suite) to call the methods.

Method Parameters

Method parameters and method return types must be primitive. This is mainly due to the need to marshal data to and from the COM data types that are used in the type libraries which described the interface. A list of COM data types is available at:

<http://msdn.microsoft.com/en-us/library/sak564ww.aspx>

AB Suite allows you to specify an external type for the method parameters. For example, if you define a method in your External Class of type Number, then you will be able to further specify whether it is an int, or a long, or one of several other numeric primitive types.

Returning Data to AB Suite

The first method returns a result via a method parameter, which has an 'out' qualifier.

```

public void Add(int Operand1, int
Operand2, out int Result)

```

An 'out' qualifier is required if data is to be returned to AB Suite via a method parameter. Note that the 'out' qualifier is distinctly different from a 'ref' qualifier, which poses a problem for .NET languages that do not have the concept of an 'out' parameter qualifier. We will see later, that the matching parameter on the External Class' method will have a matching direction of 'out'.

The second method returns a value via the method return.

```

public int Subtract(int Operand1,
int Operand2)

```

This is matched on the External Class method by the return variable (see above section regarding return variable definition). As with the method parameters, you can only return primitive types, such as numbers, strings and Boolean types. For languages that do not have an 'out' qualifier, this would be the recommended way of returning data to AB Suite.

Assembly Configuration

Finally, there is some additional configuration for the assembly, required to allow AB Suite to extract the necessary information from the type library.

COM visibility

If AB Suite needs to generate the type library (that is, a tlb file is not provided in the External Class configuration properties), then the assembly needs to be made COM visible. This can be set globally in the Assembly Information properties window.

Assembly Information

Title: CSharpExternal

Description:

Company: Microsoft

Product: CSharpExternal

Copyright: Copyright © Microsoft 2012

Trademark:

Assembly Version: 1 0 0 0

File Version: 1 0 0 0

GUID: fc890b39-b3b2-49d5-b91d-90587e37c

Neutral Language: (None)

☒ Make assembly COM-Visible

OK Cancel

Or can be manually added to the AssemblyInfo.cs

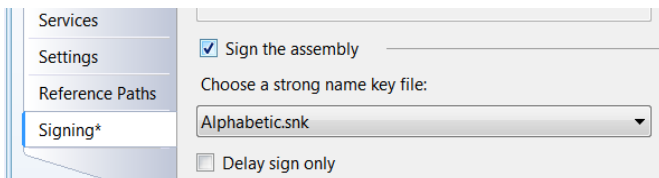
```
// Setting ComVisible to false makes the
// types in this assembly not visible
// to COM components. If you need to
// access a type in this assembly from
// COM, set the ComVisible attribute to
// true on that type.
[assembly: ComVisible(true)]
```

It is also possible to set the COM visibility at a lower level of granularity to control the interfaces, classes, and methods that will be exposed to AB Suite. If you have parts of an assembly that you wish to avoid exposing, then setting COM visibility to true only for the methods that you wish AB Suite to use is a better option. You can make a specific interface (or class, or method) to be COM visible by adding the ComVisible attribute as below.

```
[ComVisible(true)]
public interface IMathsClass
{
```

Signing

AB Suite generated application assemblies have strong names, and due to C# rules (see <http://support.microsoft.com/kb/313666>) any referenced assemblies must also have strong names. Therefore, any assemblies you use for AB Suite must also be signed. You can sign your assembly by providing a Strong Name Key file (the snk file) and specifying it in the Signing properties for your assembly.



Note: The snk files can be easily created using the sn.exe utility provided with Visual Studio. See [http://msdn.microsoft.com/en-us/library/k5b5tt23\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/k5b5tt23(v=vs.71).aspx) for details. Some businesses may already have their own snk files already

generated, which can be used without having to generate a new strong name key.

Deploying the assembly for Runtime

After you building the C# assembly successfully, you will need to consider how to deploy it for the Runtime. In most cases, the Runtime server will be a different machine to the one used during the development of the assembly. So the built assembly will need to be copied over to the runtime server and sometimes registered.

You can copy the built assembly manually, or make use of Visual Studio's in-built post-build events to copy the updated assembly automatically. For example:

Command Line	copy "\$(TargetPath)" C:\NGENSystems\Sort\Release_...
Description	
Excluded From Build	No

Setting up the assembly on Runtime

Note that the Runtime will look for the assembly using Windows' standard assembly location rules. Once you have copied the assembly to the Runtime server, you can help AB Suite Runtime locate the assembly by either adding the path to the copied assembly, to the PATH environment variable on the machine, or you could register the assembly in the Global Assembly Cache (GAC).

Registering the assembly in the GAC can make the assembly easily accessible for other programs on the machine and requires simple utilities (such as GACUtil.exe, available in the .NET Framework 2.0 SDK), without needing to specifically configure your environment to locate the assembly. While this may make deployment more flexible, you need to decide whether it is preferable to specify the path to the assembly explicitly.

Writing a VB.NET Assembly for use with AB Suite

VB.NET assemblies are handled in a very similar way to C# assemblies. The assembly you build will be accessible to AB Suite as a .NET type External Library. .NET type External Libraries also need to have interfaces, namespaces, classes and functions defined.

The Interface

VB.NET interfaces are defined using the Interface keyword, for example:

```
Public Interface ISelectionSort
End Interface
```

Note that the interface is marked as public. This allows the Type Library Exporter to expose the interface for AB Suite to use.

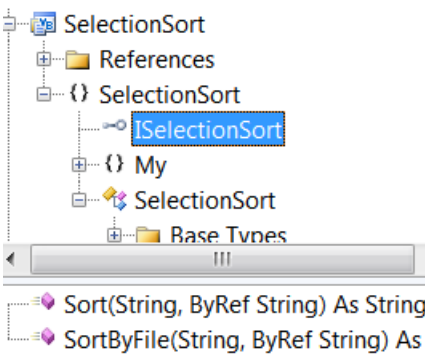
The interface also contains the functions that will be exposed. As with all interfaces, only add functions to the interface that are to be used by external programs.

```
Public Interface ISelectionSort
    Function Sort(ByVal input As String,
ByRef output As String) As String
    Function SortByFile(ByVal inputFile
As String, ByRef output As String) As
String
End Interface
```

The name of the Interface (ISelectionSort in this example) is the value that should be specified in the External Class' Default Interface configuration property, as below:

External	
Component Type	.NET
Identifier	SelectionSort.SelectionSort
.NET Assembly	C:\Users\LeeA2\Documents\Visual Studio 2008\
Default Interface	ISelectionSort
Type-Library	

The interface and its functions can be more clearly seen in the Class View, as shown below.



The Namespace

The Namespace property in a VB.NET application is set in the project properties. This namespace provides context for the interface and for the class, and is used when AB Suite needs to build the necessary external class wrappers.

