The Power of ObjectARX®

Fernando Malard - Senior Developer Manager, OFCDesk LLC

CP311-5 This session for advanced AutoCAD® users and developers will present a simple and complete introduction on creating and using custom entities with the fantastic ObjectARX programming environment. The session will begin by presenting some basic concepts about ObjectARX programming. Attendees will learn how to create basic ObjectARX applications using an ARX Wizard. Next, the class will explore how the AutoCAD database works, organizes and uses its objects. The session will end with custom object concepts, code examples, and a great step-by-step example.

About the Speaker:

Fernando is a civil engineer and has worked with AutoCAD® and ObjectARX® since 1996. He has also been an Autodesk Developer Network member since then. He has worked on several AutoCAD applications for civil engineering, architectural, and GIS using ObjectARX, .NET, SQL, and Oracle® databases. Fernando has extensive experience teaching AutoCAD, C++, MFC, COM, SQL, .NET and ObjectARX during the last 12 years. Today, he continues to apply ObjectARX to complex ERP solutions and maintains a blog about ObjectARX programming. Fernando holds a Master's degree in Structural Engineering from the Federal University of Minas Gerais, Brazil.

Introduction

During the last years AutoCAD has evolved to become a generic and powerful CAD platform providing a huge development in CAD and Design industries. This evolution, in its majority, was due the reduction of the existing lack between real situations and conceptual projects. This was only possible with the AutoCAD open architecture which was introduced at AutoCAD **R13**. At that moment Autodesk® has decided to open part of AutoCAD core to developers in form of a **C++ SDK** (*Software Development Kit*) providing a powerful tool and allowing developers to create applications and solutions that can take full advantage of AutoCAD core.

ObjectARX, named at that time **ARX**, was introduced officially at release **13**. After this release, AutoCAD evolved to a powerful interface with release **14** and next, to a **MDI** (*Multiple Document Interface*) environment with release **2000**. Nowadays, AutoCAD in its release **17.2** (also known as **2009**) is going beyond all optimistic forecasts and it is providing great tools for developers like **.NET** interface through **C#** and **VB.NET** languages.

ObjectARX, through C++, stays at the top of performance and power list when compared with other AutoCAD development languages. The reason is really simple, until today the major part of AutoCAD is still coded in C++. It's true that there are parts that provide interfaces with .NET but this tier only provide an indirect access to the real C++ core.

ObjectARX takes one great advantage when compared to the other AutoCAD customization languages. Currently it is the only one that fully implements **Custom Entities** (Objects). This great capability of ObjectARX was intensively used by Autodesk itself to create its vertical products since AutoCAD **R13**. Today we can see several AutoCAD based solutions made by Autodesk using the power provided by ObjectARX. We can quote a couple of verticals like **Autodesk Architectural Desktop** and **Autodesk MAP**. These two products, for instance, use a lot of Custom Entities to bring the CAD solution near to the real industry scenario. Instead of using lines and circles to represent architectural symbols, Autodesk Architectural Desktop users are able to create specific entities like windows, doors, stairs, etc. This makes the application much more suitable to its specific industry and reduces the time spent to create specific projects.

In this class session you will be able to taste part of this power by implementing a simple **Custom Entity** inside AutoCAD. You will start by learning about ObjectARX application basics. Next you will learn some very important concepts about AutoCAD database, how its objects are organized, stored and how to manipulate them. Finally, you will learn how to create a simple Custom Entity with basic features and how to expand it to a higher level of complexity.

Note: *We will use **AutoCAD 2009**, **Visual Studio 2005** and **ObjectARX 2007** on this class. ObjectARX modules created with ObjectARX 2007 are **binary compatible** with AutoCAD 2009.*

AutoCAD Database

AutoCAD database is well organized to make easier to manipulate its objects. To better understand how things happen you may imagine it as some kind of "**object oriented database**" which receives **commands**, react to **events**, fire its own events and act as a **storage place** for objects. This database needs to operate under certain conditions and requirements. It is sometimes a little complex to manipulate but this enforces the **database integrity** and the **application stability**.

When you are creating a drawing inside of AutoCAD you are creating, deleting or changing objects inside its database. AutoCAD database is a set of objects **hierarchically organized** which represents a drawing (a DWG file). If we take as an example, a simple entity, say a LINE, it contains all the necessary data to graphically represent itself. As a vector based application, AutoCAD doesn't need to store the pixels between the start and end points for each line. It stores only both the start and end three-dimensional points (**X,Y,Z**). The line graphics are then generated by a math line equation and mapped to the screen coordinates by a translation to the **X,Y pixels**.

The process of creating a LINE is not finished by specifying its properties but it also requires you to specify where and how this LINE will be stored. Inside AutoCAD database there are several existing objects called **containers** which are intended to **store and keep other objects**. Each container has its

own procedures and rules to store and retrieve its child objects. In most cases, the container is responsible for keeping and saving its children. In our example, a **LINE** can be stored into containers made to store entities. These entity's containers are called **Block Table Records** (**BTR**). Each **BTR** can store several entities including lines, circles, texts, blocks, etc. The diagram presented in Figure 1, shows part of the AutoCAD database structure.

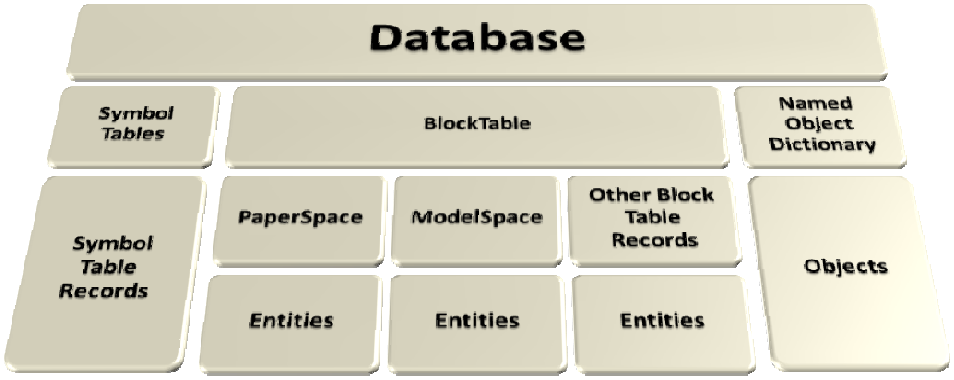


Figure 1 – AutoCAD Database structure showing entity's containers

The Figure 1 shows that entities can be stored into **Model Space**, **Paper Space** and other **Block Table Records**. These "**other**" **BTR** can be a **Layout** page or even another **Block**. Each block inserted inside AutoCAD drawing is a **Block Reference** entity which refers to the original Block which is a Block Table Record. Due that, when you edit a Block with **REFEDIT** command it reflects on all its pointed Block References because you are really editing the original Block entity objects. Next you will find a code fragment which demonstrates the outline of **LINE** creation process through a sequence of C++ code instructions which will create a **LINE** from point **(1.0, 1.0, 0.0)** to point **(10.0, 10.0, 0.0)**:

```
01      // create two points and the line
02      AcGePoint3d startPt (1.0, 1.0, 0.0);
03      AcGePoint3d endPt (10.0, 10.0, 0.0);
04      AcDbLine* pLine = new AcDbLine (startPt, endPt);
05      // open the proper entity container
06      AcDbBlockTable* pBlock = NULL;
07      AcDbDatabase* pDB = acdbHostApplicationServices()->workingDatabase();
08      pDB->getSymbolTable(pBlock,AcDb::kForRead);
09      AcDbBlockTableRecord* pBTR = NULL;
10      pBlock->getAt(ACDB_MODEL_SPACE, pBTR, AcDb::kForWrite);
11      pBlock->close();
12      // now, add the entity to container
13      AcDbObjectId Id;
14      pBTR->appendAcDbEntity(Id, pLine);
15      pBTR->close();
16      pLine->close();
```

The LINE creation process is similar to all other entities creation and it basically consists of the following steps:

- We first **Instantiate a pointer** to the desired entity class (some constructors will allow or require you to pass some basic parameters);
- Next, we need to **open the Block Table** where we will find the desired entity container. In this example we will use the Model Space container;
- After that, we **open the Model Space** (a Block Table Record) and append to it the recently instantiated entity.

