Autodesk University 2008

A Pattern for Storing Structured Data in AutoCAD® Entities

Jeffery Geer – RCM Technologies

CP401-2 Xrecords provide storage of information in AutoCAD entities, but defining that structure can be a cumbersome task. Together we'll explore an approach for storing structured information in an Xrecord that lets your structure change as your data requirements change, without redefining the Xrecord data. Learn how to model entity data as custom .NET classes, and then to persist this data to AutoCAD entities using the built-in serialization capabilities of the .NET Framework. This a flexible approach to storing complex data structures as extended data on AutoCAD entities.

About the Speaker:
Jeff is a Senior Consultant for RCM Technologies working in the Enterprise Integration Solutions division. He has successfully developed Microsoft technology-based solutions for the engineering, manufacturing, health care, communications and education business sectors. Jeff has been programming using the Microsoft® .NET Framework for over eight years and has 15 years experience with the AutoCAD® product family.
jeff.geer@rcmt.com

Introduction

AutoCAD provides a number of ways for storing custom data inside drawing databases; block attributes, Xdata, Xrecords and custom AutoCAD entities. In this session we will focus on Xrecords as a means for storing data structures with standard AutoCAD entities. This technique provides a way to develop sophisticated custom applications on top of AutoCAD using only the Managed ObjectARX API. We will also see how Microsoft .NET serialization takes care of the heavy lifting needed to represent these data structures in a way that can be stored on AutoCAD entities.

All code for this session is presented in C#, however Visual Basic.NET could also be used to accomplish the same functionality.

Session Goals

During this session, we will discuss the technology of .NET serialization and the two serialization formatters provided by the .NET Framework; XML and Binary serializers.

We will explore the AutoCAD extension dictionary, which is a container used to store database objects on an AutoCAD entity, and we will look at the Xrecord as the class best suited for storing our data structures.

We'll also discuss ways for handling difficulties that come into play with using the Managed ObjectARX classes, which are not serializable.

.NET Framework Serialization

Background

Serialization is the process of converting the state of an object into a form that can be persisted. Classes that are useful for serializing and de-serializing objects can be found in the *System.Runtime.Serialization* and *System.Xml.Serialization* namespaces of the .NET Framework.

Serialization is at the heart of the Windows Communication Foundation (WCF), where objects are persisted, sent over the wire, and then reconstituted on another computer to carry out some remote task. Here, we'll leverage the same dehydrate/rehydrate functionally to save the state of our custom objects in AutoCAD.

In order to make a class serializable, it must first be marked with the serializable attribute:

```
[Serializable]
public class ObjectBase
{
  ...
}
```

Note:

If a class inherits from another class, the parent class must also be marked with the serializable attribute.

Binary vs. XML Serialization

The .NET Framework supports two types of serialization, binary serialization and XML serialization.

Binary serialization converts the state of an object based upon its public and private fields into a series of bytes. The process ignores the public properties used to get or set these fields.

XML serialization, on the other hand, looks to the public properties and the public fields of an object to persist its state. XML serialization relies upon an XML schema definition document (XSD) file, which specifies the structure of the XML that will represent the object. The Web Service functionality of the .NET Framework is built upon this type of XML serialization.

Since binary serialization represents a class as an array of bytes, this representation is more compact then it would be using XML, and it provides a more efficient dehydrate and rehydrate mechanism. Throughout this session, we will be using binary serialization as the mechanism for saving our object state.

Binary Serialization

```
    using System.Runtime.Serialization;
    using System.Runtime.Serialization.Formatters.Binary;
    ...

    private void SerializeBinary()
    {
       try
       {
          BaseBinary objectB = new BaseBinary();
          SetPropertiesB(objectB);

          IFormatter formatter = new BinaryFormatter();

          using (Stream stream = new FileStream(Application.StartupPath +
 "\\BaseBinary.bin", FileMode.Create, FileAccess.Write, FileShare.None))
          {
             formatter.Serialize(stream, objectB);
             stream.Close();
          }
       }
       catch (Exception ex)
       {
          MessageBox.Show(ex.Message);
       }
```

```
    private void DeserializeBinary()
    {
       try
       {
          IFormatter formatter = new BinaryFormatter();
          BaseBinary objectB = null;

          using (Stream stream = new FileStream(Application.StartupPath +
 "\\BaseBinary.bin", FileMode.Open, FileAccess.Read, FileShare.None))
          {
             objectB = (BaseBinary)formatter.Deserialize(stream);
             stream.Close();
          }

          SetFormFieldsB(objectB);
       }
       catch (Exception ex)
       {
          MessageBox.Show(ex.Message);
       }
    }
```