Assembly name:	Root namespace:
SelectionSort	SelectionSort

And the corresponding namespace is used in specifying the first part of the External Class' Identifier property:

External	
Component Type	.NET
Identifier	SelectionSort.SelectionSort
.NET Assembly	C:\Users\LeeA2\Documents\Visual Studio 2008\
Default Interface	ISelectionSort
Type-Library	

The Class

The definition of a VB.NET class must also follow the same rules as any other managed assembly:

- be Public
- implements the interface defined above
- belong to the same namespace as the interface

These rules are required, to allow AB Suite to correctly locate the class in runtime.


```
Public Class SelectionSort
    Implements ISelectionSort
```

The name of the class is used as the suffix in the External Class' Identifier property:

External	
Component Type	.NET
Identifier	SelectionSortSelectionSort
.NET Assembly	C:\Users\LeeA2\Documents\Visual Studio 2008\
Default Interface	ISelectionSort
Type-Library	

As with the C# assemblies, the Identifier for a VB.NET external class type, is specified as <Namespace>.<Class Name>.

Once the class is defined, you can add the implementation of the functions into the body of the class.

The Function

VB.NET functions can also only use primitive types to pass values to and from their matching External Class.

A simple example of functions that use primitive types was seen in the Interface section above, and is shown in fully implemented form within the class, below.

```
Public Class SelectionSort
    Implements ISelectionSort
    Public Function Sort(ByVal input As
String, ByRef output As String) As String
    Implements ISelectionSort.Sort
        output = "all sorted"
        Return output
    End Function
    Public Function SortByFile(ByVal
inputFile As String, ByRef output As
String) As String Implements
ISelectionSort.SortByFile
        output = "input file was" +
inputFile
        Return output
    End Function
End Class
```

Note that both the methods are public and implement functions defined in the interface. As the class implements the interface, the class must also contain the implementation of all the functions that you exposed in the interface.

Function Parameters

As mentioned previously, functions can only use primitive types.

One key difference for VB.NET functions from their C# equivalents is the unavailability of the 'Out' direction attribute in VB.NET. The only available parameter attributes, are by value (ByVal) and by reference (ByRef). This means that you can only use directions of 'In' and 'InOut' for any parameters you wish to pass to a VB.NET assembly.

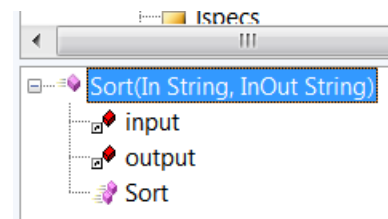
Use direction 'In' to pass data to a function declared as passing 'ByVal'.

Use direction 'InOut' to pass data to and from a function declared as passing 'ByRef'.

Therefore, the function previously defined:

```
Public Function Sort(ByVal input As
String, ByRef output As String)
```

is matched by a method on the External Class with one In String parameter and one InOut String parameter.



The function also can return data through a return variable. In this example, the function is defined 'As String'. This allows any return data to be captured by a String type return variable in the External Class.

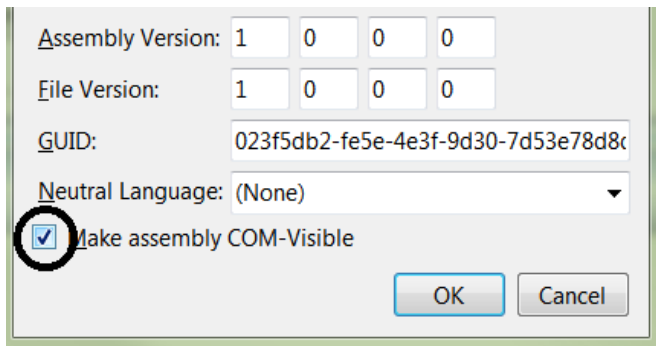
```
Public Function Sort(ByVal input As
String, ByRef output As String) As String
```

Assembly Configuration

Finally, similar additional configuration for the assembly is also required to allow AB Suite to extract the necessary information to communicate with a VB.NET assembly.

COM visibility

If AB Suite needs to generate the type library (that is, a tlb file is not provided in the External Class configuration properties), then the assembly needs to be made COM visible. This can be set globally in the Assembly Information properties windows.

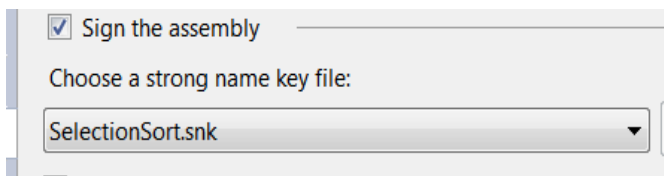


Or can be manually added to the class declaration:

```
<Microsoft.VisualBasic.ComClass()> Public  
Class SelectionSort
```

Signing

As with C# assemblies, you need to sign the VB.NET assembly using the sn.exe utility, since AB Suite assemblies are also strongly named.



Deploying the assembly for Runtime

Deployment and Runtime setup of the built VB.NET assembly is done the same way as any other managed assembly. See the C# section above for details. Post-build events are available and are recommended for automating the deployment and distribution process.

Writing a C++\CLI DLL for use with AB Suite

C++\CLI are another example of managed .NET programs that you can use with AB Suite. As with all .NET assemblies, they are accessed as a .NET component type External Class, and have the same configuration properties.

The Interface

The Interface of a C++\CLI assembly should be defined as public using the interface keyword.

```
public interface class ISort  
{
```

The interface name is used in the Default Interface property on the External Class.

External	
Component Type	.NET
Identifier	MergeSort.MergeSort
.NET Assembly	C:\Users\LeeA2\Documents\Visual Studio 2008\
Default Interface	ISort
Type-Library	

The Interface belongs to a namespace, defined using the namespace keyword.

The Namespace and Class

```
using namespace System;  
using namespace  
System::Runtime::InteropServices;  
  
namespace MergeSort  
{  
    public interface class ISort  
{
```

Note: The System::Runtime::InteropServices namespace is required for assemblies used by AB Suite.

In this example, the class is also named MergeSort, and you can see the declaration below

```
public ref class MergeSort : ISort  
{  
    // methods declared here  
};
```

So the Identifier, is again <Namespace>.<Class> which becomes MergeSort.MergeSort. Remember that the Identifier relates to the class within the assembly which needs to be instantiated, and may not always be in a namespace of the same name (it can even have no namespace!).

Methods

Methods in C++\CLI require the use of hat (^) operators when declaring a reference to managed objects, such as strings.

```
virtual String^ Sort(System::String^
input, System::String^ result);
```

Directional attributes are also required if you wish to pass data back to AB Suite via method parameters.

