IJIS Institute Technical Advisory Committee
Adrian Withy, Tiburon, Inc
Joe Mierwa, PDSG

NIEM INDUSTRY FAQ SERIES

May 2010

NIEM IEPD XML Code Generation in C# with .NET 3.5

Programmatically working with NIEM IEPDs in C# can be challenging due to the overall complexity of working with large XML schema models and the limitations in Microsoft's Xsd.exe code generation tool. This document seeks to provide answers to frequently asked practitioner questions that may come up when working with NIEM IEPDs in .NET and hopefully decrease the effort required to achieve successful NIEM IEPD programming. These questions address using Xsd.exe code generation and rely on XML object serialization. It is worth noting that there are other viable approaches to working with C#.NET and XML that are available such as LINQ to XML.

NIEM Industry FAQ Series

NIEM IEPD XML Code Generation in C# with .NET 3.5

What is Xsd.exe? 2

Why use Xsd.exe instead of Svcutil.exe? 2

How do I run Xsd.exe on a NIEM IEPD? 2

Is there a way of generating a file for each namespace of an IEPD? 5

Why are some classes generated with a numeric postfix (Example: PersonType1, PersonType2, etc.)? 5

What is an XmlChoiceIdentifierAttribute? 5

Known Substitution Group Problems and Workarounds 8

Can generic lists be generated instead of fixed-length arrays? 11

What is Sgen.exe and how do I use it? 11

When I run Sgen.exe I get the error "The top XML element 'X' from namespace 'Y' references…". 12

Can generated code be used for multiple IEPDs? 12

Can I use the generated code in ASMX and WCF web services? 12

Should validation or other business logic be added to the generated classes? 12

What architectural patterns have been found successful in working with generated XML code? 13

NIEM Industry FAQ Series

NIEM IEPD XML Code Generation in C# with .NET 3.5

WHAT IS XSD.EXE?

Xsd.exe (or XML Schema Definition Tool) is a code generation tool that is part of the Windows SDK and is packaged with Visual Studio. It allows various forms of code generation relating to XML and XML schema for both C# and Visual Studio.NET. It can create an XML schema document from XML, sample XML from an XML schema, XML schema from a .NET assembly, or C# and Visual Basic from XML schema. It is this last code generation capability that is valuable to us as we can generate C# and Visual Basic XML serialization code from the NIEM IEPD schemas.

WHY USE XSD.EXE INSTEAD OF SVCUTIL.EXE?

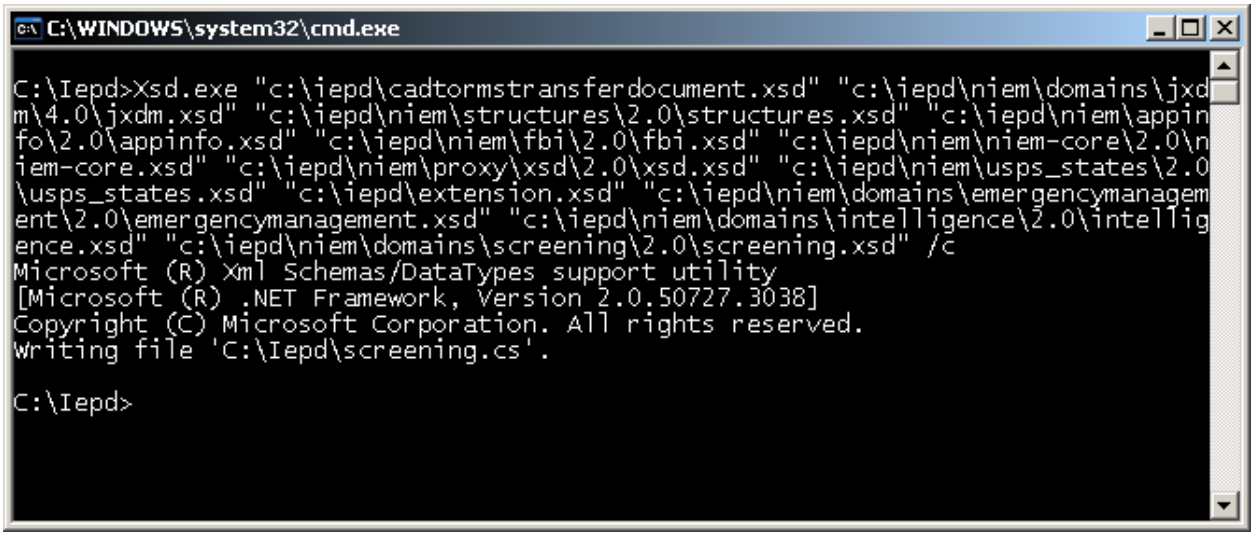
Svcutil.exe is a newer code generation utility that Microsoft has released as part of WCF in .NET 3.0. Unfortunately, SvcUtil.exe does not support the more advanced XML schema features that are required by NIEM such as abstract types and substitution groups.