Note:

During binary deserialization, the constructor is not called for performance considerations.

Xml Serialization

```
   using System.Xml.Serialization;
   ...

   private void SerializeXml()
   {
      try
      {
         BaseXml objectX = new BaseXml();
         SetPropertiesX(objectX);

         XmlSerializer formatter = new XmlSerializer(typeof(BaseXml));

         using (Stream stream = new FileStream(Application.StartupPath +
"\\BaseXml.xml", FileMode.Create, FileAccess.Write, FileShare.None))
         {
            formatter.Serialize(stream, objectX);
            stream.Close();
         }
      }
      catch (Exception ex)
      {
         MessageBox.Show(ex.Message);
      }
   }
```

```
      private void DeserializeXml()
      {
         try
         {
            XmlSerializer formatter = new XmlSerializer typeof(BaseXml));
            BaseXml objectX = null;

            using (Stream stream = new FileStream(Application.StartupPath +
"\\BaseXml.xml", FileMode.Open, FileAccess.Read, FileShare.None))
            {
               objectX = (BaseXml)formatter.Deserialize(stream);
               stream.Close();
            }

            SetFormFieldsX(objectX);
         }
         catch (Exception ex)
         {
            MessageBox.Show(ex.Message);
         }
      }
```

Notes:

XML serialization requires the presence of a default constructor in the class definition.

The XSD.exe /c command is useful for creating classes from XSD documents with the appropriate attributes applied to public properties, so that the serialized class conforms to the XSD specification.

Selective Serialization

At times, it may be desirable to exclude certain fields within a class from the serialization process, for example, when the fields are pointers to unmanaged memory. This exclusion can be achieved by marking a field with the Nonserialized attribute:

```
 [NonSerialized]
private Dictionary<string, object> _objectCache;
```

Note:

To gain further control over the serialization process, your class can also implement the ISerializable interface. With this technique, you specify the data that is serialized to the stream and can perform substitutions if necessary.

Version Tolerant Serialization (VTS)

Version Tolerant Serialization (VTS) is a set of features introduced in .NET Framework 2.0 that makes it easier, over time, to modify serializable types. This functionality is a great strength of the .NET Framework, and it allows you to add fields to classes without breaking compatibility with older versions whose states have already been serialized.

VTS provides a developer with the following features:

Tolerance of missing data – this allows applications to use a version of a type that contains more fields than the type used during the initial serialization. These fields can be marked with the OptionalField attribute, so that during the deserialization process, the missing fields will not throw an exception.

Serialization callbacks – this allows an application to explicitly set the default value of a field which is missing on older versions of a serialized type. Callback can be set before and after the serialization or deserialization process.

```
    public enum PlacementType : byte
    {
       Top = 0, ...
    }

    [Serializable]
    public class BaseBinary
    {
       #region Fields

       private string _caption;
       private int _option;
       private Color _color;

       [OptionalField(VersionAdded=2)]
       private PlacementType _placement;

       #endregion

       #region Non-Serialized Fields…

       #region Public Properties…

       #endregion

       public BaseBinary()
       {
          _caption = string.Empty;
          _option = 0;
          _color = Color.Blue;
          _objectCache = new Dictionary<string, object>();
          _placement = PlacementType.Top; //notice that top is not used on
the deserialized v1.0 object
       }

       [OnDeserializing]
       public void SetOptionalFieldDefaults(StreamingContext sc)
       {
          _placement = PlacementType.Left; // will be used for the
deserialized v1.0 object
       }
    }
```

Serialization Best Practices

To ensure proper versioning behavior, follow these rules when modifying a type from version to version:

- Never remove a serialized field.
- Never apply the NonSerialized attribute to a field if the attribute was not applied to the field in the previous version.
- Never change the name or the type of a serialized field.
- When adding a new serialized field, apply the OptionalField attribute.
- When removing a NonSerialized attribute from a field (that was not serializable in a previous version), apply the OptionalField attribute.
- For all optional fields, set meaningful defaults using the serialization callbacks unless 0 or null as defaults are acceptable.