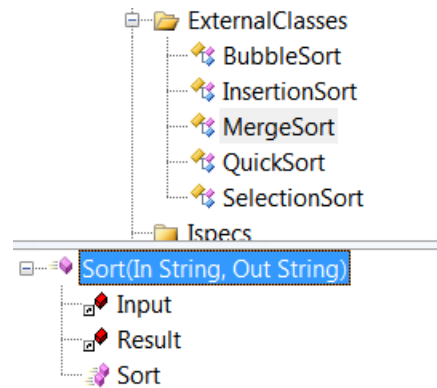
Parameters that pass data back to an External Class parameter with direction Out, must be declared with an 'Out' attribute, as well as a tracking reference (which specifies a parameter is 'pass by reference').

Thus, if the second parameter in the method above is an 'Out' parameter, it should be declared as follows in C++\CLI:

```
virtual String^ Sort(System::String^
input,
[System::Runtime::InteropServices::Out]
System::String^% result);
```

An 'Out' attribute is not required in C++\CLI if the parameter is 'InOut'. For InOut External Class method parameters, only the tracking reference is needed for the C++\CLI method.

The Sort method above would have a corresponding External Class method signature shown below. A method called Sort with one String 'In' parameter and one String 'Out' parameter.



As with other methods in managed External Libraries, the return variable can also be used. You can see it defined in the External Class method above. In C++\CLI, the return parameter is defined as a reference to a String managed object.

```
virtual String^ Sort(System::String^
input,
[System::Runtime::InteropServices::Out]
System::String^% result);
```

Assembly Configuration

Similar Assembly Configuration rules apply for C++\CLI assemblies. COM Visibility must be set to true, either at an assembly level, or via class or method level attributes. Assemblies must also be signed, as AB Suite itself generates strongly names assemblies. See the C# assembly section above for details.

Deploying the assembly for Runtime

Finally, the same deployment and runtime configuration recommendations described for C# assemblies, also apply to C++\CLI ones.

Writing a C++ DLL for use with AB Suite

Unmanaged DLLs require little in the way of configuration, and only need to be available on the runtime host's PATH. The functions within a DLL need to be exposed (or exported) for external usage, and the corresponding External Class' methods must match the functional signature of the function being called.

The Entry Point

Visual Studio unmanaged C++ DLLs contain much of the required structures for a DLL. The entry point for the DLL is added by default.

```
// dllmain.cpp : Defines the entry point
for the DLL application.
#include "stdafx.h"
```

```
BOOL APIENTRY DllMain( HMODULE hModule,
                      DWORD
ul_reason_for_call,
                      LPVOID lpReserved
                      )
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}
```

The Identity property of the External Class, is the name of the DLL, without the .dll extension. You can easily find this by looking up the Output File property in the DLL's project properties.

Common Properties	Output File	\$(OutDir)/DeliveryPeople.dll
Configuration Properties	Show Progress	Not Set
General	Version	
Debugging	Enable Incremental Linking	No (/INCREMENTAL:NO)
C/C++	Suppress Startup Banner	Yes (/NOLOGO)
Linker	Ignore Import Library	No
General	Register Output	No

In this example, we would set the Identity to 'DeliveryPeople'.

External	Component Type	Dll
	Identifier	DeliveryPeople
	NET Assembly	

The Functions

Functions called by AB Suite need to be exposed. Visual Studio provides the necessary wrappers when you

specify an exported function of a DLL. The header file would contain `__declspec` declarations, and the declaration of the exported function.

```
#ifdef DELIVERYPEOPLE_EXPORTS
#define DELIVERYPEOPLE_API
__declspec(dllexport)
#else
#define DELIVERYPEOPLE_API
__declspec(dllimport)
#endif
```

```
// This is an example of an exported
function declaration
DELIVERYPEOPLE_API int
GetDeliveryTime(int ItemID, int
ItemAmount, char* DeliveryCountry);
```

Function Parameters

Note: Only primitive parameter types can be used with AB Suite.

More so than managed External Library methods, it is necessary to note the exact parameter types that are being used in the function, as you will need to set the External Class method parameters to the same matching types.

For example, the 3rd parameter in the example function is a `char*` parameter. The configuration property for the 3rd parameter in the corresponding method in the External Class is also given a primitive type of `char*`.

External	Primitive Is Variant	False
	Primitive Type	char*

Note that the same functional signature should be used if you are intending to implement the exported function:

```
// This is an example of an exported
function implementation.
DELIVERYPEOPLE_API int
GetDeliveryTime(int ItemID, int
ItemAmount, char* DeliveryCountry)
{
    // Some Code
}
```

No out attributes are available for C++. Similar to VB.NET functions, passing by value (In) and by

reference (InOut) are the only available options. However, similar to other External Library types, a C++ function can make use of the return variable.

Deploying the assembly for Runtime

Finally, the same deployment and runtime configuration recommendations described for C# assemblies, also apply to unmanaged DLLs. The DLL needs to be copied to the runtime host, and available to AB Suite through the PATH environment variable. Use of Visual Studio's post-build events to automate the deployment process is recommended.

Writing a C++ Executable for use with AB Suite

Executables include any program that can be run on a command line. So, it covers a very large range of possible program types available for the Windows operating system. However, not all executables are suitable for use with AB Suite. AB Suite calls executables via the External Call Helper component, which instantiates a new process to make that call on behalf of the caller. So, executables will run as a separate process with the user's credentials, and most crucially - on the runtime host.

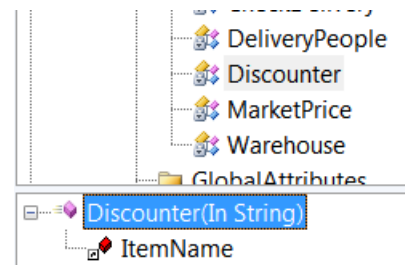
This means that an executable will run without being seen by the calling client. If there is any input required by the executable, then the user will generally not be able to provide that input, unless it was provided during the initial call, through the command line. Popup messages, and dialog boxes will generally not be accessible – in many cases Windows prevents pop up dialogs from appearing on server processes. Any executable processes which do require user interaction, will appear to hang.

Also, since the executable is being called on behalf of the caller, the executable will run with the same Windows operating system credentials as the caller. If the caller does not have privileges to perform the work, or to make the specific call, or access required resources for the executable, then the call will fail with access denied errors. An example is where non-privileged Anonymous users are used to access the runtime system. Since the Anonymous user is not privileged, the external call to access a privileged

resource (for example the registry), will not succeed. Keep this in mind when designing external calls, and ensure that the privileges to make the call will actually be available to the calling user.

The Command Line interface

The 'interface' for an executable, is its command line interface. This command line interface can accept command line parameters. In AB Suite, the interface is represented by a Method, while the command line parameters are represented by a single String parameter in the Method.