HOW DO I RUN XSD.EXE ON A NIEM IEPD?

The first step in working with an IEPD is to create a batch file that will execute Xsd.exe. Because NIEM IEPDs are comprised of multiple xsd files and Xsd.exe will not automatically find imported schemas, we will need to include every xsd in the Xsd.exe input. The batch file will take the format:

```
> Xsd.exe "<Path to xsd file 1>" "<Path to xsd file 2>" … "<Path to xsd file n>" /c
```

Example:



The trailing `/c` generates classes instead `/d` to generate DataSets. A best practice has been found to work with classes rather than DataSets due to performance benefits and alignment to the structured non-relational nature of XML.

Once this batch file has been created and ran, the output should be a single large C# file. This step can be automated fairly easily by tooling that recursively searches a directory for xsd files or one that navigates the schema import tree. The following is a simple program that will generate this batch file from the root xsd by recursively navigating through xml schema imports:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.Xml.Schema;
using System.IO;

namespace BuildXsdInput
{
    class Program
    {
        static void Main( string[] args )
        {
            try
            {
                if( args.Length != 1 )
                {
                    Console.WriteLine( "Invalid arguments.  Sample:
                       BuildXsdInput.exe <inputSchema>.xsd" );
                    Environment.Exit( 0 );
                }

                string inputSchema = ( new FileInfo( args[ 0 ] ) ).FullName;
                string outputDirectory = Environment.CurrentDirectory;

                List<string> includeFilenames = new List<string>();
                includeFilenames.Add( inputSchema );

                BuildIncludeList( inputSchema, ref includeFilenames );

                OutputXsd( includeFilenames, outputDirectory );

                Console.WriteLine( "Created XSD input batch file successfully." );
            }
            catch( Exception Ex )
            {
                Console.WriteLine( "ERROR: " + Ex.ToString() );
                Console.ReadKey();
            }
        }

        private static void BuildIncludeList( string xsdFilename, ref List<string>
includeFiles )
        {
            XmlSchema xs = XmlSchema.Read( XmlReader.Create( xsdFilename ), null );

            foreach( XmlSchemaObject schemaObject in xs.Includes )
```

```
            {
                if( schemaObject is XmlSchemaImport )
                {
                    string currentDirectory = Path.GetDirectoryName( xsdFilename );

                    XmlSchemaImport schemaImport = ( XmlSchemaImport )schemaObject;

                    string fileName = Path.Combine( currentDirectory,
schemaImport.SchemaLocation
                                    );
                    FileInfo fi = new FileInfo( fileName );
                    if( !includeFiles.Contains( fi.FullName ) )
                    {
                        includeFiles.Add( fi.FullName );

                        // Recursion
                        BuildIncludeList( fi.FullName, ref includeFiles );
                    }
                }
            }
        }

        private static void OutputXsd( List<string> includeFilenames, string
outputDirectory )
        {
            string xsdExeArgs = "";

            List<string> alreadyAdded = new List<string>();

            foreach( string includeFilename in includeFilenames )
            {
                string filenameToInclude = includeFilename.ToLower().Trim();
                if( !alreadyAdded.Contains( filenameToInclude ) )
                {
                    xsdExeArgs += "\"" + filenameToInclude + "\" ";
                    alreadyAdded.Add( filenameToInclude );
                }
            }

            xsdExeArgs += "/c";

