****

The Microsoft Insurance Value Chain ACORD Service Factory

Implementing ACORD XML

In Microsoft WCF

A White Paper by David S. Platt

1. Abstract

Implementing ACORD XML applications, particularly web services, has always been difficult. This paper discusses the ways in which Microsoft guidance can make this task easier using Windows Communication Foundation and Visual Studio 2008. I introduce the beginnings of Microsoft’s Insurance Value Chain ACORD Software Factory (IVC ASF), a set of classes, tools, documentation and sample applications that helps implementers overcome these problems. Finally, I discuss future developments in the ASF, and ask for users’ input in determining its direction.

2. Problem Background

Implementers of ACORD XML web services and other programs have great difficulty in knowing where to start (and continue, and stop). ACORD XML has a relatively small user base compared to some of the larger standards such as SOAP (used in all web services, of which there are many) and HL7 (used in the healthcare industry, required by the US government for electronic submission of Medicare bills, so it also has a very large application base). ACORD’s user base is much smaller, which means that the market for help in implementing it is much smaller as well. Hence there is a lack of detailed knowledge in this industry, few magazine articles, not many consulting services or tools, not much education, and I don’t think I’ve ever seen a book on it.

As a standards agency, ACORD has not deemed its mission to include providing guidance on implementations, especially because any such efforts would necessarily involve the choice of one platform over others, which ACORD does not like to do. ACORD will examine your program’s XML to see if it conforms to their standard, and certify it if it does or throw it back at you if it doesn’t. But ACORD hasn’t, doesn’t, and won’t say much about how you went about producing or consuming that certified XML.

Developers wanting to implement ACORD XML applications all had to solve similar design problems. What should the SOAP look like? Where should the ACORD elements go? How should the namespaces map? And so on. They spent a lot of time scratching their heads, saying, “How about this?” Because they were all seeing these problems for the first time, they came up with many different answers, most of them requiring custom coding to talk to each other, the very antithesis of what a standard is for. Each spent more time than he wished writing infrastructural code, with a relatively low volume to spread those development costs over. The small unit volume also meant that a third party tools market, which could have eased some of the task and simplified and unified some of the techniques, never really developed. There are lots of SOAP tools today, and a fair number of HL7 tools, but darn few ACORD XML tools, so every application is an expensive custom task.

The problems of developing ACORD applications are very similar to the problems of developing composite applications, by which I mean applications stitched together from components rather than built in a monolithic block – application programmers don’t know where to start, everyone has common infrastructural problems, the unit volume doesn’t allow for much infrastructure development.

Microsoft has helped developers of composite applications solve their problems by providing the Composite UI Application Block and the Smart Client Software Factory. These provided patterns, classes, tools, documentation, and samples, for solving these common infrastructural problems. Figure 1 shows this combination graphically.

3. Solution Architecture

In the same way, Microsoft has started developing the ACORD Software Factory (ASF). This document and its accompanying samples and classes represent the first prototype of it. It contains documentation, some recommended patterns, a quick start application, and some utility classes. Future releases will build on this foundation. Let’s go have a look at it.

4. Sample Application

As part of the ASF, I wrote a sample program to make sure that we could get the thing to really work from end to end, and to demonstrate the techniques that we used to do so. It uses the TxLife 228 transaction, subtransaction 10 – Can Agent Sell? Regulations are complex, agents come and go, and their licenses and carrier appointments are in constant flux. At least one company (GlobeNet Technologies,www.globenettech.com) makes a living by providing a web service that carriers or consolidators can use to keep track of whether an agent proposing to issue a policy actually does have the correct authority to do so.

The client program is shown in Figure 2. The user makes selections from the various drop down lists and clicks “Can Agent Sell?” This calls the web service, described in the next section, passing it a TxLifeRequest object that asks that question in the approved ACORD format. The BackOfficeProcessing module on the server side simulates the behavior. If the agent’s full name doesn’t begin with the letter ‘A’, and if the line of business is Life, and the Jurisdiction is Alabama, then the answer is yes, I can successfully answer your question and the agent can sell. Other situations produce different results for the purpose of the demonstration – yes, I can answer that question but no, the agent can’t sell for this reason or that one, or no, sorry, I can’t successfully answer the question at all.