Multiple parameters can be added to the AB Suite external class' method, but they are still passed to the executable as a single concatenated string. Note that the direction of command line parameters is always 'In', as the parameters are passed in by value only.

Returning Data

A typical command line interface for an executable in C++ is defined as below:

```
int _tmain(int argc, _TCHAR* argv[])
{
```

The arguments are passed in by the operating system's command processor, with 'argc' containing the number of parameters (including the executable name itself), while the argv array contains the parameters themselves. Note that by default, the executable returns an integer return value.

```
int _tmain(int argc, _TCHAR* argv[])
{
```

Executables can also return values via the stdout and stderr handles. AB Suite can capture the Standard Output in the GLB.PARAM value. So for example, a

value normally displayed on the command line as a result of the following printf command:

```
printf("%d", discount);
```

can be captured in the GLB.PARAM. Note however that this functionality is not working correctly as of 2.0.1600, and needs to be rectified.

Example 1: Calling a WebService using a C# assembly

A common scenario for external classes, is to call (or consume) a web service. AB Suite is unable to reference web services natively, and must rely on custom code to be written which passes the data to and from the web service and AB Suite. We will go into detail about how to create a C# External Library to consume a currency exchange web service below. A basic knowledge of C# programming, and familiarity with the C# External Library section above, is assumed here.

External Library – a C# wrapper to a Web service

We will first look at the External Library – the C# assembly which does the actual work of calling the external web service, and retrieving results. AB Suite will call the External Library through an External Class, which we will describe separately.

The Namespace and Interface

The namespace of the External Library will be used to provide scope for the class and the interface.

```
namespace CurrencyWs
{
}
```

The interface is declared within the namespace and is made public. By convention, we prefix the interface name with an 'I' to indicate that it is an interface.

```
namespace CurrencyWs
{
    public interface ICurrency
    {
    }
}
```

Referencing the external Web service

We will also complete the interface, by specifying the method which AB Suite will use to call the web service.

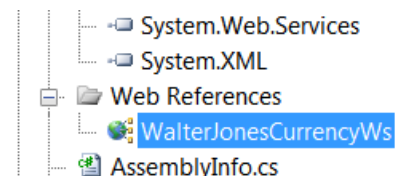
For this, we need a little bit more detail about the web service that we are going to call, to understand what we will need to be passing in.

We use an example web service:

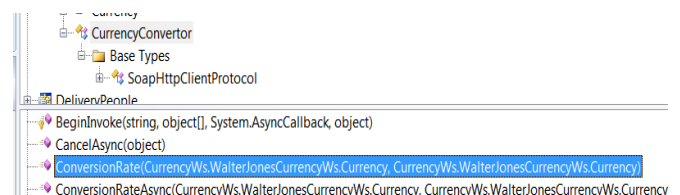
<http://www.webservicex.com/CurrencyConvertor.asmx?WSDL>

This web service provides the exchange rate given two three letter currency codes.

In order to use it, we will need to first add it to the project as a web reference. This can be easily done by right-clicking on the project in Visual Studio, and selecting Add Web Reference.



Once you have the web reference added, you can explore it in the class view, to see that you will need to supply the two three letter currency codes (for example, AUS or USD).



So from this information, we know that we will need to pass it two currency codes, and receive a conversion rate (as a floating point number) in return. There is a slight complication to do with providing a valid 'Currency' to the web service, but essentially we are looking at a method that will take two parameters (the two country currency codes we are trying to calculate exchange rates for), and a return value containing the actual rate.

Defining the interface method

So, the method on the interface can be as simple as:


```

namespace CurrencyWs
{
    public interface ICurrency
    {
        int GetRate(string
srcCurrency, string destCurrency);
    }
}

```

In order to have the method do anything, we need it to make the call to the web service, having passed in those two string parameters.

We implement methods, in a class.

The Class definition

The class is defined as public and implements the interface we defined earlier. Remember our interface contains one method which we have included in the body of the class. We've also added a constructor, as the C# class needs a constructor (although we don't need it to do anything specific).

```

public class Currency : ICurrency
{
    public Currency()
    {
    }
    public int GetRate(string
srcCurrency, string destCurrency)
    {
        // we will implement the
method here
    }
}

```

The method GetRate then implements the code which makes the actual call to our referenced web service. Here's the body of the method GetRate.

```

public int GetRate(string srcCurrency,
string destCurrency)
{
    try
    {
        CurrencyWs.WalterJonesCurrencyWs.Cu
urrency fromSymbol =
GetCurrency(srcCurrency);

        CurrencyWs.WalterJonesCurrencyWs.Cu
urrency toSymbol =
GetCurrency(destCurrency);

        double dRate =
1.0;

```

```

        if (fromSymbol !=
toSymbol)
        {
            CurrencyConvertor myCurrencyWs =
new CurrencyConvertor();

            dRate =
myCurrencyWs.ConversionRate(GetCurrency(s
rcCurrency), GetCurrency(destCurrency));
            if (dRate
== 0.0) dRate = 1.0;
        }
        // Convert rate
to int with 4 decimals
        dRate = dRate *
10000;
        return
Convert.ToInt32(dRate);
        //return 15000;
    }
    ///catch
    (System.Web.Services.Protocols.SoapExcept
ion error)
        catch
    (System.Web.Services.Protocols.SoapExcept
ion)
    {
        // When error,
return rate 1.0000
        return 10000;
    }
}

```

There is another utility method which is not exposed to AB Suite, but is used for getting a 'Currency' enumeration that the WebService call expects. This 'GetCurrency' method is detailed below.

```

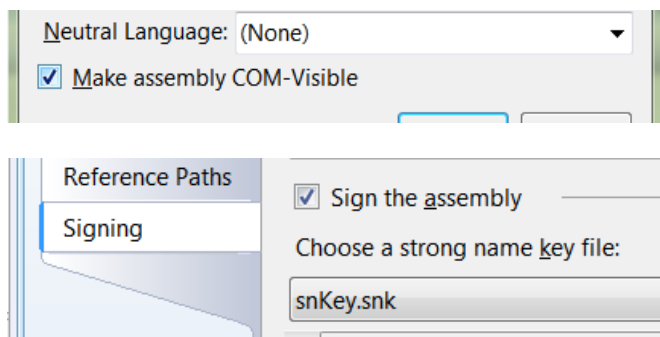
private
CurrencyWs.WalterJonesCurrencyWs.Currency
GetCurrency(string symbol)
{
    if (symbol != "")
    {
        foreach (string s in
Enum.GetNames(typeof(CurrencyWs.WalterJon
esCurrencyWs.Currency)))
        {
            if (s == symbol)
                return
(CurrencyWs.WalterJonesCurrencyWs.Currenc
y)Enum.Parse(typeof(CurrencyWs.WalterJone
sCurrencyWs.Currency), symbol, true);
        }
    }
}

```

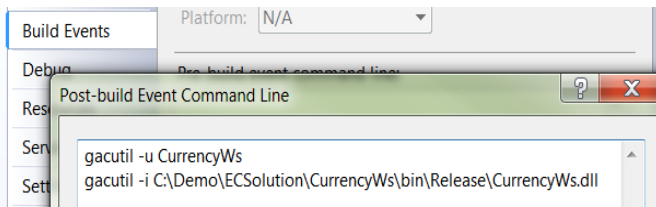
```
// Return default
currency when symbol specified is invalid
return
CurrencyWs.WalterJonesCurrencyWs.Currency
.USD;
```

Notice that the utility method is 'private' and is not defined in the interface. This protects it from being inadvertently called by external users and is an example of properly encapsulating your code.