            TextWriter writer =
                        new StreamWriter( Path.Combine( outputDirectory,
"ExecXsd.bat" ), false );
            using( writer as IDisposable )
            {
                writer.Write( "Xsd.exe " + xsdExeArgs );
            }
        }
    }
}
```

More information about Xsd.exe can be found at the following link: http://msdn.microsoft.com/en-us/library/x6c1kb0s.aspx.

IS THERE A WAY OF GENERATING A FILE FOR EACH NAMESPACE OF AN IEPD?

No. This is unfortunately a painful process and that is probably not worth the effort unless you create additional tooling for code parsing of the Xsd.exe output that can automatically split it into multiple files.

WHY ARE SOME CLASSES GENERATED WITH A NUMERIC POSTFIX (EXAMPLE: PERSONTYPE1, PERSONTYPE2, ETC.)?

Because Xsd.exe does not create namespaces for different XML Schema namespaces, when an extension type is encountered, it will append a numeral to the name of the generated class. A common approach is to either separate the generated code into multiple namespaces and remove the numeral, or renaming the extension type with a prefix that indicates which namespace it is from (example: LeitscPersonType). The Visual Studio refactoring tools can be very helpful in this type of batch renaming.

WHAT IS AN XMLCHOICEIDENTIFIERATTRIBUTE?

An XmlChoiceIdentifierAttribute is an attribute used in .NET XML serialization of XML substitution groups. It is used to tag the root substitution group property when there are multiple substitutions of the same type. It will be accompanied by an enumeration of possible types in a secondary collection.

As an example we will show how Xsd.exe generates code for the StructuredAddressType in the NIEM Core. The StructuredAddressType is defined as:

```
<xsd:complexType name="StructuredAddressType">
    <xsd:annotation>
      <xsd:documentation>A data type for an address.</xsd:documentation>
      <xsd:appinfo>
        <i:Base i:namespace="http://niem.gov/niem/structures/2.0" i:name="Object"/>
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="s:ComplexObjectType">
        <xsd:sequence>
          <xsd:element ref="nc:AddressRecipientName" minOccurs="0"
maxOccurs="unbounded"/>
          <xsd:element ref="nc:AddressDeliveryPoint" minOccurs="0"
maxOccurs="unbounded"/>
          <xsd:element ref="nc:LocationCityName" minOccurs="0"
maxOccurs="unbounded"/>
          <xsd:element ref="nc:LocationCounty" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="nc:LocationState" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="nc:LocationCountry" minOccurs="0"
maxOccurs="unbounded"/>
          <xsd:element ref="nc:LocationPostalCode" minOccurs="0"
maxOccurs="unbounded"/>
          <xsd:element ref="nc:LocationPostalExtensionCode" minOccurs="0"
maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
```

```
    </xsd:complexContent>
  </xsd:complexType>
```

Within the StructuredAddressType, the AddressDeliveryPoint element is an abstract type, as shown here:

```
<xsd:element name="AddressDeliveryPoint" abstract="true"/>
```

There are then a number of substitutions for this abstract type:

```
<xsd:element substitutionGroup="nc:AddressDeliveryPoint"
name="AddressDeliveryPointID" type="niem-xsd:string" nillable="true">
    <xsd:annotation>
      <xsd:documentation>An identifier of a single place or unit at which mail is
delivered.</xsd:documentation>
      <xsd:appinfo>
        <i:Base i:name="AddressDeliveryPoint"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:element>
  <xsd:element substitutionGroup="nc:AddressDeliveryPoint"
name="AddressDeliveryPointText" type="nc:TextType" nillable="true">
    <xsd:annotation>
      <xsd:documentation>A single place or unit at which mail is
delivered.</xsd:documentation>
      <xsd:appinfo>
        <i:Base i:name="AddressDeliveryPoint"/>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:element>
  <xsd:element substitutionGroup="nc:AddressRepresentation"
name="AddressFullText" type="nc:TextType" nillable="true">
    <xsd:annotation>
      <xsd:documentation>A complete address.</xsd:documentation>
      <xsd:appinfo>
        <i:Base i:name="AddressRepresentation"/>
      </xsd:appinfo>
    </xsd:annotation>
</xsd:element>
```

When Xsd.exe generates code for the StructuredAddressType, it will use two collections: one to hold the concrete substitution, and another to specify the type of the substitution. This is necessary because the .NET XmlSerializer is not able to differentiate between concrete substitutions of the same type. In this example, both: AddressDeliveryPointText and AddressFullText are both of type TextType. When in a collection, XmlSerializer will use the enumeration collection (in this example ItemsChoiceType) as an identifier.

Example:

```
public partial class StructuredAddressType : ComplexObjectType
{

…

[XmlElementAttribute("AddressBuildingText",typeof( TextType ),IsNullable=true )]
[XmlElementAttribute("AddressDeliveryPointID", typeof( @string ), IsNullable=true )]
[XmlElementAttribute("AddressDeliveryPointText", typeof( TextType ), IsNullable=true
)]
[XmlElementAttribute("AddressFullText",typeof( TextType ),IsNullable=true )]
[XmlElementAttribute("AddressPrivateMailboxText", typeof( TextType ),IsNullable=true
)]
[XmlElementAttribute("AddressSecondaryUnitText", typeof( TextType ), IsNullable=true
)]
[XmlElementAttribute("LocationStreet", typeof( StreetType ), IsNullable=true )]
[XmlChoiceIdentifierAttribute( "ItemsElementName" )]
public object[] Items
{
    get
    {
        return this.itemsField;
    }
    set
    {
        this.itemsField = value;
    }
}