It is very important to note that this process requires you to **open and close each specific object** to have access to its properties and methods. Once you add the new entity to database it will receive a unique identifier, provided automatically by AutoCAD database, called **AcDbObjectId**. This is the key to access the entity whenever you want and it is valid and kept unchanged only during the AutoCAD drawing session (database lifecycle).

This is just the outline process to make clear to you the basic steps required to create a simple entity and to show the basic protocol to access some database objects and procedures. Next you will learn how to create an ObjectARX application project using Visual Studio.

ObjectARX Application Overview

The ObjectARX application is in fact a **DLL** (***Dynamic Link Library***). This allows it to be loaded at runtime inside AutoCAD. This concept has evolved to the **Object Enabler** which was introduced some versions ago. This splits the application out to two modules. The Object Enabler is an ObjectARX application with only a set of custom objects/entities classes. It is called **ObjectDBX** and it is compatible to all Autodesk applications which are built at the top of **DWG** technology like all DWG viewers and all other AutoCAD verticals. The last 3 characters of both types of application are meant to define its use: **ARX** (***AutoCAD Runtime Extension***) and **DBX** (***Database Extension***). Basically your application will be constructed by several modules of these two types:

Interfaces: They will contain all dialogs, commands, events and all further AutoCAD interface specific code. It will consist of an **ObjectARX** application DLL (**.arx** file extension) and will be compatible only with AutoCAD and its based verticals;

Custom classes: They will contain all custom classes (objects and/or entities) which do not rely on AutoCAD application presence. It will consist of an **ObjectDBX** application DLL (**.dbx** file extension) and it cannot contain any AutoCAD specific interaction or information. It can be loaded inside AutoCAD, its verticals and all **Autodesk applications** with **ObjectDBX** support;

You can split your application into several **ARX** and **DBX** modules depending on its complexity and this will make you application much more organized and flexible to expand. Imagine that you will work with a set of clients which belong to the same industry segment with some small differences among their products. You may organize your application classes to be implemented using two levels of inheritance. The first level will implement the common behavior among client's products. The next level will then implement specific client requirements and will allow you to fit your application perfectly to each client demands. The outline of your solution modules would be something like this:

- MyClientBasics.dbx;
- MyClient1.dbx, MyClient2.dbx, MyClient3.dbx, etc.

This way you can reuse the same core from client to client, and better, when you fix a problem or add a new feature to your core modules all of your clients will get improved on the next update. This is a pretty good strategy to keep your application organized, up to date and suitable to each client's industry business rules.

Both **ARX** and **DBX** application modules can be loaded into AutoCAD using one of the following procedures:

- **ARX** command;
- **acad.rx** text file;
- **APPLOAD** command;
- **arxload()** AutoLISP method or **acedArxLoad()** ObjectARX method;
- AutoCAD **–ld** or **/ld** startup option;
- **Demand loading** registry process.

These procedures may be used on different situations depending on your application scenario requirements and deployment.

Using ARXWizard – Exercise 1

The ObjectARX SDK comes with a very nice **Visual Studio plug-in** called **ARXWizard**. This plug-in helps you not only to create an **ARX** or **DBX** application module but it also helps a lot with the creation process of your custom classes. The ARXWizard installation module is located at **\ObjectARX 2007\utils\ObjARXWiz** folder. You just need to run the install (with all Visual Studio instances closed) and it will create a new project type option inside your Visual Studio "**New Project**" dialog (inside Visual C++ tree node) like shown on Figure 2.

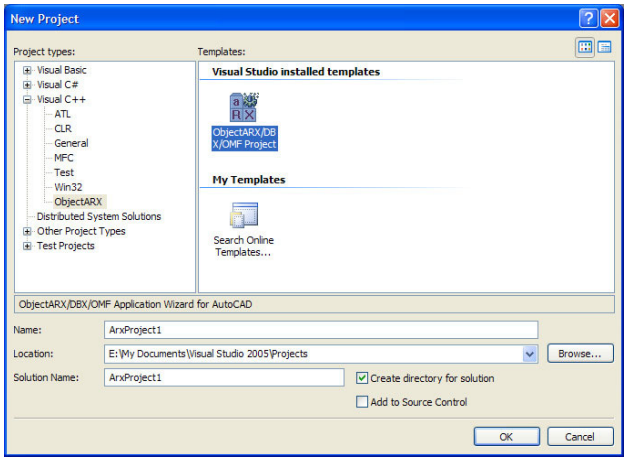
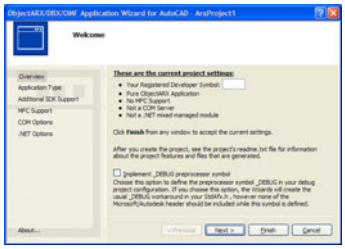


Figure 2 – Visual Studio New Project dialog

To create a new **ARX** or **DBX** module, open Visual Studio and do the following:

- Open menu **File** > **New** > **Project…**;
- Select **ObjectARX** into the **Visual C++** tree node;
- Select **ObjectARX/DBX/OMF Project** item;
- Specify the project name, location and solution's name;
- Click **OK** to proceed.

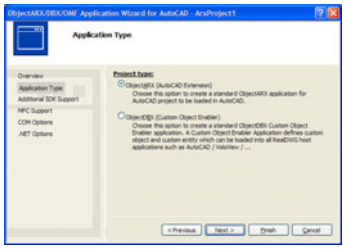
Next, you need to choose your project type (**ARX** or **DBX**), additional **SDK** support, **MFC**, **COM** and **.NET** options like described below.



Welcome

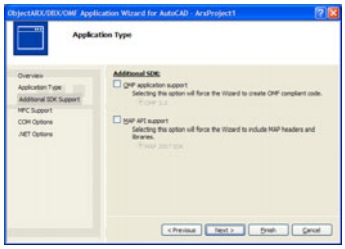
Specify your Developer symbol to be used as prefix for all classes (this avoid conflict with other ObjectARX/ObjectDBX applications).

If desired, enabled the **_DEBUG** symbol to allow custom code compilation for Debug sessions.



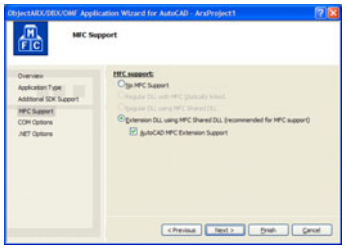
Application Type

Select your application type: ObjectARX or ObjectDBX.



Additional SDK

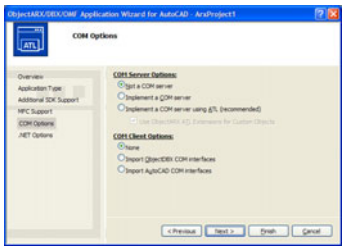
If you are planning to develop an application which will use specific features or classes from Autodesk Architectural Desktop (OMF SDK) or Autodesk MAP (MAP API) enable the desired options.



MFC Support

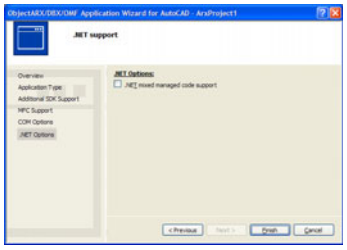
If your application will use MFC features like MFC dialogs it will require you to enabled MFC support. Further, if you also need to use AutoCAD MFC extensions, enable this option.

*The last option is recommended only to ObjectARX type applications.