We also need to make the assembly COM visible and signed for it to be usable by AB Suite.



After the assembly is built, we can then copy it to the runtime host ready to be used with AB Suite. Remember, that Runtime can only find our assembly, if it is either its path is present in the PATH environment variable, or is registered in the GAC. For testing purposes, we take the latter approach, and have added the assembly to the GAC on our machine with a post-build event.



The first line in the post-build event unregisters any pre-existing versions of the assembly. This is useful if you are rebuilding the assembly repeatedly, to ensure you are only testing against the latest version of your assembly.

The second line in the post-build event installs the recently rebuilt version of your assembly into the GAC.

It is very important to realize here, that the assembly will be registered in the GAC *on the same machine that Visual Studio is running on*. This might be fine if we are simply testing the assembly on the same machine before we release it, but we will need to remember to manually register the assembly on our target Runtime host server when we are satisfied with the assembly.

A simple way of deploying the assembly onto your runtime host would be to copy the assembly DLL to your runtime host, and then run those two lines from the post build event on the runtime host. This can be automated with a script or a batch file, if there is a direct access (either through a file share or through FTP) from your development machine to your runtime host.

External Class - Configuring AB Suite to call the C# assembly

AB Suite will call the new assembly, using the corresponding External Class. We start by creating a new class – we call it CurrencyWS to mirror the corresponding assembly.

CurrencyWs Properties	
(Name)	CurrencyWs
Alias	Crrnc
Author	ldatest
Caption	
Created	25/05/2011 2:43 PM
Description	
InterfaceType	None
IsConstant	False
IsExternal	True
Kind	Attribute
Length	0
Modified	28/02/2012 11:00 AM
Multiplicity	1
Owner	Orders
Primitive	Class
ReservedBy	
SynchronizeFile	
VersionFile	
Visibility	Public

When we create the new class, we also set the IsExternal configuration property to True. We've also set the multiplicity to 1 so that we have an instance of the External Class that we can invoke when calling the methods.

Since, the External Library that we are calling is actually a .NET assembly, we now have to configure the External Class with additional configuration properties. Right-clicking on the new External Class, we can set the Component Type to .NET, which enables the rest of the configuration properties.

External	
Component Type	.NET
Identifier	CurrencyWs.Currency
.NET Assembly	:\CurrencyWs\bin\Release\CurrencyWs.dll
Default Interface	ICurrency
Type-Library	

The Identifier is made up of two parts, the Namespace and the Class. In the previous section, we used

'CurrencyWs' as the namespace and 'Currency' as the class. So, in the Identifier property we use 'CurrencyWs.Currency'.

The .NET Assembly field points to the assembly that we built on our development workstation. Simply locate the .dll file using the browse button. Make sure it is the latest version of the assembly (which is typically the version in the output directory of your assembly's project).

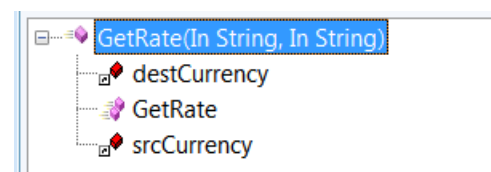
The Default Interface field is the interface that contains the method that we wish to call. In the above example, we defined that interface as 'ICurrency'.

Adding the Method and Parameter to the External Class

Now that we have our External Class defined, we need to define the matching method. Recalling the C# assembly we built previously, we have a single method - GetRate

```
public int GetRate(string srcCurrency,
string destCurrency)
```

So, we will define a method that has two input string parameters and accepts an integer return value. By right-clicking on the External Class, we add a method called GetRate and we add two String parameters (destCurrency and srcCurrency) and set their directions to In.



The first parameter 'srcCurrency' is defined as length 3 because we know that it will contain a 3 letter country code. Note that also has a Sequence of 1 because it will be the first parameter.

(Name)	srcCurrency
Author	ldatest
Caption	
Created	25/05/2011 2:43 PM
Description	
Direction	In
Inherits	
Kind	Parameter
Length	3
Modified	25/05/2011 2:43 PM
Multiplicity	1
Owner	CurrencyWs.GetRate
Primitive	String
ReservedBy	
Sequence	1
Value	
VersionFile	

The second parameter 'destCurrency' is also defined as length 3 because it will also contain a 3 letter country code. 'destCurrency' has a Sequence of 2, because it will be the second parameter to be passed.

destCurrency Properties Parameter Properties	
(Name)	destCurrency
Author	ldatest
Caption	
Created	25/05/2011 2:43 PM
Description	
Direction	In
Inherits	
Kind	Parameter
Length	3
Modified	25/05/2011 2:43 PM
Multiplicity	1
Owner	CurrencyWs.GetRate
Primitive	String
ReservedBy	
Sequence	2
Value	
VersionFile	

The return variable is a Signed Number of length 6, as it returns the rate multiplied by 10000.

GetRate Properties Variable Properties	
DecimalCharacter	.
Decimals	0
DecimalsKeyed	Optional
Description	
Direction	None
Inherits	
IsConstant	False
Kind	Variable
Length	6
Modified	25/05/2011 2:43 PM
Multiplicity	1
Owner	CurrencyWs.GetRate
Primitive	Signed Number
ReservedBy	
SeparatorCharacter	<none>
Stereotype	
Value	
VersionFile	

Finally, while we have specified that the two parameters are Strings, and the return variable is a Signed Number, we still need to further configure the parameters to tell AB Suite exactly what kind of string of number is being expected by the External Library.

Fortunately, for a C# assembly External Library, strings only have one option, System.String. So for both string parameters, they already have a configuration property which sets their Primitive Type to System.String.

External	
Primitive Type	System.String

Right-click on each parameter to view the Primitive Type.

For the Return Variable, we have set the primitive type to 'int' since we know that the corresponding GetRate method on the External Class, returns an int.



Implicit conversions between some of the other available primitive types (long, float, double or decimal) means that the system would also work if you specified some the other types, but it is better to use the exact type that is expected, to avoid any unintended behaviors.

Putting it together – calling our Web Service from AB Suite

So, now we have our External Class that calls a corresponding External Library, which in turn calls a web service.