5. WCF Implementation

The ASF contains two sample projects, one for life and one for P&C. In the interests of succinctness, this paper will cite only the former except where differences are significant. They are currently demonstrations from which a developer could cut and paste code (the old “editor inheritance”), but someday soon we hope to have a guidance package that would generate them, with options, on demand.

WCF is based on the notion of contracts, and the first instance we encounter is the notion of a Service Contract. This is an interface that is exposed as a unit, containing a set of logically related methods, each of which performs a particular task. The ASF has broken down the ACORD standards into several interfaces based on the transactions that they implement. For example, the life standard contains such service interfaces as IProducerLicensingAndAppointments (Figure 3), which implements methods pertaining to the transactions that deal with such things. The P&C version provides such interfaces as IHomeOwnersSubmission and IHomeOwnersNotification.

The ACORD standard is silent on the desirability or undesirability of spreading the different transactions in this manner. The standards are both designed with all-encompassing envelopes, <TxLife> and <ACORD> respectively, and you don’t know what type of transaction is in either one until after you have received it and opened it and looked at its contents. The design question for your choice of web methods is whether to be a splitter or a clumper. The sample application demonstrates the former approach, multiple interfaces containing multiple methods. In the latter approach, would you have a single method that accepted all incoming ACORD transactions, looked at them, then routed them to the proper location based on their type. The latter is like a single mailroom at a large company that accepts all incoming mail at a single point, then distributes it internally. The former is like a situation in which individual messages have to bear the address of the specific room in which they are processed. The ACORD standard was originally designed with the latter in mind, but is certainly usable in the former with no significant modifications. One of the first questions that the developers of the ASF wants answered by the developer community is which of these approaches better suits your development plan: splitter, clumper, both, neither, or I don’t care just pick either damn one and hurry up?

On the service interface definition, you see a base namespace, a problem domain vocabulary identifier. [Colin, the change to the ACORD namespace was necessary to have the root data element TxLifeRequest or TxLifeResponse be placed into that namespace. This is necessary for ACORD certification. The MessageParameter attribute controlled the element name, but not its namespace]. Each interface method contains a number of attributes that affect its behavior, for example, the names of the request and response elements. The OperationContract attribute can specify such properties as IsOneWay, to be used in a fire-and-forget message (not currently much used in ACORD, but at least under consideration for the prototype standard AWSP in certain circumstances).

Each method name contains a fault contract definition which is used in the case of error handling. Today each ACORD standard contains its own internal mechanisms for reporting business errors, such as sending a transaction to the wrong address, or failing to include the elements needed to process a transaction. If a fault makes it through the WCF infrastructure to reach the business message processing, compliance with the ACORD standard will generally require the use of the ACORD method for signaling faults. Accordingly, this document will not deal further the fault contracts, unless the user community so demands. They will be coming up in AWSP, though, so we won’t completely ignore them, just defer them until version 2 of the ASF.

Having defined the service interface, WCF now connects a class to it. The service behavior class for the sample program is shown in Figure 4. It contains code that implements that particular method on the interface, some of which can be seen in the figure. The separation of external interface from internal implementation is considered to be one of the main architectural improvements of WCF. (Most application programmers I know look at this and say, “Now I have to write every message signature twice, once for the interface and once for the implementation, where I only used to have to do it once. Explain to me how that makes my life easier.” But we digress.) The methods of this behavior class are where the business logic goes. In this case, you see the check for the ACORD transaction type and subtype. If we stick with the separate interfaces, then every method will need to check for its own types and throw back an error if found. In this case, perhaps we would build this capability into a base class, perhaps signaled by a programming attribute. And if we switch to the single method, then checking the incoming types and routing the messages based on it will have to be done in every app, in which case that functionality is also a prime candidate for a base class. (Note: the ACORD AWSP standard seems to imply that ACORD is looking to move in the direction of finer granularity, which means more interfaces and methods, not the single mailroom approach. But the spec also says that they’re far from it today, and not quite sure how they’ll get there, except that the PCS 2.0 standard is indeed moving in the direction of greater granularity. So I think if we provide a catch-all clumper method, it should be for backwards compatibility.)

On the service behavior class we see the name of the service behavior attribute. This points us at a section of the web.config file (shown in Figure 5) in which the web service behaviors of this particular implementation are specified. For example, this configuration specifies the binding that this implementation uses (basic HTTP, HTTP with WS-I enhancements, etc.) These behaviors are not specified by the ACORD standard, at least not yet. These are to be specified between trading partners. Having gotten you to the point where you can have this discussion with your trading partners, this introductory document will not deal with them further.