[XmlElementAttribute( "ItemsElementName" )]
[XmlIgnoreAttribute()]
public ItemsChoiceType[] ItemsElementName
{
    get
    {
        return this.itemsElementNameField;
    }
    set
    {
        this.itemsElementNameField = value;
    }
}
…

}

public enum ItemsChoiceType
{

/// <remarks/>
AddressBuildingText,

/// <remarks/>
AddressDeliveryPointID,

/// <remarks/>
AddressDeliveryPointText,

/// <remarks/>
AddressPrivateMailboxText,
```

```
/// <remarks/>
AddressSecondaryUnitText,

/// <remarks/>
LocationStreet,

}
```

In order to use, you must keep the two collections in sync such that each object in the Items collection has a corresponding enum in the ItemsChoiceType collection.

As you can see in the example above, Xsd.exe does a poor job naming these substitution group root elements. Here the AddressDeliveryPoint is simply called Items. Visual Studio refactoring tools can be used to rename such properties.

When a substitution group does not have concrete substitutions of the same type, this is not necessary and only one collection is used.

KNOWN SUBSTITUTION GROUP PROBLEMS AND WORKAROUNDS

When using XSD.exe there is an issue where substitution groups having members in a namespace different from the one the substitution group head is declared will not always serialize. This issue will manifest itself during testing with unexpected exceptions occurring because of null values in the rendered classes.

Note that the circumstances where a substitution group member will or will not serialize is somewhat dependent on the overall complexity of the IEPD, however, they can be broken down into the following scenarios:

1. A substitution group with a head element and members in different namespaces; and
2. A substitution group without head and extension types in different namespace.

Also note that it does not matter which namespaces are used. All NIEM examples are shown here but the same holds true for extensions. For this discussion, an IEPD using only NIEM components was fabricated that meets the cases as follows:

```
<xsd:complexType name="ExchangePackageType">
            <xsd:sequence>
                  <xsd:element ext="PackageName"/>
                  <xsd:element ref="nc:Activity" minOccurs="0"/>
                  <xsd:element ref="nc:TangibleItem"/>
                  <xsd:element ref="nc:Conveyance"/>
            </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="ExchangePackage" type="sub:ExchangePackageType"
nillable="true"/>
```

Case 1: Substitution Group with a Head Element and Members in Different Namespaces

In this case, the element `nc:Activity/nc:ActivityDate/nc:DateRepresentation` has the substitution members `nc:Date, nc:DateTime, nc:Year, it:Month & it:DayOfWeek` specified in the NIEM subset schema. In this example, nc:DateRepresentation has a cardinality of 0 to unbounded.

Looking at the following snippet of the rendered class DateType:

```
    [System.Xml.Serialization.XmlRootAttribute("ActivityDate",
       Namespace="http://niem.gov/niem/niem-core/2.0", IsNullable=true)]
    public partial class DateType : ComplexObjectType {

        private object[] itemsField;

        /// <remarks/>
        [System.Xml.Serialization.XmlElementAttribute("Date", typeof(date), IsNullable=true)]
        [System.Xml.Serialization.XmlElementAttribute("DateTime", typeof(dateTime),
                IsNullable=true)]
        [System.Xml.Serialization.XmlElementAttribute("Year", typeof(gYear), IsNullable=true)]
        public object[] Items {…
```

You can see that the XmlElementAttribute attributes above the Items array only contains the substitution group members that were contained in the niem-core namespace. The two members from the intelligence namespace are not shown, which means that deserializing the following XML instance fragment would not occur on the intelligence time values.

To remedy this issue, you would need to go into the rendered class file and add an XmlElementAttribute attribute for each element name from another namespace, specifying the type and the namespace as shown below.