To call the web service, all we need in AB Suite, is a method call.

```
CurrencyRate :=  
(CurrencyWs.GetRate("USD", "AUD") /  
10000)
```

We have an instance of the CurrencyWs External Class and it has a method called 'GetRate', which takes two string parameters. We pass in two string literals "USD" and "AUD" for the source and destination country codes. Noting that the GetRate method returns the rate as an integer (where the rate was multiplied by 10000 by our External Library), we convert the result back to a multiplied, by dividing the return value of the GetRate call by 10000.

Example 2: Calling AB Suite as an External Library

AB Suite 2.0 Runtime for Windows applications are generated and deployed as .NET assemblies. Therefore, they can themselves be called via an External Class, which provides some interesting collaboration opportunities between multiple AB Suite applications. Note that there are many other ways for getting AB Suite applications to communicate with one another (most notably through the HUB protocol), but this example helps to understand issues that might arise when you are trying to call an External Library that has been written elsewhere.

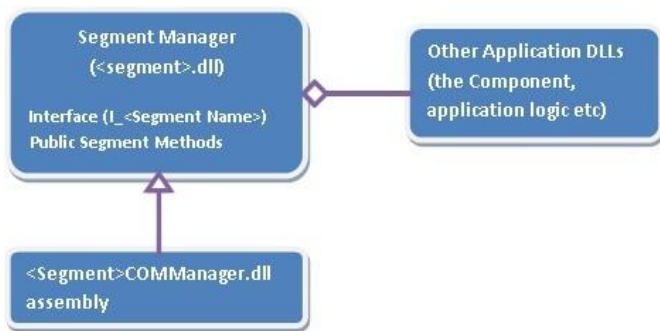
External Class interface

As discussed in previous sections, we need to know the details of the External Library's namespace, interface and method names. This is easy when you are writing your own External Library, but if we are calling AB Suite, then we won't have access to the source code to help us determine the correct configuration settings. Let's start by considering an External Class that will represent an AB Suite application.

The Identifier

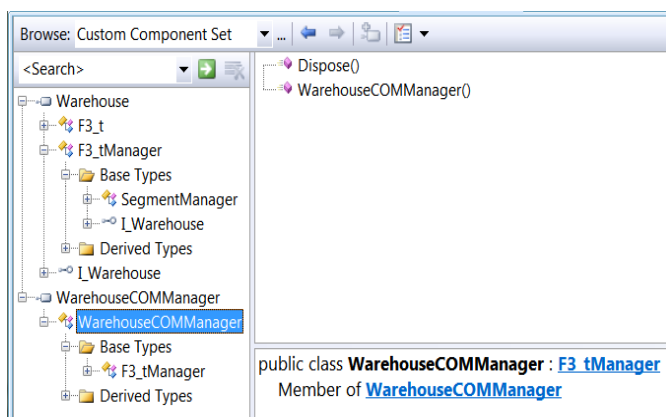
Previously, it was recommended that C# assemblies used by AB Suite have an Identifier that was constructed from a Namespace and a Class name. Unfortunately, AB Suite assemblies do not use namespaces to fully qualify class names. This can cause naming conflicts, which are outside the scope of this document. Furthermore, the way AB Suite applications are constructed are complicated by the need for them to behave as COM+ applications, support pooling and scalability features, support the segment processing cycle as well as providing access to the public segment methods.

The AB Suite application is actually made up of several DLLs, and the <Segment>COMManager.dll is the actual assembly DLL that gets registered with COM+ and the GAC. This DLL in itself does not contain the interface or methods that we wish to call. They are located in another DLL, called the segment manager DLL. The segment manager dll typically has the same name as the segment.



So there's no namespace, and the Interface and Methods are in a separate DLL, and we don't have access to the source code. The best place to start is to have a look at the COMManager.dll, in the Visual Studio object browser. We can actually open the COMManager and the Segment Manager DLLs in the object browser to better understand their relationship. See the example below.

This is the COMManager and Segment Manager DLLs for an AB Suite system called Warehouse. The COMManager assembly DLL is called WarehouseCOMManager, and its Segment Manager is Warehouse.dll. Internally, the segment manager class is actually called 'F3_tManager' for efficiency reasons. Almost all AB Suite systems will have a similar structure in the Object Browser.



The WarehouseCOMManager, is the class that we want to instantiate because it is the .NET assembly that logically contains the whole application. Note that it does

not belong to an explicitly defined namespace, so our Identity need only be 'WarehouseCOMManager'. Fortunately, the name WarehouseCOMManager is unique enough not to cause naming conflicts (unless you have another class with the same name in use on the same environment).

External	
Component Type	.NET
Identifier	WarehouseCOMManager

The WarehouseCOMManager's base class is actually the Segment Manager. You can see that clearly listed under the WarehouseCOMManager in the Object Browser window above. This means that when you instantiate the WarehouseCOMManager, you will also get an instance of the Segment Manager, which in turn contains the interfaces and methods that we want to call.

The Object Browser can be used to deduce the correct class to call for other .NET Assemblies as well. The key here is to determine the class that you would want to instantiate to get the required functionality. Note that the Object Browser can also be used for unmanaged COM components as well.

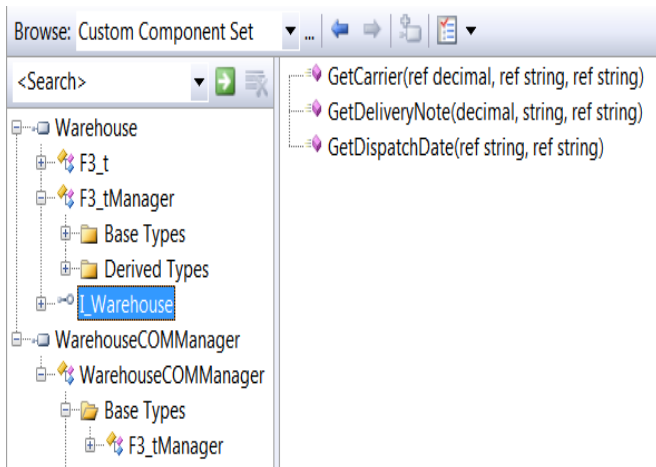
The .NET Assembly

From the description above, we can deduce that the WarehouseCOMManager.DLL is the actual .NET assembly, so we will use that in the '.NET Assembly' configuration property.