COM Options

If your application will act as a COM client or COM server (ATL recommended), select the desired options in this dialog.



.NET Support

If you plan to develop a mixed-mode application enable this option. It will allow you to use the .NET Framework classes and also access the AutoCAD .NET API from inside your application.

This process will create the so called "**Application Skeleton**" which will avoid you to spend some significant time doing the project configuration manually. Once you have finished the setup process your application solution will be something like the Figure 3.

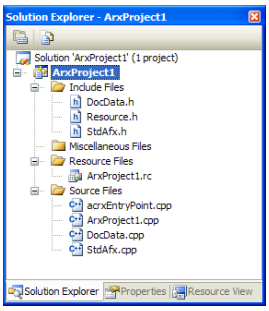


Figure 3 – Typical ObjectARX project solution

To be able to compile this project we need to add the ObjectARX include and library paths to Visual Studio Options. This needs to be done only once because it is not project specific. Go to the **Tools** > **Options…**, then open the **VC++ Directories** node. Follow the Figure 4 and 5 instructions and then click **OK** to finish.

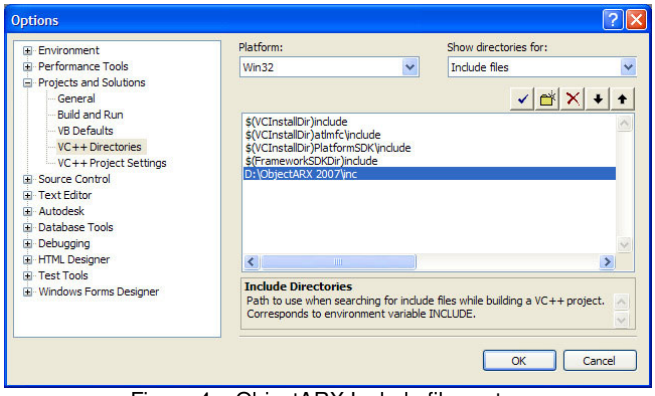


Figure 4 – ObjectARX Include files setup

In the "**Show directories for:**" combo box, first select "**Include files**" and add a new entry pointing to the "**[drive:]\ObjectARX 2007\inc**" path (Figure 4).

*[drive:] is your local drive, typically **C:**

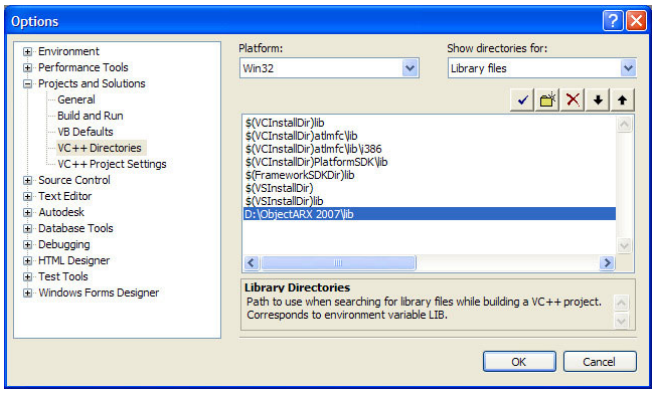


Figure 5 – ObjectARX Library files setup

Next, select "**Library files:**" at the same combo box and add a new entry path "**[drive:]\ObjectARX 2007\lib**" (Figure 5).

To compile and build your project, first change the default build configuration to **Release** mode. To do that, go to menu **Build**, choose **Configuration Manager…**, select **Release** into "**Active solution configuration**" combo box. Next, go to menu **Build**, choose **Build Solution**. You should get something like the following into **Output**/**Build** screen (Figure 6).

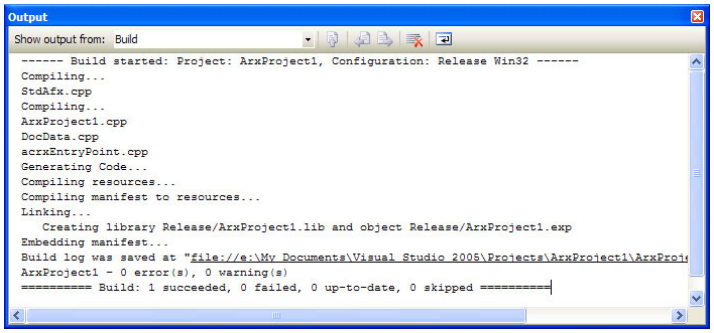


Figure 6 – Typical Build output screen

Now it is time to load your application inside AutoCAD. Open AutoCAD and then fire **APPLOAD** command. Search for the **ArxProject1.arx** file inside **ArxProject1\Release** folder, select it and click **LOAD** button.

Custom Entities Concepts

Once you have learned how to create an **ObjectARX**/**ObjectDBX** application we can dig into **Custom Entities**. As a powerful **Object Oriented** language, C++ allows developers to inherit classes providing a large scale code reuse. The inheritance concept is not one of this sessions targets but we can take a look at part of ObjectARX's class tree to better understand how AutoCAD classes are organized (Figure7).

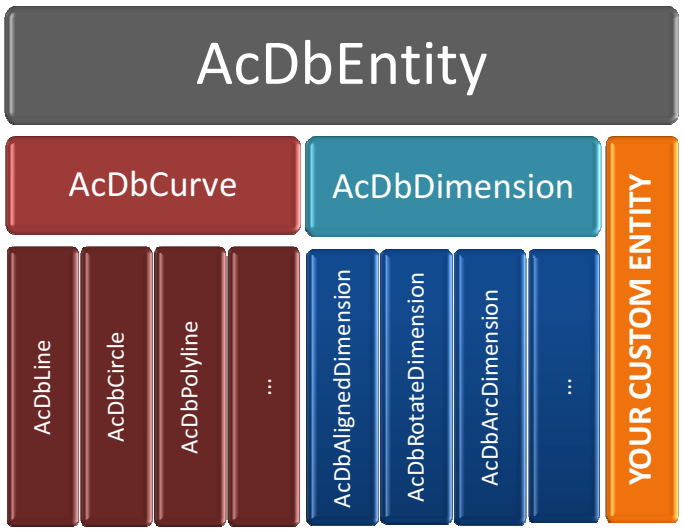


Figure 7 – Part of ObjectARX class hierarchy

The same technique used on AutoCAD vertical products is available through ObjectARX. The first step to create a custom entity is to choose from which class it will derive. Next you need to choose which methods your entity will override. When your entity class is loaded inside AutoCAD it will be inserted into ObjectARX runtime class tree and from this point it is available to be used. Your custom entity will participate, by default, into several types of operations fired by AutoCAD itself.

As an example, the method **worldDraw**() is responsible for your entity's graphics. This method has a fixed signature and it is **virtual** (can be override on derived classes) allowing your custom entity to implement it. Once AutoCAD needs to regenerate its drawing graphics it starts a process iterating over its database and calling firstly the **worldDraw**() method and, if it returns false, calling **viewportDraw**() method. But I'm sure you are thinking: "***How AutoCAD knows how to call my own custom entity's method?***" It does not know but it knows your entity inherits (for sure) from **AcDbEntity** where the **worldDraw**() method is firstly defined.

The virtual C++ mechanism will then, at runtime, forward the call AutoCAD made at **AcDbEntity** level down to the entity's instance level where your method is defined. The same process occurs to all other virtual methods. This way AutoCAD abstracts it management at base class level reflecting down to all derived classes. If your class does not implement some virtual method, it will be called at the next top class level going up and up until reach the **AcDbEntity** class. The major part of AcDbEntity's methods does nothing and by default a derived class will not make so much without your code improvements.

Creating a Custom Entity – Exercise 2 – Step 1

You will learn in this exercise how to create a **simple custom entity**. I will keep it simple as much as I can to reinforce the basic concepts involved. On the next steps we will improve this custom entity by adding great features step by step.