```
    [System.Xml.Serialization.XmlRootAttribute("ActivityDate",
       Namespace="http://niem.gov/niem/niem-core/2.0", IsNullable=true)]
    public partial class DateType : ComplexObjectType {

        private object[] itemsField;

        /// <remarks/>
        [System.Xml.Serialization.XmlElementAttribute("DayOfMonth", typeof(DayType),
                Namespace = "http://niem.gov/niem/domains/intelligence/2.1", IsNullable = true)]
        [System.Xml.Serialization.XmlElementAttribute("Month", typeof(gMonth),
                Namespace = "http://niem.gov/niem/domains/intelligence/2.1", IsNullable = true)]
        [System.Xml.Serialization.XmlElementAttribute("Date", typeof(date), IsNullable=true)]
        [System.Xml.Serialization.XmlElementAttribute("DateTime", typeof(dateTime),
                IsNullable=true)]
        [System.Xml.Serialization.XmlElementAttribute("Year", typeof(gYear), IsNullable=true)]
        public object[] Items {…
```

Case 2: Substitution Group without Head and Extension Types in Different Namespace

In this case, the element `nc:Conveyance` is being implicitly substituted with `cbrn:Conveyance` which is an extension adding an augmentation to `nc:Conveyance`. Note that in this case the cardinality of `nc:Conveyance` within ExchangePackageType is a minimum and maximum of one. Looking at the rendered class for the "TangibleItem" element, you will see that there is no element name associated with the Conveyance XmlElementAttribute attribute.

```
        [System.Xml.Serialization.XmlElementAttribute(
                        Namespace="http://niem.gov/niem/niem-core/2.0", IsNullable=true)]
        public TangibleItemType TangibleItem {
            get {… [deletia]
        }

        /// <remarks/>
        [System.Xml.Serialization.XmlElementAttribute(
                        Namespace = "http://niem.gov/niem/niem-core/2.0", IsNullable = true)]
        public ConveyanceType Conveyance {
            get {…[deletia]
        }
```

Consequently, if an XML instance contains a cbrn:conveyance element is substituted in place of nc:conveyance, when deserialized, Conveyance in class Tangible item will be a null value. To remedy this issue, the highlighted XmlElementAttribute attribute above, must be replaced with the ones below, where both must explicitly state the element name and the associated namespace.

```
        [System.Xml.Serialization.XmlElementAttribute(
                            Namespace="http://niem.gov/niem/niem-core/2.0", IsNullable=true)]
        public TangibleItemType TangibleItem {
            get {… [deletia]
        }