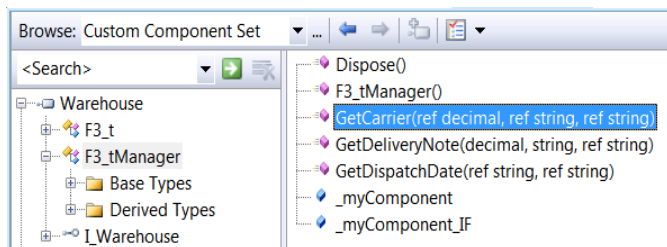
External	
Component Type	.NET
Identifier	WarehouseCOMManager
.NET Assembly	ws\F3\core\bin\WarehouseCOMManager.dll

The Default Interface

The interface for an AB Suite application is I_<Segment name>. So. for the example above, the interface is I_Warehouse. We can also clearly see it and its available methods in the Object Browser view.



Here, we can see that there are 3 methods with a mix of number (Decimal) and string parameter types. These are the public segment methods, which are defined for the Warehouse AB Suite system. When the Segment Manager is instantiated, it includes the implementation of the methods of the interface.



Thus, the Default Interface configuration property should be set as I_<Segment>.

External	
Component Type	.NET
Identifier	WarehouseCOMManager
.NET Assembly	C:\Program Files\Unisys\AB Suite 2.0\BuilderOut
Default Interface	I_Warehouse

Type-Library Configuration Property

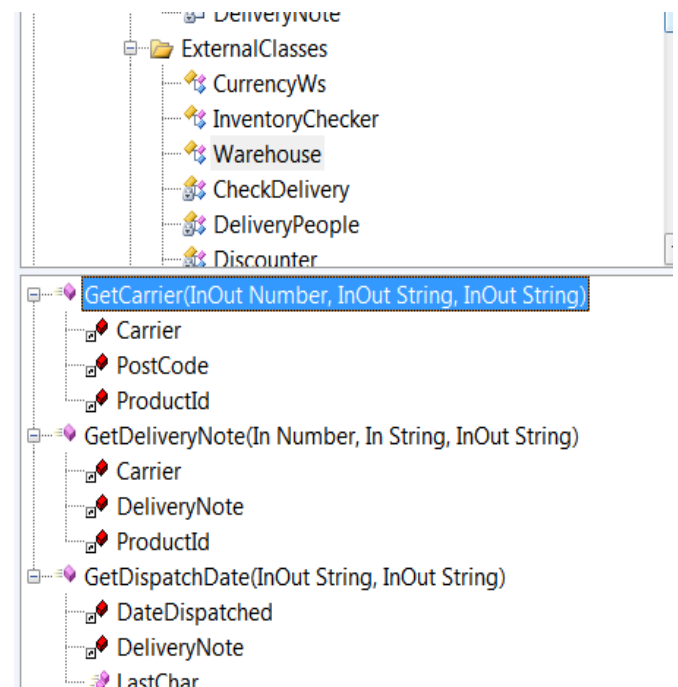
With the Default Interface field set, there is only the remaining, optional configuration property left is the Type-Library Configuration Property. The target AB Suite system's type library can be generated manually by running tlbexp.exe against the

<Segment>COMManager.dll. Otherwise, the property can be left blank and AB Suite will generate the default type library automatically.

External	
Component Type	.NET
Identifier	WarehouseCOMManager
.NET Assembly	C:\Program Files\Unisys\AB Suite 2.0\BuilderOut
Default Interface	I_Warehouse
Type-Library	

Method Definition

To complete the External Class, we add the methods to the class. The procedure for adding methods and their parameters is the same as for other .NET assembly component types. Simply right-click on the external class, add methods and then add matching parameters to the methods.



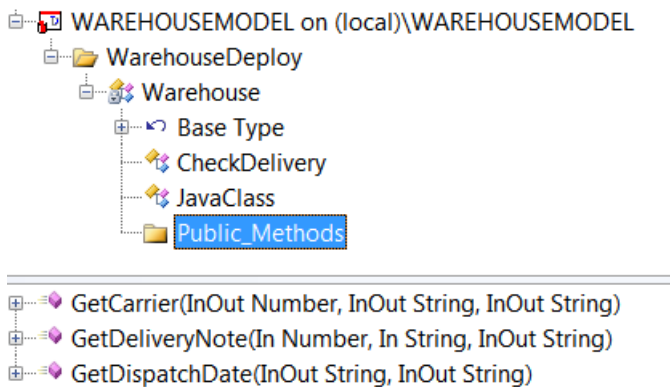
Notice that most of the Number parameters were defined as 'Decimal' primitive types when we looked at the methods inside the Object browser. AB Suite Number parameters in Public Methods have a decimal primitive type. As mentioned in previous sections, it is possible to

use other primitive types, as there are implicit conversions between different number types, but it is recommended to use the exact primitive types to avoid any unintended conversions.

External Library Definition

Now that we have defined the External Class, we can take a brief look at the External Library. In this case, the External Library is actually another AB Suite application (which we are treating as a .NET assembly). An AB Suite system has a segment cycle (which can actually be accessed as an external class, but is not recommended). Access to an AB Suite system as an External Library (or through its COM interface), should be through properly defined Public Segment methods.

Any methods which are defined as public, on the segment, will appear as methods on the default I_<Segment> interface. So in the example above, our Warehouse AB Suite application is expected to have 3 public segment methods. A quick look at the Warehouse AB Suite application in Developer, confirms that this is the case.



The 3 methods have identical method signatures as the External Class' methods. They are also identical to the versions defined in the interface when viewed through the Object Browser. Note that they have been placed in their own 'Public_Methods' folder to allow for easier maintenance of the application.

AB Suite applications do not have to be fully deployed in order for Builder to pick up the assembly, interface and method details. As long as the target AB Suite

application's DLLs are available, AB Suite will be able to use them to build the External Class for the caller AB Suite application. Both caller and target AB Suite applications, however have to be deployed on the runtime host for the call to succeed. The target AB Suite application should be deployed first, as it is a dependency of the caller application.

Debugging Tips

In general, the same steps for setting up an External Class and External Library for use with the AB Suite Runtime for Windows are the same as setting it up for a Debugger environment. The variation relates mainly to path and file location differences.

For Runtime, the External Library is typically available during the build of the AB Suite application on the same build machine, but it then copied over to the Runtime Host for use in Runtime.

For Debugger, the External Library is typically run on the same development workstation, and there is less need to consider copying assemblies to another machine.

If the actual locations for the External Libraries are different (for example between a Build server and the Debugger user's workstation), then there will be configuration changes needed in External Classes that refer to where the External Libraries actually reside.

Debugging the External Libraries

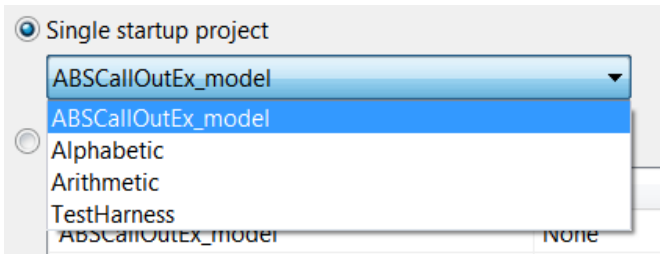
Since AB Suite Debugger runs in a Visual Studio session, it is generally not possible to step into the External Library itself (since Visual Studio is already running a debugger session). It is however, possible to use a different level of Visual Studio to start a debug session within the External Library, and watch the call from both sides of the External Call interface.