6. Wrapper Classes

ACORD XML packets are represented in the Microsoft Common Language Runtime as .NET classes. When a packet comes in from a client, it gets converted from XML into such a class (“deserialized”). And when we return a .NET object as the result of a web service method, it gets converted into XML (“serialized”) before being sent out on the wire. All of this is basic web service operation. The trick is where to get this .NET class from.

Historically we’ve just fed the ACORD schema to the xsd.exe tool, which generated wrapper classes. In .asmx web services, that was all we needed to do. We’d take this wrapper class, and put it in a web method. We’d apply the SoapDocumentAttribute as shown in Figure 6, which would take the method name out of it and give us XML as shown in Figure 7. The ACORD element is the only content in the body. This gets a little bit tricky in WCF, but let me finish my discussion of wrapper class internals before dealing with this particular serialization problem.

The wrapper class serializes well, (or can be made to, of which more anon), but it’s a little bit sparse and hard to use. We can make it easier by following certain design patterns. The first one is to have constructors for easily creating the object in a particular state. For example, when a client presents a TxLifeRequest to a service, the specs call for echoing a fair amount of data back to the client, such as the transaction type and subtype and the TransRefGuid. Rather than write code to copy all of these in our business logic, I provided a constructor that performed these operations behind the scenes, as shown in Figure 8. For another example, a TransResult element always contains a ResultCode element, and often also contains a ResultInfo object inside it (Figure 9). I wrote a constructor for this class that provides the entire structure if you pass the needed parameters in, as shown in Figure 10. I added a file ExtendedTxLife to contain these, and the partial class mechanism makes it easy to add things here.

Another problem that we run into is the problem of arrays. Repetitive elements happen frequently in the ACORD standard. For example, the top level of any OLifE element can contain any number of Party, Holding, etc. objects, which are represented in the wrapper class by an array named Items. It’s annoying and tedious to have to figure out in advance how many parties or whatever we are going to contain, create a hard array of exactly that number, and then use it. It would be much handier to have an expandable array on which we could just call Add, or perhaps AddNew.

I tried one on the TransResult class to see how I liked it. I replaced the array of ResultInfo objects in the wrapper class with a generic List, as shown previously in Figure 10. Adding became a matter of simply putting one in, as shown in Figure 11. In general, this is a useful thing, and one I think we want to work on in the future. There are a number of ways in which this can be accomplished, which primarily depend on the modification of wrapper class generating tools. For example, the XsdObjectGen by Colin Cole and Dan Rogers (see link at <http://www.microsoft.com/downloads/details.aspx?FamilyID=89E6B1E5-F66C-4A4D-933B-46222BB01EB0&displaylang=en>) uses lazy instantiation and carefully wrapped, non-generic ArrayLists to accomplish this goal. However, as I will discuss in the next section, matching the suggested SOAP output to WCF’s serialization mechanisms wrapper class will require significant work in the generation of wrapper classes, and I’d suggest deferring discussion of arrays until we have the discussion of attributes and tools. For example, to support addition of external constructors, I would suggest that the classes generated by this tool be marked as partial.

7. Problems with WCF Serialization

I encountered one problem with XML serialization in WCF. Since it affects everything that we do with the ASF, we need to figure out what to do with it sooner rather than later, and we need the involvement of the user community in choosing a solution.

The SOAP shown previously in Figure 7 shows an ACORD element as the only content of a Soap body. There are no other wrapper elements around it. The Soap provided by WCF, on the other hand, is shown in Figure 12. The method name appears in the Soap packet as a wrapper. ACORD standards currently allow this, inelegant as it may seem, but the future AWSP seems not to allow it, seems to require the ACORD message to be the root and only content of the soap message, as in Figure 7. This was easy to do in .asmx, see the SoapDocumentAttribute that I discussed earlier. How can we do this in WCF?

It is in one sense quite difficult, in another sense quite easy. Here’s what you have to do. WCF allows an application to take complete control over the format of its messages. You do this by means of a message contract, specifying attributes in the wrapper class file. You have to mark the root of the element with a MessageContract attribute, specifying name and namespaces. This puts it into the output. You then have to specify each member of the message on an opt-in basis, instead of the default XML serializer behavior in which public members were automatically sent by default and non-public were not. You do this by marking each attribute that you need to send with the MessageBodyMember attribute. Having done this, the message contract mechanism places the output into alphabetical order, not the order in which the schema listed them. To conform to the ACORD schema, we would have to write our wrapper class to look like Figure 13. It’s possible that there may be other modifications, as we didn’t have time or budget for a full exploration before press time.