The first step is to name our entity and choose from which base class it will derive. Remember we have to split our application into two modules, an **ARX** (user interfaces) and a **DBX** (the custom entity class itself). These two projects will be placed into the same Visual Studio Solution and the ARX module will depend on DBX module. The names will be:

- **Solution**: **Exercise2** (Visual Studio will create a ***Exercise2.sln*** file);
- **DBX Project**: **AuCustomObjects** (Visual Studio will create a ***AuCustomObjects.vcproj*** file);
- **ARX Project**: **AuUserInterface** (Visual Studio will create a ***AuUserInterface.vcproj*** file);

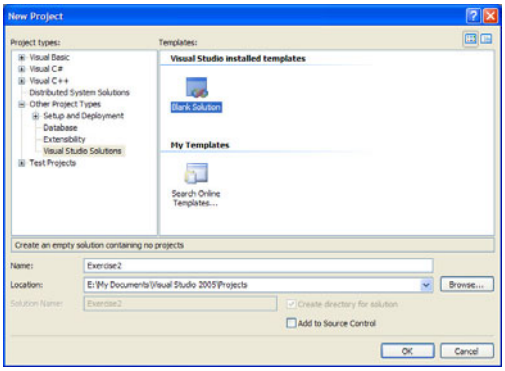


Figure 8 – Blank Solution project

Open Visual Studio, go to the menu **File** > **New Project…** Open "**Other Project Types**" node, click on "**Visual Studio Solutions**" item. It will show a template called "**Blank Solution**". Choose this template and name it as **Exercise2** (the location can be any folder you want). Click **OK** to proceed (Figure 8).

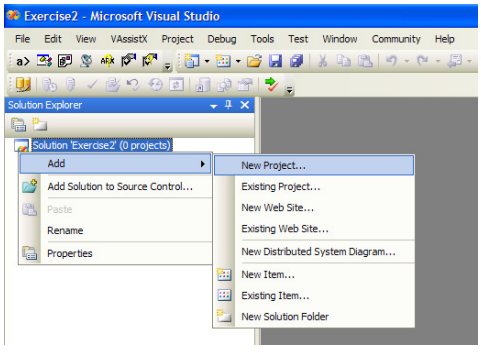


Figure 9 – Creating solution's projects

Next, right click the solution icon; select **Add** and then **New Project…**, (Figure 9). The dialog, presented on **Exercise1** will appear. Select on the list the **ObjectARX** template and create both **AuUserInterface** and **AuCustomObjects** modules. Remember to choose ARX or DBX project type accordingly.

Each project location will be created inside the existing solution by default. Don't change this location. Enable **MFC** option on both projects (don't need to enable AutoCAD MFC extensions) Enable **_DEBUG** symbol too. Don't enable any **COM** or **.NET** feature in both projects. Make **AuUserInterface** project depend on **AuCustomObjects**. To do that, **right click** project AuUserInterface and select **Dependencies…** Then mark AuCustomObjects project on the list. Click **OK**.

Right click project **AuUserInterface** again and select "**Set as Startup Project**" (it will turn to **bold**). Now test your solution building it: go to menu **Build** > **Build Solution**. You should get 2 Builds with 0 errors and (1+3) warnings that are safe to ignore.

The final test at this stage is to load the application inside AutoCAD. Remember our project AuUserInterface depends on AuCustomObjects so the DBX module needs to be loaded first than the ARX module. The unload process must be done in reverse order, AuUserInterface first and then AuCustomObjects. You should get everything working this way.

Note: *Depending on which type of build you made (**Debug** or **Release**) it is recommended to load both projects with the same compilation type.*

Now we have our **DBX** and **ARX** module it is time to add our custom entity's class. In this example, our entity will be derived from **AcDbPolyline** which represents the AutoCAD **POLYLINE** entity. The reason for this option is that our custom entity will behave almost exactly like a polyline but it will add some extra features like vertex numbers, direction symbols, hatch, etc. To add this custom class we will use the **Autodesk Class Explorer** tool which is located at ARXWizard's toolbar. It is the second button like the Firgure 10 shows.

Figure 10 – ARXWizard toolbar

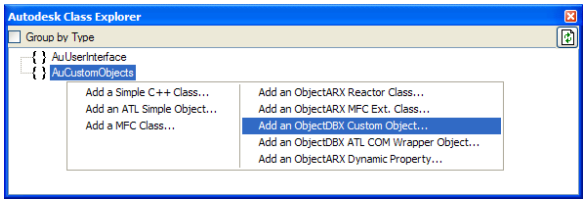


Figure 11 – Autodesk Class Explorer

Once you click on this button a dialog bar will appear allowing you to explore all existing classes into your projects. At this time, there are no custom entity classes available. Select our **DBX** module and then **right click** on it. A pop-up menu will be displayed. Select the "**Add an ObjectDBX Custom Object**" option (Figure 11).

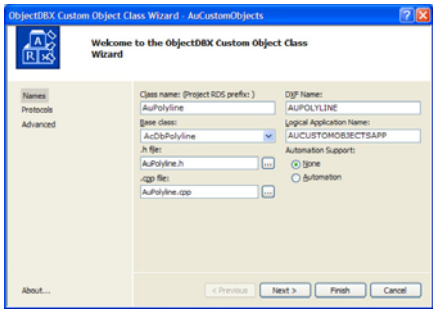


Figure 12 – Custom Object Wizard - Names

This wizard has **3 steps**. The first step, called **Names** (Figure 12), allows you to specify all basic custom entity's features. First, name it as "**AuPolyline**". Choose as base class the **AcDbPolyline** class (note you have several other classes available to derive from). Remaining fields will be filled automatically. Press **Next**.

Once you select the polyline as your base class you are saying that your custom entity will behave like a polyline except where you redefine it. This is done through the virtual methods I have mentioned before. The **Custom Object Wizard** also will help you to implement basic class features. This

is done on the second step of this Wizard, called **Protocols**. Enabling these options will instruct the wizard to add the related virtual methods simplifying the class creation process for you.

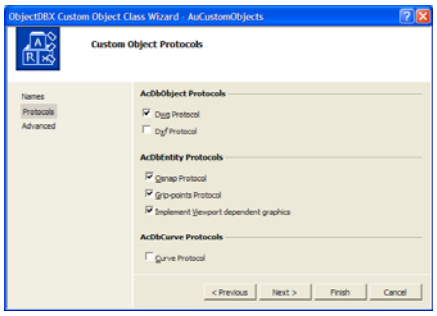


Figure 13 – Custom Object Wizard - Protocols

Inside this dialog you can specify if your entity participate on **DWG**/**DWF** protocols, if your entity implements **OSNAP** points, **GRIP** points and **viewport** dependent graphics and, at last, if it implements curve protocols. In this example we will enable only **DWG protocol** and all 3 AcDbEntity protocols which will allow us to implement some basic features (Figure 13).

The third step, called **Advanced**, will allow you to add notification and cloning features to your custom entity. In this example we will not use these features. There are many other features you can redefine through a huge set of virtual methods but to keep this example simple we will only implement these ones. Click **Finish** to create your Custom Entity's Class. Remember, it will be placed at you **DBX** module.

The Custom Entity Wizard will create default virtual methods but, in this particular case, our base class **AcDbPolyline** does not support all of these methods signatures. If you compile the solution you will get 4 error messages exactly due that. For now, we will replace the explicit **AcDbPolyline::** prefix on these 4 return calls by **AcDbCurve::** prefix which is the AcDbPolyline base class. Now you should get no errors.

If you inspect your projects Solution Explorer you will note two new files: **AuPolyline.h** and **AuPolyline.cpp**. These files, inside your **DBX** module, are responsible to declare and implement your custom entity. Open these two files and walk through the generated code.