Xrecords

Xrecords enable you to store application specific information in an AutoCAD drawing without the need for defining your own custom ObjectARX class. Using Xrecords will allow us to build our solution entirely in the Managed ObjectARX API. Although Xdata could also be used to store information on entities, its size is limited to 16K, which may present a problem depending upon the size of the data structure being saved.

Xrecord DXF group codes

The Xrecord is comprised of a result buffer chain, which is a list of TypedValue items. The TypedValue object pairs a DXF Code with its associated data. The DXF Code indicates the type of data contained in the item. Below is a partial list of DXF Codes:

DFX Code	**enum DatabaseServices.DxfCode**
1	Text
40	Real
70	Int16
90	Int32
310	BinaryChunk

One of the challenges of using Xrecords is defining and maintaining the mappings between variables and the various data types that can be represented in an Xrecord.

We intend to use the BinaryChunk DXF code to store a serialized version of our custom .NET class in the Xrecord. The .NET framework will take care of mapping the class fields and properties and we will place the serialized bytes into the Xrecord.

Chunking

Before we can store the binary information in an Xrecord, however, we must break our binary data into a series of chunks where the maximum length of any chunk is 127 bytes.

Note:

Through some experimentation, I've found that a chunk no greater than 255 bytes in length will work, but this isn't documented in the ObjectARX API specifically for Xrecords. I've taken my lead from the documentation written for binary chunks as it applies to Xdata and this documentation indicates 127 max lengths.

The following code takes a MemoryStream returned from serialization and splits the bytes contained into a jagged byte array with maximum 127 byte lengths:

```
    private const int MAX_CHUNK_LENGTH = 127;

    public static byte[][]ChunkStream(MemoryStream stream)
    {
       // determine the number of chunks needed to hold stream
       int chunkCount = (int)Math.Ceiling((decimal)stream.Length /
 (decimal)MAX_CHUNK_LENGTH);

       byte[][] result = new byte[chunkCount][];

       // start read at beginning of stream
       stream.Position = 0;

       int chunkIndex = 0;
       int numBytesToRead = (int)stream.Length;
       int numBytesRead = 0;

       // read through the stream, assigning each chunk of bytes into the
       // next sequential byte array
       while (numBytesToRead > 0)
       {
          int chunk = MAX_CHUNK_LENGTH;
          if (chunk > numBytesToRead)
          {
             chunk = numBytesToRead;
          }

          byte[] buffer = new byte[chunk];
          int n = stream.Read(buffer, 0, chunk);

          if (n != 0) // make sure we haven't passed the end of the stream
          {
             result[chunkIndex] = buffer;
             chunkIndex += 1;
          }
          else
          {
             break; // we've reached the end
          }

          numBytesToRead -= n;
          numBytesRead += n;
       }

       return result;
    }
```

Now we are ready to place these bytes into an Xrecord and save them with an AutoCAD entity. Our entity is represented below as the dbObj argument, which must be opened ForWrite:

```
    internal static ObjectId StoreObjectInExtensionDictionary
       (string xRecordKey, DBObject dbObj, Transaction trans, BaseBinary o)
    {
       // remember the handle of the DBObject.
       o.DBObjectHandle = dbObj.Handle;

       ObjectId result = ObjectId.Null;

       using (MemoryStream stream = new MemoryStream())
       {
          BinaryFormatter serializer = new BinaryFormatter();
          serializer.Serialize(stream, o);
          byte[][] objectChunks = ChunkStream(stream);

          if (objectChunks.Length > 0)
          {
             List<TypedValue> typedValues = new List<TypedValue>();

             // identify the type name in a readable section
             typedValues.Add(new TypedValue((int)DxfCode.Text,
o.GetType().FullName));

             // add sequence of binary chunks to the result buffer
             for (int i = 0; i < objectChunks.Length; i++)
             {
                typedValues.Add(new TypedValue((int)(DxfCode.BinaryChunk),
objectChunks[i]));
             }

             ResultBuffer data = new ResultBuffer(typedValues.ToArray());

             result = SaveXRecord(obj, data, xRecordKey, trans);
          }
       }

       return result;
    }
```