How can this be both difficult and easy at the same time? The same way as my business paperwork as a one-man corporation is difficult and easy at the same time. “What the hell?” you ask. Ah, listen and learn, tadpoles.

When I incorporated my business, I found that I had to file many more forms with the government that I did as a sole proprietor. That was the bad news. The good news is that they were so difficult that I couldn’t touch them, so I hired an accountant. I actually got more free time than as a sole proprietor, the only drawback was that I had to pay my accountant a couple of grand per year to make it go away. It was a good tradeoff in the end.

Making this new wrapper class by hand would be a colossal pain in the butt for any particular implementer, and you wouldn’t have the volume to amortize the cost over. But if a) Microsoft were to provide it as part of the ASF, and b) it was in the form of a generator tool that you could just run, same as you run xsd or xsdobjectgen today, that would actually be pretty easy. My first quick look at the problem seems to indicate that this would be a solvable problem. If we had the source code to xsdobjectgen, we could probably knock it out quickly, as it seems to require just these two attributes (albeit many, many times). Further investigation is needed. Producing the class is so difficult that we really have no choice other than to produce a tool that would make it easy – just like my business paperwork, no?

8. Other Auxiliary Classes: The Type Code Combo Box

There are, as I have said, many, many operations that need to be performed routinely again and again. It only makes economic sense to provide prefabricated solutions to some of them. For example, an ACORD service base class providing common functionality would be nice, but we’re just starting out now.

The one I chose to attack for this demo is the problem of selecting from code lists. It’s a very common thing in ACORD, there are, for example 444 separate code lists in the 2.13 life schema. Items such as a party’s gender, which can be male and female (and unisex, used by some gender-neutral life carriers such as SBLI, and combined, used in policy types such as husband-and-wife-second-to-die. The state of Massachusetts legalizing gay marriage in 2004 may or may not require changes to this table.) The actual codes in XMLife are integers (this goes back to the OLE Automation techniques used in the original OLE for Life Insurance standard back in 1995). They are contained in an element containing a value attribute called tc. The element may also contain text, which is considered advisory. Sometimes it’s included (for debugging sessions, at least), and other times it’s not. A typical type code element is shown in Figure N. The user thinks in terms of the text. It would be awful to make the user remember the number. (“Dang, was unisex 3 or 4?”) So we need to present the user with human readable choices. The correspondence between numbers and names isn’t in any schema, but it does exist in an ACORD-supplied file called metadata, provided as the basis for documentation and code generation. An excerpt is shown in Figure 14. P&C also has many lookup code tables, but without the separation of integer and text string.

To solve this problem, I wrote a class called TypeCodeComboBox. You give it the name of the XML file to look in. At declaration time, you pass it as s a generic parameter the type of element that it returns . At construction time, the box looks in the XML file for all of the values that type can have. It populates the combo box with the dictionary entries contains the human-readable name and the type code of all these. Then it gives back the element, fully formed and populated based on the user’s choice. P&C does mostly the same thing. It’s in the Client project in the code samples.

If you don’t want all the choices in the XML file, you have several options. For example, the list of states accepts the complete list verbatim, including provinces of Canada and prefectures of Japan. You can change the underlying data file so that the user only sees the choices that she should see (for example, here in MA, the small agency from which I get most of my insurance is licensed only in MA and NH). Or you can write code that adds and removes elements. Certainly there will be requests. I put it out here as a straw man.

One problem I encountered with this box is that because of the generic parameters that make it strongly typed, I couldn’t place it on the design surface in VS 08. The designer doesn’t know what to pass to create one, so it petulantly doesn’t create it and shows an error instead. If you want to do layout, you have to do it with a plain combo box, which then raises the issue of how to replace it with the new combo box. So I put a replacement method on it for this purpose.