Now you need to add a new command, inside your **ARX** module, to create an instance of your new custom entity. To do that, on the **Solution Explorer**, right click the **AuUserInterface** project and click "**Set as Startup Project**" (project name will turn to bold). Now, click on the first icon of **ARXWizard** toolbar (a icon with "**a>**" text). This button will open the "**ObjectARX Commands**" dialog box. It has two command lists. **Right click** the first list and select **New**. A new line will be added to this list with a default command called "**MyCommand1**". Click **OK**.

If you open the **acrxEntryPoint.cpp** file of **AuUserInterface** project you will find, inside class **CAuUserInterfaceApp**, the **AuUserInterface_MyCommand1()** empty method which will be called when you fire **MYCOMMAND1** command inside AutoCAD. Inside this method you will create an instance of your **AuPolyline** entity.

Remember that your custom entity class implementation is located into the **DBX** project. Due that you will need to add (through **#include** compiler instruction) a reference to the **AuPolyline** class declaration file. To do that, add the following compiler instruction at the beginning of **acrxEntryPoint.cpp** file of your ARX module right after **#include "resource.h"** line:

```
#include "..\AuCustomObjects\AuPolyline.h"
```

The "**..**\" part of this include path goes up one folder level (getting back to the Solution's root folder) and then goes into the **DBX** project folder. Now you can compile your project and shouldn't get any errors. Next, open AutoCAD, load first the DBX module and then the ARX module (there will be two text messages at AutoCAD command prompt confirming that the both load processes were succeeded. Type your project's command "**MYCOMMAND1**" and run a **ZOOM EXTENTS** command to see the

generated entity. If you also run the **LIST** command and select this entity you will see its detailed information.

The following code will create a **10 sided polyline** (closed) and will add it to the **Model Space** Block Table Record.

```
01      // - AuUserInterface._MyCommand1 command (do not rename)
02      static void AuUserInterface_MyCommand1(void)
03      {
04          // AuPolyline entity
05          AuPolyline* pL = new AuPolyline();
06          int nSides = 10;
07          double incAngle = 2*3.141592 / nSides;
08          // Add vertex list
09          for (int i=0; i<nSides; i++)
10              pL->addVertexAt(i,AcGePoint2d(10*cos(i*incAngle),
        10*sin(i*incAngle)));
11          // Set Polyline as closed
12          pL->setClosed(Adesk::kTrue);
13          // open the proper entity container
14          AcDbBlockTable* pBT = NULL;
15          AcDbDatabase* pDB = acdbHostApplicationServices()-
        >workingDatabase();
16          pDB->getSymbolTable(pBT,AcDb::kForRead);
17          AcDbBlockTableRecord* pBTR = NULL;
18          pBT->getAt(ACDB_MODEL_SPACE, pBTR, AcDb::kForWrite);
19          pBT->close();
20          // now, add the entity to container
21          AcDbObjectId Id;
22          pBTR->appendAcDbEntity(Id, pL);
23          pBTR->close();
24          pL->close();
25      }
```

On lines **05-07** we instantiate the entity and initialize some local variables. Next, on lines **9-10** we add a list of vertexes to the entity. Line **12** set the polyline as closed. On lines **14-19** we open the AutoCAD database, open the **Block Table** container and then get the **Model Space** container. On lines **21-24** we add our entity do Model Space and then close it and the entity itself.

Note: *You cannot delete the entity's pointer once it is added to AutoCAD database. In this case its management is delegated to AutoCAD and you only need to call the **close()** method. If you delete this pointer you will cause AutoCAD to crash.*

Creating a Custom Entity – Exercise 2 – Step 2

Now after you have created a basic custom entity you will learn how to add custom graphics to it. The custom entity's graphics is generated primarily by the **worldDraw()** method and optionally by the **viewportDraw()** method.

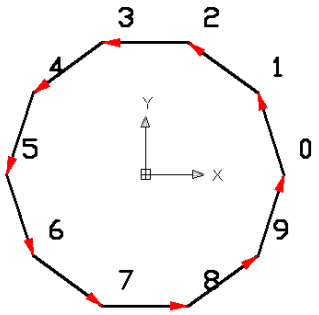


Figure 14 – Custom Entity with vertex numbers and arrows

As this entity's base class already generates a polyline graphic we will add some nice extra graphics to it. Each vertex will receive an **index number and an arrow** indicating the polyline construction direction (Figure 14).

We will use different colors for each graphic type. The index number will use the **256** color code which is the **ByLayer** color and the arrows will use a fixed **1** color code which is **red**. If you change the color of this entity's layer it will change except on the arrows that will stay red. Further, all arrows will be filled.

To create these graphics you will need to use some trigonometric methods. The following code demonstrates what you can reach this:

```
01      Adesk::Boolean AuPolyline::worldDraw (AcGiWorldDraw *mode)
02      {
03        assertReadEnabled();
04        // Call base class first
05        AcDbPolyline::worldDraw(mode);
06        double szRef = 5.0;
07        // ====================================================================
08        // DIRECTION AND VERTEX NUMBERING
09        int signal = 1;
10        double ht2 = szRef/4.0;
11        for(int i=0; i<numVerts(); i++)
12        {
13           AcGePoint3d pti;
14           this->getPointAt(i,pti);
15           // Draw vertex text
16           CString strNum;
17           strNum.Format(_T("%d"),i);
18           AcGePoint3d ptTxt = pti + (AcGeVector3d::kXAxis*ht2) +
       (AcGeVector3d::kYAxis*ht2);
19           mode->subEntityTraits().setColor(256); // ByLayer
20           mode->geometry().text(ptTxt, AcGeVector3d::kZAxis,
       AcGeVector3d::kXAxis, ht2, 1.0, 0.0, strNum);
21           // Arrow direction
22           AcGePoint3d ptj;
23           this->getPointAt(i<(numVerts()-1) ? (i+1) : 0, ptj);
24           AcGeVector3d dir = (ptj - pti).normalize();
25           // Side perpendicular vectors
26           AcGeVector3d perp = dir;
27           perp.rotateBy(3.141592/2.0,AcGeVector3d::kZAxis);
28           AcGePoint3d pt1 = ptj - (dir*ht2) + (perp*(ht2/4.0));
29           AcGePoint3d pt2 = ptj - (dir*ht2) - (perp*(ht2/4.0));
30           AcGePoint3d pts[3];
31           pts[0] = ptj;
32           pts[1] = pt1;
33           pts[2] = pt2;
```

```
34          // Draw arrow polygon
35          mode->subEntityTraits().setFillType(kAcGiFillAlways);
36          mode->subEntityTraits().setColor(1); // red
37          mode->geometry().polygon(3,pts);
38          mode->subEntityTraits().setFillType(kAcGiFillNever);
39       }
40       //------ Returning Adesk::kFalse here will force viewportDraw() call
41       return (Adesk::kTrue) ;
42     }
```

On line **03** we call the proper assert method which inform what type of access we are executing on this method. As we are not changing any data we need only to **READ** the entity so we call **assertReadEnabled()** method. On line **05** we forward up the call to its base class method which will draw the polyline curves. On lines **06-10** we initialize some local variables.

Next, from line **11-39** we perform a loop on each polyline vertex to draw our custom graphics. On lines **13-14** we get the current vertex point. On line range **16-20** we draw the vertex text using the **text()** geometry primitive at a point with a small displacement related to the vertex point. On lines **22-24** we calculate the end point (**ptj**) of each polyline segment and its unitary direction vector (**dir**). Next, on lines **26-33** we calculate the 3 points which will build an arrow head graphic. The **perp** vector will allow us to draw each side of the arrow head according to the Figure 15. On lines **35-38** we draw the filled arrow head using the **polygon()** primitive with **color red** and with the fill type always.

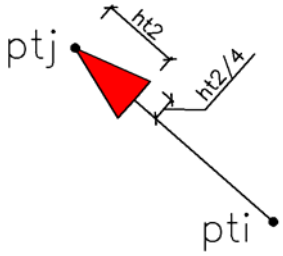


Figure 15 – AuPolyline side graphics.