Otherwise, Debugger will simply step over the external call (even if you try to step into the external call), and you will be able to observe the parameters as they are passed into the external call, and when the external call completes, and returns back to AB Suite.

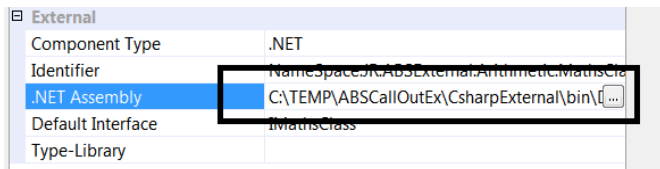
Troubleshooting Tips

Several tips to speed up the setup and configuration of Debugger and Runtime projects that use external class follows:

- Ensure that the Debugger solution is set to debug the AB Suite project:



- Ensure that the external class wrappers are correctly set to point to the locations of the external libraries. This may change if the project was initially developed on another machine.



- Build the external libraries first, to ensure that they are available for the AB Suite build (this applies to both debugger and runtime).
- Interface resolution errors

Information: Debugger has skipped the call to "ExtClassB.Concatenate" as the external binary could not be resolved.

02/02/2012 08:53:28 --

ABSCallOutEx1_model.ABSCallOut.Txn1.Main(Line 23) "

This error can occur if the class in the assembly isn't correctly inheriting from the interface specified in the external class' configuration properties. Check that the class inherits the interface.

- Class resolution errors

Error: Type 'AddCheck.AddressCheck' not defined in 'C:\Program Files\Unisys\AB Suite 2.0\BuilderOutput\MACHINEA\SAMPLEMODEL\Release_Windows\core\AddressCheck_JG3LBB2MUPQHCJELBQQAOTCZG.tlb'

This error can occur if COM visibility is disabled for a particular class. Either set the COM visible attribute to True on the assembly level, or add it as an attribute to the class.

- System.DateTime conversion errors noted in compile log.

This is due to a bug where conversion routines in the data types to handle conversions to and from Date types, are missing.

- Regasm and tlbexp paths missing

The external class builder relies on REGASM.EXE and TLBEXP.EXE paths. If these paths are invalid, or the files are missing, then there will be build errors when trying to generate the external class wrappers. This applies to both Runtime and Debugger.

- Diagnosing Assembly load failures using FUSLOGVW.exe

In some cases, the External Call fails with a file not found exception in the system.log.

=====

Exception:

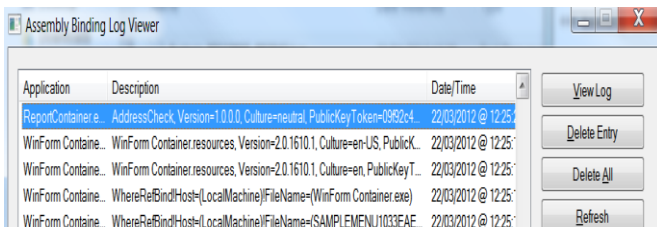
System.IO.FileNotFoundException

Message: Could not load file or assembly 'AddressCheck, Version=1.0.0.0, Culture=neutral, PublicKeyToken=09f92c4b756f83bf' or one of its dependencies. The system cannot find the file specified.

This error indicates there is a problem locating the Assembly on the Runtime Host server. (This could also apply to a Debugger session where you have configured the External Class to look elsewhere for the assembly).

Microsoft provides a useful utility called the Assembly Binding Log Viewer (FUSLOGVW.exe) ([http://msdn.microsoft.com/en-us/library/e74a18c4\(vs.71\).aspx](http://msdn.microsoft.com/en-us/library/e74a18c4(vs.71).aspx)) to help diagnose assembly load issues. This utility is supplied with the .NET Framework, so it is conveniently available on the Runtime Server itself. Set up a Log path as per the Microsoft instructions above, and re-attempt the External Call in runtime.

Refreshing the viewer should indicate a series assembly load attempts. It is likely that there will be an entry for the assembly that you are trying to load, similar to the first entry below.



Opening up these entries will give some clues as to where the .NET Runtime is expecting to find the assembly. The log entry will show you the version numbers that the .NET Runtime is expecting, and also all the paths where the Runtime had tried to look for it. This information is very useful in diagnosing load issues, and will more than likely be enough to figure out where the assembly needs to be placed.

```
LOG: Appbase =
file:///C:/NGENSystems/SAMPLE/Release_Windows/SAMPLE/Core/Bin/
```

```
LOG: Initial PrivatePath = NULL
```

```
LOG: Dynamic Base = NULL
```

```
LOG: Cache Base = NULL
```

```
LOG: AppName = NULL
```

```
Calling assembly : SAMPLE_E3-1,
Version=1.0.1.0, Culture=neutral,
PublicKeyToken=329a8fc2e8e70d4b.
```

```
===
```

```
LOG: This bind starts in default load
context.
```

```
LOG: No application configuration file
found.
```

```
LOG: Using machine configuration file
from
```

```
C:\Windows\Microsoft.NET\Framework\v2.0.5
0727\config\machine.config.
```

```
LOG: Post-policy reference: AddressCheck,
Version=1.0.0.0, Culture=neutral,
PublicKeyToken=09f92c4b756f83bf
```

```
LOG: GAC Lookup was unsuccessful.
```

```
LOG: Attempting download of new URL
file:///C:/NGENSystems/SAMPLE/Release_Windows/SAMPLE/Core/Bin/AddressCheck.DLL.
```

```
LOG: Attempting download of new URL
file:///C:/NGENSystems/SAMPLE/Release_Windows/SAMPLE/Core/Bin/AddressCheck/Address
Check.DLL.
```

```
LOG: Attempting download of new URL
file:///C:/NGENSystems/SAMPLE/Release_Windows/SAMPLE/Core/Bin/AddressCheck.EXE.
```

```
LOG: Attempting download of new URL
file:///C:/NGENSystems/SAMPLE/Release_Windows/SAMPLE/Core/Bin/AddressCheck/Address
Check.EXE.
```

```
LOG: All probing URLs attempted and
failed.
```

This information is also extremely useful should you need to collect information for Support.

For more information visit www.unisys.com

©2012 Unisys Corporation. All rights reserved. Specifications are subject to change without notice.

Unisys and the Unisys logo are registered trademarks of Unisys Corporation. All other brands and products referenced herein are acknowledged to be trademarks or registered trademarks of their respective holders.

Printed in the United States of America

05/12

12-005