9. Future Directions

We know that this first iteration of the MS IVC ASF is small, though hopefully still useful. What we wanted to do, and I would respectfully submit have done, is to give you a straw man to poke at, something about which you can say, “Hey, I like this, but I don’t greatly care for that, and I really need such-and-such which you haven’t even mentioned.” The future direction of the package depends of course on you, the readers and the users of it. We’ve given some thought to what we THINK are the most important evolutions. But at the end of the day, it really only matters what YOU think, what you’d most like to see. What would make your life the easiest? Here are some of our initial thoughts:

A. Slimming The Schema

The ACORD standards are very large, covering as they (attempt to) do the entire insurance industry. Any given application is using only a tiny fraction of it at any given time. For example, the 228-10 example that comes with this package uses only about 30 of the 773 separate classes and about 7 of the 444 enumerations in the 2.13 schema. Admittedly this is a simple case; I chose it as the sample application for that very reason. But we’re using about 3% of the schema, and carrying the other 97% of it around for no reason. If nothing else, it slows down Visual Studio greatly.

The same applies on the P&C side as well. Previous experiments have shown that any given family of messages (HomePolicy, for example) uses about 1/3 of the schema if you use every field in it, which no one ever does. You could probably get it down to 20% for any specific application if you tried.

So essentially every app could benefit from schema slimming, and a little of it would go a long way. That’s one idea for future development.

B. Guidance Packages

A Guidance Package is basically a set of code generation tools that gets packaged into Visual Studio. It adds pieces of code that you need. For example, a wizard to generate a WCF application and a test client would be the sort of thing you would find in a GP. It would ask questions (what type of project, where to find the schema files, perhaps what business messages you wanted in it) and then generate a project type for you. Perhaps you’d be able to add or remove business messages after the fact. Perhaps the schema slimmers would operate along with it. What would you like to see here?

C. Additional wrapper classes – Client Side

We’ve seen useful wrapper classes, with constructors and smart arrays as in section 6. When you generate a proxy for a WCF client, none of these enhancements is described in the WSDL, so none of them appears on the client-side proxy. Anyone wanting to write a client program would find them useful. Perhaps we could work on taking the server-side wrapper class enhancements and figuring out a way to splice them into the client side code.

D. Future ACORD Projects

ACORD’s releases since around summer of 2001 (the P&C standard) have been evolutionary rather than revolutionary. They add a few fields, deprecate a few others and so on. However, recent developments suggest that there are about to be some major new changes in the ACORD standard. For example, AWSP 1.0, currently in candidate recommendation, is coming out soon. It officially deals for the first time with the way the ACORD messages compose with SOAP. Version 1.0 uses Soap with Attachments, and its planned successor AWSP 2.0 plans to use MTOM. Also, the P&C 2.0 spec is coming out soon, which moves away from the all-encompassing envelop approach towards the notion of different top-level elements for each message. I know that the P&C community has been waiting for AWSP to ensure that they’ll properly compose with it. The life standard’s plans aren’t known at this time. But the point is that they are coming, and it would be handy if the IVC were ready for them sooner rather than later.

10. Conclusions

Microsoft WCF is a good way of implementing ACORD XML standards, now and in the future. It requires some amount of work, which would make sense to have done once and for all by a central agency. The Microsoft IVC does this. It is a set of classes which does good things. It would be nice, for future versions, to be able to slim the schema, to be able to have guidance packages to generate code in accordance with choices, and to keep up with future ACORD releases.



Figure 1

Combination of patterns, classes, tools, documentation, and samples that make up a software factory.



Figure 2

Sample client program

Figure 3

Service interface definition, Life standard

 [ServiceContract(Name = "ProducerLicensingAndAppointments",

 Namespace = "http://ACORD.org/Standards/Life/2")]

 public interface IProducerLicensingAndAppointments

 {

 /// <summary>

 /// Used to return licensing information for a party that is a

 /// producer.

 /// </summary>

 /// <param name="request">TXLife Request Data</param>

 /// <returns>TXLife Response Data</returns>

 [OperationContract(Name = "GetProducerLicensingDetails")]

 [FaultContract(typeof(GetProducerLicensingDetailsFt\_Type))]

 [return: MessageParameter(Name = "TXLifeResponse")]

 TXLifeResponse\_Type GetProducerLicensingDetails(

[MessageParameter(Name = "TXLifeRequest")]

TXLifeRequest\_Type request);

Figure 4

Service behavior class for Life standard

[ServiceBehavior(Name = "ProducerLicensingAndAppointmentsService",

 Namespace = "http://ACORD.org/Standards/Life/2")]

public class ProducerLicensingAndAppointmentsService :

 IProducerLicensingAndAppointments

{

 #region IProducerLicensingAndAppointments Members

 public TXLifeResponse\_Type RequestProducerLicense(

TXLifeRequest\_Type request)

 {

 throw new Exception(

"The method or operation is not implemented.");

 }

 public TXLifeResponse\_Type GetProducerLicensingDetails(

TXLifeRequest\_Type rq)

 {

 // Create a TxLifeResponse based on the incoming TxLifeRequest.