Finally, on line **41**, we return **Adesk::kTrue** to avoid the graphics to be made by the **viewportDraw()** method.

These classes we have used, with **AcGe** prefix, are part of the AcGe library which contains several utility classes and methods to help us to deal with **geometric calculations**. This really helps a lot once most of these calculations are a little bit complex.

Creating a Custom Entity – Exercise 2 – Step 3

Now you will learn how to add **GRIP** points to your entity. In fact, as the default implementation of **AuPolyline** forward the call to its base class you may already noted that its GRIP points are visible and working. The GRIP point behavior is handled by two methods. The first method, called **getGripPoints()** is responsible for acquiring all GRIP points from your entity. The second, called **moveGripPointsAt()**, is responsible for the action fired by each grip.

Each default polyline GRIP action is to move the related vertex. In this example we want to add **one extra grip** positioned at our **polygon's center**. To do that we first need to create a helper method to calculate this point. We will walk through all polygon points and will sum the coordinates dividing the result by **numVerts()**. This method will not change our entity data so it is recommended to be **CONST** and will require only the **assertReadEnabled()** call:

```
01     AcGePoint3d AuPolyline::GetPolylineCenter() const
02     {
03          assertReadEnabled();
04          AcGePoint3d ptC,pti;
05          double cx = 0.0, cy = 0.0, cz = 0.0;
06          for (int i=0; i<numVerts(); i++)
07          {
08                this->getPointAt(i,pti);
09                cx += pti[X];
10                cy += pti[Y];
11                cz += pti[Z];
12          }
13          cx = cx / numVerts();
14          cy = cy / numVerts();
15          cz = cz / numVerts();
16          ptC.set(cx, cy, cz);
17          return ptC;
18     }
```

Lines **04-15** apply the **center formula** and at line **16** we build the point using cx, cy and cz as the point's X, Y and Z coordinates respectively.

Next we need to change the default implementation of the both GRIP point methods. First we will redefine the **getGripPoints()** method. This method has 2 signatures and we will use the new **AcDbGripData** method instead of the "old style" method.

This method receives an array of **AcDbGripData** pointers. These objects represent the **GRIP** point information. We need to inform at least the point and an arbitrary data (**void***). This arbitrary data can be used later to get back information from each **GRIP** point allowing the **moveGripPointsAt()** method to perform the custom actions.

Further, it will not change our entity's data so it is also a **CONST** method and calls **assertReadEnabled()** method. On lines **09-12** we instantiate an **AcDbGripData** pointer and set its application data (**9999** in this case) and the grip point which is calculated by the **GetPolylineCenter()** method. Next, on line **13** we forward the call to AcDbPolyline's grip point method so it is able to add its own GRIP points at last. These GRIP points are those vertexes points we have mentioned before:

```
01     Acad::ErrorStatus AuPolyline::getGripPoints (
02        AcDbGripDataPtrArray &grips,
03        const double curViewUnitSize,
04        const int gripSize,
05        const AcGeVector3d &curViewDir,
06        const int bitflags) const
07     {
08          assertReadEnabled () ;
09          AcDbGripData* gpd = new AcDbGripData();
10          gpd->setAppData((void*)9999); // Center Grip code
11          gpd->setGripPoint(GetPolylineCenter());
12          grips.append(gpd);
13          AcDbPolyline::getGripPoints (grips, curViewUnitSize, gripSize,
       curViewDir, bitflags);
14          return (Acad::eOk);
15     }
```

The next step is to change the **moveGripPointsAt()** method so when the user clicks on this center GRIP it will reflect a custom action. In this example, our custom action will **move the entire polyline**. The following code shows how to do that:

```
01      Acad::ErrorStatus AuPolyline::moveGripPointsAt (
02           const AcDbVoidPtrArray &gripAppData,
03           const AcGeVector3d &offset,
04           const int bitflags)
05      {
06          assertWriteEnabled () ;
07          for (int g=0; g<gripAppData.length(); g++)
08          {
09            // Get grip data back and see if it is our 0 Grip
10            int i = (int)gripAppData.at(g);
11            // If it is our grip, move the entire entity
12            if (i == 9999)
13                  this->transformBy(offset);
14            else
15                  AcDbCurve::moveGripPointsAt (gripAppData,offset,bitflags);
16          }
17          return (Acad::eOk);
18      }
```

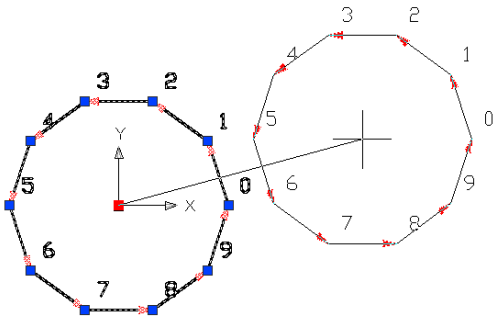


Figure 16 – Center GRIP in action.

This time, our method cannot be **CONST** once we will change our entity's data. Due that, we need to call **assertWriteEnabled()** at its beginning. On line **07** we start to inspect the **AcDbVoidPtrArray** (an array of **void***) looking for our application data (**9999**). This method also receives a **3D vector** which represents the transformation being applied to the GRIP. If the GRIP being modified is our **9999** we will apply this vector transformation to the entire polyline. This can be done through the **transformBy()** method passing the same vector (Figure 16).

If the fired **GRIP** is not our **9999** code we will forward the call to **AcDbCurve** class (AcDbPolylin's base class) which will handle it for us. The resulting behavior is when you select a GRIP over the polyline boundary the entity stretches and when you select the center GRIP the entity moves.

Creating a Custom Entity – Exercise 2 – Step 4

Now you will learn how to add **OSNAP** points to your custom entity. If you run the application before this step you will see that our custom entity already shows some Object Snap points by default (**ENDPOINT** and **MIDPOINT**). In this example we want to add a **CENTER** object snap to our polyline that will allow users to select the exact **center point**. To provide this, we need to change the standard behavior of **getOsnapPoints()** method. This method has **4 signatures** and 2 of them, containing the **AcGeFastTransform** parameter, are for future use. We will use the first signature and will handle the **CENTER** object snap responding with our center point:

```
01     Acad::ErrorStatus AuPolyline::getOsnapPoints (
02          AcDb::OsnapMode osnapMode,
03          int gsSelectionMark,
04          const AcGePoint3d &pickPoint,
05          const AcGePoint3d &lastPoint,
06          const AcGeMatrix3d &viewXform,
07          AcGePoint3dArray &snapPoints,
08          AcDbIntArray &geomIds) const
09     {
10          assertReadEnabled () ;
11          switch(osnapMode)
12          {
13                case AcDb::kOsModeCen:
14                      snapPoints.append(GetPolylineCenter());
15                break;
16          }
17          return (AcDbPolyline::getOsnapPoints (osnapMode, gsSelectionMark,
       pickPoint, lastPoint, viewXform, snapPoints, geomIds)) ;
18     }
```

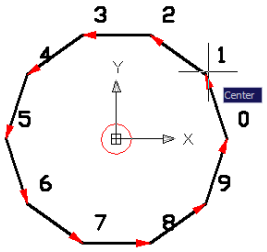


Figure 17 – CENTER object snap.

This method receives an Array of points where we need to add the center point whenever the **OSNAP CENTER** is requested. On line **13** we handle the **CENTER** object snap by adding our **center point** to the **snapPoints** array (Figure 17). Even handling this particular object snap this method needs to call the polyline class level. The polyline will handle all other possible object snaps for us.