Note:

It is useful to record the Handle of the AutoCAD entity along with the serialized object, so that the entity can later be identified when all you have in memory is the object. The above DBObjectHandle property wraps a field which is defined as a long, since AutoCAD Handles are not serializable.

Dictionaries

Xrecords are contained in a dictionaries; either the Named Object Dictionary or in an extension dictionary. We will use the extension dictionary of our entity to hold the Xrecord:

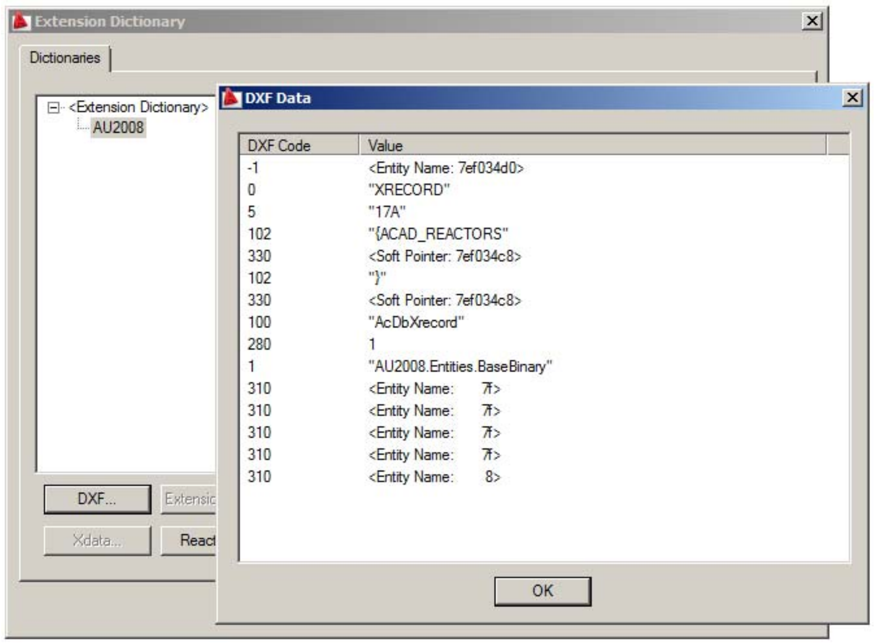
```
    private static ObjectId SaveXRecord
        (DBObject o, ResultBuffer buffer, string key, Transaction trans)
    {
        if (o.ExtensionDictionary == ObjectId.Null)
        {
            o.CreateExtensionDictionary();
        }

        using (DBDictionary dict = trans.GetObject(o.ExtensionDictionary,
OpenMode.ForWrite, false) as DBDictionary)
        {
            // check to see if dictionary contains XRecord
            // if so, update the data - important for Undo Operations
            if (dict.Contains(key))
            {
                Xrecord xRecord = (Xrecord)trans.GetObject(dict.GetAt(key),
OpenMode.ForWrite);
                xRecord.Data = buffer;
                return xRecord.ObjectId;
            }
            else
            {
                Xrecord xRecord = new Xrecord();
                xRecord.Data = buffer;

                dict.SetAt(key, xRecord);
                trans.AddNewlyCreatedDBObject(xRecord, true);
                return xRecord.ObjectId;
            }
        }
    }
```

ArxDbg (Snoop Entities)

Once the entity has been stamped with data, we can use the ArxDbg tool to inspect the entity and see how our custom object has been saved:



DXF Code	Value
-1	<Entity Name: 7ef034d0>
0	"XRECORD"
5	"17A"
102	"{ACAD_REACTORS"
330	<Soft Pointer: 7ef034c8>
102	"}"
330	<Soft Pointer: 7ef034c8>
100	"AcDbXrecord"
280	1
1	"AU2008.Entities.BaseBinary"
310	<Entity Name: 7f>
310	<Entity Name: 7f>
310	<Entity Name: 7f>
310	<Entity Name: 7f>
310	<Entity Name: 8>

In the above example, our class was serialized into a binary stream that was just over ½ kb. Even though the number of fields was minimal for the custom BinaryBase object, space is needed to store the mapping information for the fields, as well as to define any external assemblies which are referenced by our class.

Reading the Xrecord

To get the data back out of the entity's extension dictionary, we essentially reverse the process with addition. We have structured our solution in such a way that the custom class BaseBinary resides in its own assembly. The serialization logic is contained in a different assembly and will need the assistance of a SerializationBinder in order to reconstruct our class.

Doman Binder

The DomainBinder class inherits from the SerializationBinder type and includes the logic for finding our assembly, which is loaded in the current application domain. During binding, the DomainBinder reads the short assembly name from the manifest information stored in the stream, disregarding the assembly version information:

```
   internal class DomainBinder :
System.Runtime.Serialization.SerializationBinder
   {
      public override Type BindToType(string assemblyName,
                                                  string typeName)
      {
         string shortAssemblyName = assemblyName.Split(',')[0];

         // search all loaded assemblies for our serialized type
         Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
         for (int i = 0; i < assemblies.Length; i++)
         {
            if (shortAssemblyName == assemblies[i].FullName.Split(',')[0])
            {
               return assemblies[i].GetType(typeName);
            }
         }

         return null;
      }
   }
```

Reversing the Process

With everything now in place, we can now rehydrate our object from the information stored in the extension dictionary of our object. To do this we first get the entity and read the Xrecord with our key from the extension dictionary. Next we take all of the binary information stored in our Xrecord and rebuild a MemoryStream from our binary chunks. This stream is then passed into the BinaryFormatter.Deserialize method and we now have our object with its previous state:

```
   BinaryFormatter serializer = new BinaryFormatter();
   // allow serializer to bind to all assemblies in application domain
   serializer.Binder = new DomainBinder();

   using (Xrecord xrec = FindXRecordInExtensionDictionary(xRecordKey,
dbObj, trans))
   {
      if (xrec != null)
      {
         int count = 0;
         using (ResultBuffer rb = xrec.Data)
         {
            if (rb != null)
            {
               using (MemoryStream stream = new MemoryStream())
               {
                  TypedValue[] tvs = rb.AsArray();
                  if (tvs != null)
                  {
                     //  first item should be the Type name of object
stored as binary in Xrecord
                     TypedValue typeRecord = tvs[0];
                     if (typeRecord.TypeCode == (short)DxfCode.Text)
                     {
                        for (int i = 1; i < tvs.Length; i++)
                        {
                           if (tvs[i].TypeCode ==
(short)DxfCode.BinaryChunk)
                           {
                              byte[] buffer = (byte[])tvs[i].Value;
                              count = buffer.Length;

                              stream.Write(buffer, 0, count);
                           }
                        }
                        // reset stream position
                        stream.Position = 0;

                        // safe cast will set result to null if object is
not of correct type
                        result = serializer.Deserialize(stream) as
BaseBinary;
                     }
                  }
               }
            }
         }
      }
   }
```

References

MSDN .NET Framework Developer’s Guide – Serialization:
http://msdn.microsoft.com/en-us/library/7ay27kt9(VS.85).aspx

MSDN Library - XML Schema Definition Tool (Xsd.exe)
http://msdn.microsoft.com/en-us/library/x6c1kb0s.aspx

MSDN .NET Framework Developer's Guide – Serialization Guidelines:
http://msdn.microsoft.com/en-us/library/6exf3h2k(VS.85).aspx

MSDN Library – Version Tolerant Serialization
http://msdn.microsoft.com/en-us/library/ms229752(VS.85).aspx