 // Constructor automatically copies GUID and all other data

 // that the ACORD spec says must always be echoed, and sets

 // required dates/times to Now.

 TXLifeResponse\_Type rs = new TXLifeResponse\_Type(rq);

 // If the main transaction isn't a 228, or the subtransaction a

 // 22810,then this request has been sent to the wrong place.

 // Return an error code saying that.

 if (rq.TransType.tc != OLI\_LU\_TRANS\_TYPE\_CODES\_TC.Item228

 || rq.TransSubType.tc != TRANS\_SUBTYPE\_CODES\_TC.Item22810)

 {

 rs.TransResult = new TransResult\_Type(

RESULT\_CODES\_TC.Item5, string.Empty);

 rs.TransResult.ResultInfo.Add(new ResultInfo\_Type(

RESULT\_INFO\_CODES\_TC.Item100,

"Transaction not supported by this service"));

 return rs;

 }

 Etc.

Figure 5 Web.config file specifying behavior of service

 <service behaviorConfiguration="ProducerLicensingAndAppointmentsServiceBehavior"

 name="ProducerLicensingAndAppointmentsService">

 <endpoint address ="" binding="basicHttpBinding"

 bindingConfiguration="ProducerLicensingAndAppointmentsServiceBinding"

name="ProducerLicensingAndAppointments"

bindingNamespace="urn:CompanyX.Insurance.La.ServiceContracts"

contract="CompanyX.Insurance.La.ServiceContracts.IProducerLicensingAndAppointments">

 <identity>

 <dns value="localhost"/>

 </identity>

 </endpoint>

 <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>

 </service>

Figure 6

Soap Document Attribute producing desired soap in .asmx web service

[SoapDocumentMethod (

 RequestNamespace = "http://ACORD.org/Standards/Life/2",

 RequestElementName="TXLife",

 ResponseNamespace = "http://ACORD.org/Standards/Life/2",

 ResponseElementName="TXLife",

 ParameterStyle=SoapParameterStyle.Bare)

 ]

[WebMethod()]

public TXLife\_Type LicenseCheck(TXLife\_Type rq)

{

Figure 7

SOAP with ACORD element as sole Body content

<soap:Envelope ">

<soap:Body ">

 <TXLife xmlns="**http://ACORD.org/Standards/Life/2**>

<TXLifeRequest >

 <TransRefGUID>**0cc00168-0c56-4758-ab1c-3cf2f134508a**</TransRefGUID>

 <TransType tc="**228**"></TransType>

 <TransSubType tc="**22810**"></TransSubType>

</TXLifeRequest>

 <TXLife>

</soap:Body>

</soap:Envelope ">

Figure 8

TxLife Response constructor that copies echoed information from TxLifeRequest

// Parameterized constructor. Copy required information from

// initial request, and add required TransResult

public TXLifeResponse\_Type(TXLifeRequest\_Type rq)

{

 // Copy required echo portions of request for which

 // this is the response

this.TransRefGUID = rq.TransRefGUID;

 this.TransType = rq.TransType;

 // Add other required elements for this response type

 this.TransExeDate = DateTime.Now;

 this.TransExeTime = DateTime.Now;

 this.TransResult = new TransResult\_Type();

 // Copy all of the reference types that the Life spec says should

 // be echoed. If any element is not present, copying the null is

 // OK.Note that even elements that are called out in the Life

 // spec as primitive types that seem like they should be value

 // types here (MaxRecords as an int, PendingResponseOK as a

 // boolean, etc.), this isn't the case in this serialized class.