Note: *This OSNAP solution handles only simple OSNAP types like CENTER, MIDPOINT and ENDPOINT. Complex OSNAP modes like INTERSECTION require other methods implementations and some additional procedures.*

Creating a Custom Entity – Exercise 2 – Step 5

Sometimes you need to distribute an AutoCAD drawing with custom entities inside. By default, when AutoCAD opens a drawing and find some entity it does not recognize, it protects this entity and **packs its binary data into a Proxy entity**. The proxy entity **protects your object data** avoiding unwanted users to manipulate your custom entities.

The proxy entity is merely a dummy entity with a **fixed graphical representation**. Several features of your custom entity will not be available once your code is not there to provide these methods. If the drawing is opened by an unadvised user you should inform it about the missing application.

The proxy can contain specific graphics which will be generated with a call to your **worldDraw()** method just before AutoCAD close the drawing. The worldDraw() method has a parameter, an **AcGiWorldDraw** pointer, that allows you to call a **regenType()** method to find out if the caller is requesting proxy graphics to your entity. At this time, you can draw a different graphic to make and advertisement of your missing custom entity. The following code shows how to handle the proxy graphics:

```
01      // ===========================================================================
02      // PROXY
03      if (mode->regenType() == kAcGiSaveWorldDrawForProxy)
04      {
05          // Draw dummy text
06          CString strTxt = _T("AU Polyline");
07          AcGePoint3d ptTxt = GetPolylineCenter();
08          mode->geometry().text(ptTxt, AcGeVector3d::kZAxis,
        AcGeVector3d::kXAxis, szRef, 1.0, 0.0, strTxt);
09      }
```

On this example the proxy graphics will be the standard entity graphic plus a text indicating our class name. You may also add an URL address of your product or company. We will use the center point as this text's start point.

Unfortunately this solution is not complete for AcDbPolyline derived entities. There is a problem when a polyline needs to generate its proxy graphics and the worldDraw() method is not called. To solve this problem we need to add another method, called **saveAs()**, to our class. This method has the following declaration:

```
virtual void saveAs(AcGiWorldDraw * mode, AcDb::SaveType st);
```

The implementation of this method is as follows:

```
01      void AuPolyline::saveAs(AcGiWorldDraw * mode, AcDb::SaveType st)
02      {
03          AcDbPolyline::saveAs (mode, st) ;
04          if ((mode->regenType() == kAcGiSaveWorldDrawForProxy) &&
05          (st == AcDb::kR13Save))
06              this->worldDraw(mode);
07      }
```

This method first forward the call to our base class and then test if the **regenType()** and the save type is **AcDb::kR13Save**. If they are, we forward the call to our own **worldDraw()** method which will draw the custom entity graphic plus the proxy text message.

To test this behavior create some **AuPolyline** entities, save the drawing, close AutoCAD and then open this DWG without loading the application modules. Once the DWG file is opened you will see a proxy warning dialog with some information about the missing application (Figure 18).

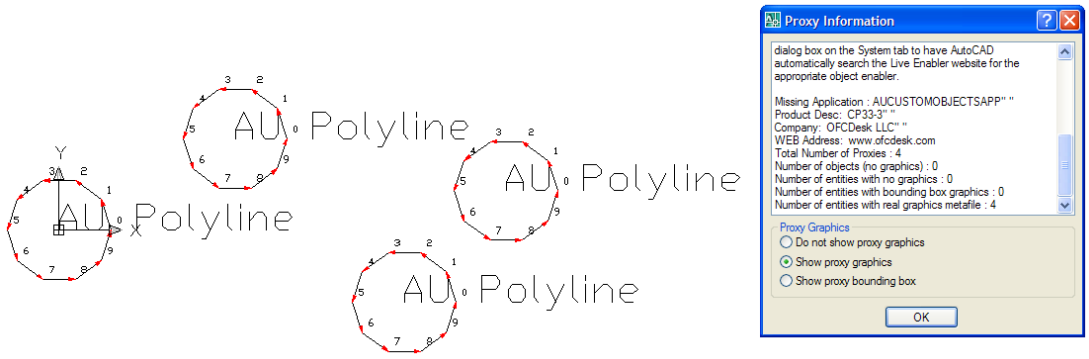


Figure 18 – Proxy Information dialog.

Note on the Figure 18 that our proxy text is displayed and also, at the proxy information dialog, you may find information about the missing application. This dialog allows users to choose one of the 3 options about how AutoCAD will handle its proxy entities. You can also setup the standard proxy behavior accessing **Tools** > **Options** > **Open and Save**.

Remember that the **DBX** module can be loaded to re-enable the custom entity or you can also send your DBX module along with the **DWG** file to enable third-party users to see your custom entity with complete information. This way you can provide a full graphical representation without the application interfaces (**ARX** module). The third-party users will be able to see it but can't modify the custom entity.

AutoCAD allows some basic operations over the proxy entity like erase, change layer, change color and transformations. These operations will act basically over its dummy graphical representation (they will not affect your custom entity's data except when it is erased). It is up to the developer to determine the most adequate flags for each custom entity. This is a compiler-time option and it is defined on the custom entity's class implementation macro (**ACRX_DXF_DEFINE_MEMBERS**). This flags can be combined to build a complete configuration. Please refer to ObjectARX SDK documentation for further information about proxy flags.

Creating a Custom Entity – Exercise 2 – Step 6

On this step we will implement a pretty nice feature. Imagine you would like to add a **hatch filling** to your custom entity. We can take advantage of ObjectARX **embedded object** feature to implement this. There is a class called **AcDbHatch** which represents the AutoCAD hatch entity. This class can be used as an embedded object and we can use its **worldDraw()** method to draw our own hatch pattern. The first thing you need to do is to add an **AcDbHatch** member to our custom entity's class. We will also declare the **SetupHatch()** method to setup the hatch properties. To do that, open the **AuPolyline.h** file, and place the following lines at the end of class declaration:

```
protected:
      AcDbHatch m_Hatch;

public:
      void SetupHatch();
```

Further, we will need to add 3 more methods to our entity to handle modifications. The first method will handle all **graphic transformations**. The 2 remaining methods will handle the **STRETCH** command:

```
public:
      virtual Acad::ErrorStatus transformBy(const AcGeMatrix3d & xform);
      virtual Acad::ErrorStatus getStretchPoints(
            AcGePoint3dArray & stretchPoints) const;
      virtual Acad::ErrorStatus moveStretchPointsAt(
            const AcDbIntArray & indices, const AcGeVector3d & offset);
```

Our hatch object needs to be configured. To do this we will place, inside the custom entity's constructor located at **AuPolyline.cpp** file, the following code (note that this configuration needs to be done only once so the constructor is the better place to put it):

```
01      AuPolyline::AuPolyline () : AcDbPolyline ()
02      {
03           m_Hatch.setNormal(AcGeVector3d::kZAxis);
04           m_Hatch.setElevation(this->elevation());
05           m_Hatch.setAssociative(true);
06           m_Hatch.setPatternScale(1.0);
07           m_Hatch.setPatternAngle(45.0);
08           m_Hatch.setHatchStyle(AcDbHatch::kNormal);
09           m_Hatch.setPattern(AcDbHatch::kPreDefined,_T("LINE"));
10      }
```

This configuration will set the hatch **pattern**, **normal** vector, **elevation**, **scale**, **angle** and **style**. In this example they are fixed but you may want to create one property for each of these parameters allowing the user to change them at runtime.