        /// <remarks/>
        [System.Xml.Serialization.XmlElementAttribute("Conveyance", typeof(ConveyanceType),
                        Namespace = "http://niem.gov/niem/niem-core/2.0", IsNullable = true)]
        [System.Xml.Serialization.XmlElementAttribute("Conveyance", typeof(ConveyanceType1),
                        Namespace = "http://niem.gov/niem/domains/cbrn/2.1", IsNullable = true)]
        public ConveyanceType Conveyance {
            get {…[deletia]
        }
```

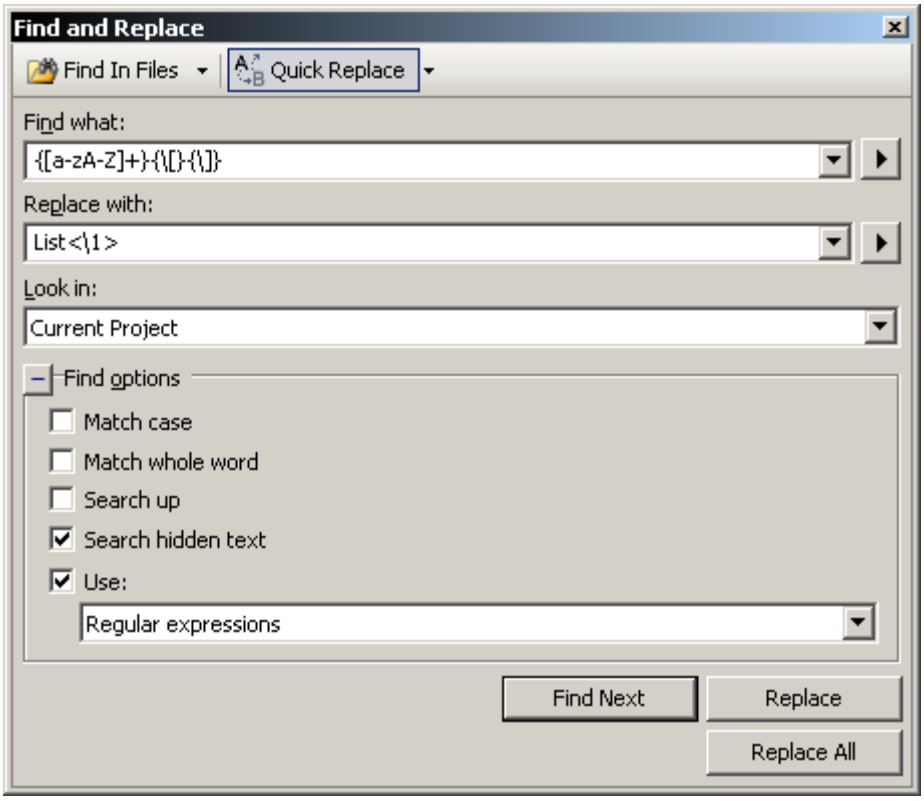
Finally, note that the key difference between these two different cases really comes down to cardinality specified for the element. Both cases do not pick up the substitution group members in the other namespaces. The only difference is that case 1, having 0 to unbounded cardinality, illustrates how the problem look with the array and the other illustrates the case of single type.

CAN GENERIC LISTS BE GENERATED INSTEAD OF FIXED-LENGTH ARRAYS?

No. However from within Visual Studio you can do a RegEx find-replace to change all arrays into generic lists.

Find: {[a-zA-Z]+}{\[]{\]}

Replace: List<\1>



Note: XML serialization may fail after performing this step. .NETs serialization of substitution groups will require fixed-length arrays when a substitution group has the attribute XmlChoiceIdentifier. You can switch these back to fixed-length arrays after the find-replace fairly easily.

WHAT IS SGEN.EXE AND HOW DO I USE IT?

Sgen.exe is a tool that is provided with Visual Studio and can be used to optimize XML serialization. When used directly, XmlSerializer will create a serialization assembly at run-time for serializing and deserializing the generated code. It relies heavily on reflection and has the potential to perform very poorly given the size of some IEPDs. Sgen.exe can be used to create a serialization DLL from an existing DLL that contains the generated classes so that they are not created at runtime. To use Sgen.exe, call it from the command line with the name of the DLL that contains the generated XML serialization classes as an input argument.

Example: > sgen.exe MyNiemIepdXmlPersistence.dll

This will generate a XmlSerializers DLL called MyNiemIepdXmlPersistence.XmlSerializers.dll with namespace Microsoft.Xml.Serialization.GeneratedAssembly with a number of type serializers that should be used instead of the default System.Xml.Serialization.XmlSerializer class.

WHEN I RUN SGEN.EXE I GET THE ERROR "THE TOP XML ELEMENT 'X' FROM NAMESPACE 'Y' REFERENCES…".

The committee has found this error to be caused by a missing TypeName from the XmlType attribute. When a class has the same name as the XML type Xsd.exe may leave off the TypeName from the XmlType attribute. Simply find the suspect class (in this case class X) and add the TypeName parameter to the XmlType attribute definition.

CAN GENERATED CODE BE USED FOR MULTIPLE IEPDS?

No. Because each IEPD has a different subset of the NIEM model, and because XML serialization relies upon XmlIncludeAttributes for implementing XML schema extensions, generated code for one IEPD cannot be re-used across IEPDs.

CAN I USE THE GENERATED CODE IN ASMX AND WCF WEB SERVICES?

Yes. These can be used directly as ASMX web method parameters. To use in WCF one must decorate the web service contract with the XmlSerializerFormatAttribute. Also one should consider wrapping the generated code into a WCF MessageContract.

SHOULD VALIDATION OR OTHER BUSINESS LOGIC BE ADDED TO THE GENERATED CLASSES?

No. See the question: "Can generated code be used for multiple IEPDs?" Because generated code cannot easily be shared between IEPDs, it is generally not advisable to add custom code to the generated classes unless this can be supported by additional code generation tooling. As well, since new versions of IEPDs may be developed, being able to quickly regenerate the XML serialization code may be necessary. We recommend using the generated XML serialization code as a low-layer persistence mechanism rather than a middle-layer domain model or active record.

WHAT ARCHITECTURAL PATTERNS HAVE BEEN FOUND SUCCESSFUL IN WORKING WITH GENERATED XML CODE?

We have found the Data Mapper (Patterns of Enterprise Application Architecture, Fowler, 2002) especially valuable when working with generated XML serialization code. Each time one works with a new IEPD, it can be developed into a persistence library with a data mapping layer between it and a business layer. This decouples each IEPD XML serialization library from one another and from the rest of the architecture.

Links:

http://www.martinfowler.com/eaaCatalog/dataMapper.html

http://msdn.microsoft.com/en-us/magazine/dd569757.aspx