 // MaxRecords is a string in our class, converted by the

 // serializing code to an int, and PendingResponseOK is a complex

 // type containing a tc and a value.

 this.TransSubType = rq.TransSubType;

 this.TransMode = rq.TransMode;

 this.InquiryLevel = rq.InquiryLevel;

 this.IllustrationRequest = rq.IllustrationRequest;

 this.MIBRequest = rq.MIBRequest;

 this.MaxRecords = rq.MaxRecords;

 this.PendingResponseOK = rq.PendingResponseOK;

 this.NoResponseOK = rq.NoResponseOK;

 this.TestIndicator = rq.TestIndicator;

 this.URLTargetRequestInd = rq.URLTargetRequestInd;

 }

}

Figure 9

XML Showing a required ResultCode element and common optional ResultInfo element.

<TransResult>

<ResultCode tc="**2**">**Success with Info**</ResultCode>

<ResultInfo>

<ResultInfoCode tc="**4000**">**Agent Can Sell**</ResultInfoCode>

</ResultInfo>

</TransResult>

 Figure 10

TransResult parameterized constructors for various situations. The lowermost

one also accepts parameters for an internal ResultInfo

public partial class TransResult\_Type

{

 // Empty constructor. Create holder for ResultInfo objects

 public TransResult\_Type()

 {

 this.ResultInfo = new

 System.Collections.Generic.List<ResultInfo\_Type>();

 }

 public TransResult\_Type(RESULT\_CODES rc)

 : this()

 {

 this.ResultCode = rc;

 }

 public TransResult\_Type(RESULT\_CODES\_TC tc, string value)

 : this()

 {

 this.ResultCode = new RESULT\_CODES(tc, value);

 }

 public TransResult\_Type(RESULT\_CODES\_TC tc, string value,

 RESULT\_INFO\_CODES\_TC ritc, string rivalue) : this()

 {

 this.ResultCode = new RESULT\_CODES(tc, value);

 this.ResultInfo.Add (new ResultInfo\_Type (ritc, rivalue)) ;

 }

 }

Figure 11

TransResult class containing generic list

public partial class TransResult\_Type

{

 private RESULT\_CODES resultCodeField;

 private string confirmationIDField;

 private string recordsFoundField;

 // private ResultInfo\_Type[] resultInfoField;

 private System.Collections.Generic.List<ResultInfo\_Type>

 resultInfoField;

}

Figure 12

Soap Produced by ASF Sample Program

<soap:Envelope >

<soap:Body>

<GetProducerLicensingDetails xmlns="**http://ACORD.org/Standards/Life/2**">

<TXLifeRequest>

<TransRefGUID>

**0cc00168-0c56-4758-ab1c-3cf2f134508a**

</TransRefGUID>

<TransType tc="**228**"/>

<TransSubType tc="**22810**"/>

<OLifE>

 Etc…

</OLifE>

</TXLifeRequest>

</GetProducerLicensingDetails>

</soap:Body>

</soap:Envelope>

Figure 13

Sample wrapper class using MessageContract attributes

**[MessageContract (IsWrapped=true, WrapperName="TxLifeRequest",** WrapperNamespace= "http://ACORD.org/Standards/Life/2")]

public partial class TXLifeRequest\_Type {

 private string transRefGUIDField;

 /// <remarks/>

 **[MessageBodyMember(Order=0)]**

 public string TransRefGUID {

 get {

 return this.transRefGUIDField;

 }

 set {

 this.transRefGUIDField = value;

 }

 }

}

Figure 14

Excerpts from ACORD Metadata file used by TypeCodeComboBox

<DataType id="D\_OLI\_LU\_GENDER" deprecated="No">

 <TypeName>OLI\_LU\_GENDER</TypeName>

 <TypeTitle>Gender Type</TypeTitle>

 <Desc>

 <Description>

 <p>Gender Type</p>

 </Description>

 </Desc>

 <RevisionHistory>

 <LastRevision date="2002-07-02" ver="2.5.99.01"/>

 </RevisionHistory>

 <Enumeration>

 <BaseType type="D\_typecode"/>

 <Codes>

 <Code deprecated="No">

 <CodeName>OLI\_UNKNOWN</CodeName>

 <Value>0</Value>

 <Desc>

 <Description>

 <p>Unknown</p>

 </Description>

 </Desc>

 <SublistName>Common</SublistName>

 </Code>

 <Code deprecated="No">

 <CodeName>OLI\_GENDER\_MALE</CodeName>

 <Value>1</Value>

 <Desc>

 <Description>

 <p>Male</p>

 </Description>

 </Desc>

 </Code>

Etc.