Now we need to add the **SetupHatch()** method implementation to build the hatch loop according to our polyline boundary. The code will be as follows:

```
01      void AuPolyline::SetupHatch()
02      {
03           assertWriteEnabled();
04           // Remove previous loop
05           for (int l=0; l<m_Hatch.numLoops(); l++)
06                m_Hatch.removeLoopAt(l);
07           // Insert the updated loop
08           AcGePoint2dArray vertexPts;
09           AcGeDoubleArray vertexBulges;
10           // Collect points and bulges
11           for(int i=0; i<numVerts(); i++) {
12                AcGePoint2d pt2d;
13                double bulge = 0.0;
14                this->getPointAt(i,pt2d);
15                this->getBulgeAt(i,bulge);
16                vertexPts.append(pt2d);
17                vertexBulges.append(bulge);
18           }
19           // Close the loop
20           vertexPts.append(vertexPts.first());
21           vertexBulges.append(vertexBulges.first());
22           m_Hatch.appendLoop(AcDbHatch::kDefault, vertexPts,vertexBulges);
23           // Refresh hatch
24           m_Hatch.evaluateHatch();
25      }
```

On lines **05-06** we make sure there is no previous loop inside hatch. At the line range **11-18** we walk through the polyline vertexes and collect its points and bulges (*the bulge is the tangent of 1/4 of the included angle for the arc between the selected vertex and the next vertex*). The collected information will be stored at two dynamic vectors: **AcGePoint3dArray** and **AcGeDoubleArray**. On lines **20-21** we close the polyline loop to ensure our hatch boundary is closed.

On line **22** we append the arrays to the hatch entity as one loop. The loop can be also a hole into the hatch surface but in this example our loop is **AcDbHatch::kDefault**. On line **24** we finish the hatch configuration process by calling the **evaluateHatch()** method which will generate the hatch itself.

We need to call the **SetupHatch()** method inside some of our methods. The first place is inside the **dwgInFields()**. Place a call to this method at the end of this method as follows:

```
01      Acad::ErrorStatus AuPolyline::dwgInFields (AcDbDwgFiler *pFiler)
02      {
03            [ some lines were not displayed for code brevity ]
04            // Setup hatch
05            SetupHatch();
06            return (pFiler->filerStatus ()) ;
07      }
```

Next, we need to place another call inside **moveGripPointsAt()** method. When user moves some of the GRIP points we need to recalculate the hatch boundary. We need to do this only in cases the selected GRIP is not our center point. The change is made on lines **15-18** as follows:

```
01      Acad::ErrorStatus AuPolyline::moveGripPointsAt (
02            const AcDbVoidPtrArray &gripAppData,
03            const AcGeVector3d &offset, const int bitflags)
04      {
05          assertWriteEnabled () ;
06          for (int g=0; g<gripAppData.length(); g++)
07          {
08            // Get grip data back and see if it is our 0 Grip
09            int i = (int)gripAppData.at(g);
10            // If it is our grip, move the entire entity.
11            if (i == 9999)
12                  this->transformBy(offset);
13            else
14            {
15                  AcDbCurve::moveGripPointsAt(gripAppData,offset,bitflags);
16                  SetupHatch();
17            }
18          }
19          return (Acad::eOk);
20      }
```

To make the hatch entity appear as part of our custom entity's graphics we need to call its **worldDraw()** method from inside our entity's **worldDraw()**:

```
01      Adesk::Boolean AuPolyline::worldDraw (AcGiWorldDraw *mode)
02      {
03            [ some lines were not displayed for code brevity ]
04            // ========================================================
05            // HATCH
06            m_Hatch.worldDraw(mode);
07            //------ Returning Adesk::kFalse to force viewportDraw() call
08            return (Adesk::kTrue) ;
09      }
```

Finally, we need to implement the code for the 3 new methods we have added to our custom entity's class. Open the **AuPolyline.cpp** file and add the following methods:

```
01      // -----------------------------------------------------------------------
02      Acad::ErrorStatus AuPolyline::transformBy(const AcGeMatrix3d & xform)
03      {
04           Acad::ErrorStatus retCode = AcDbPolyline::transformBy(xform);
05           m_Hatch.transformBy(xform);
06           return (retCode) ;
07      }
08      // -----------------------------------------------------------------------
09      Acad::ErrorStatus AuPolyline::getStretchPoints(AcGePoint3dArray &
        stretchPoints) const
10      {
11           AcDbIntArray osnapModes,geomIds;
12           return this->getGripPoints(stretchPoints,osnapModes,geomIds);
13      }
14      // -----------------------------------------------------------------------
15      Acad::ErrorStatus AuPolyline::moveStretchPointsAt(
16         const AcDbIntArray & indices,
17         const AcGeVector3d & offset)
18      {
19           Acad::ErrorStatus ret = AcDbPolyline::moveGripPointsAt (indices,
        offset);
20           SetupHatch();
21           return ret;
22      }
```

The first method on lines **02-07**, **transformBy()**, is responsible for all entity's graphic transformations such as **MOVE**, **ROTATE**, **SCALE**, etc. First we forward the call to our base class and then apply the same transformation to the hatch. This way it will follow all transformations applied to our AuPolyline.

The second method on lines **09-13**, **getStretchPoints()**, is responsible to return the points that are enabled to stretch the entity. In this case we would like to add all of our polyline vertexes. This method can reuse the **getGripPoints()** method which returns the same points we want.

The last method on lines **15-22**, **moveStretchPointsAt()**, is responsible to apply the stretch transformation over the entity. We will also reuse the existing method **moveGripPointsAt()** because it does exactly what we need. Next we just need to call **SetupHatch()** again to ensure our hatch is updated with the new boundary resulting from the STRETCH command.

Before test our custom entity, we need to place a call to **SetupHatch()** just before to close the entity on its creation stage. Open the **acrxEntryPoint.cpp** file, of **AuUserInterface** project, and locate the **AuUserInterface_MyCommand1()** method. See the code below:

```
01      static void AuUserInterface_MyCommand1(void)
02      {
03           [ some lines were not displayed for code brevity ]
04           pL->SetupHatch();
05           pL->close();
06      }
```

Now **Build you Solution** again. You should get no errors. Open AutoCAD, load your modules (remember, first **BDX** and then **ARX**). Fire **MYCOMMAND1** command and create one AuPolyline entity. Next test the several features we have implemented. Try to **COPY** you entity, **MOVE** it, **ROTATE** it, **SCALE** it and apply a **MIRROR**. You can also use the **STRETCH** command and move the **GRIP** points to change the entity's shape.

Our **AuPolyline** entity is derived from **AcDbPolyline**, right? So do you expect that a specific polyline commands like **PEDIT** work with our entity? **Yes, it works!** Try to fire the **PEDIT** command and select our polyline. It will accept it and will allow you to change the AuPolyline as if it is a native AutoCAD

Polyline. You can add new vertexes, remove existing, join new segments and even open the polyline (Figure 19). Great!

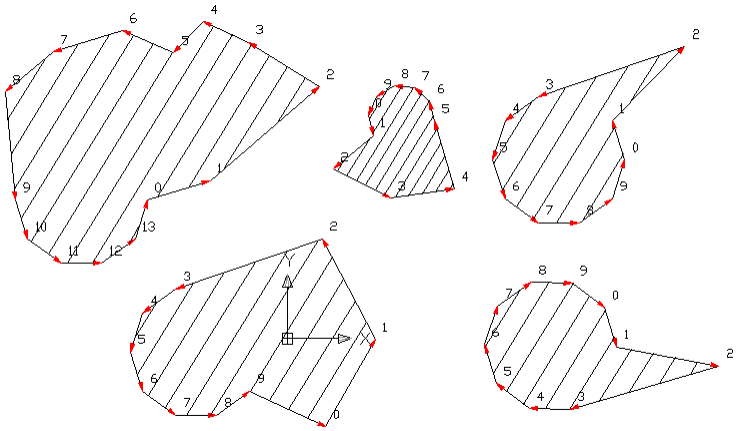


Figure 19 – AuPolyline modifications.

Conclusion

In this session you have learned how to create a custom entity and add some of the possible features **ObjectARX** allows. ***This is really only the tip of the iceberg***. There are much more you can do using **ObjectARX** classes and implementing more sophisticated features. I really hope you have enjoyed this session and hope it may help you to make the startup into the **ObjectARX** world.

All of its session complete source code will be made available through **AU-Online** website at http://www.autodesk.com/auonline right after this session. You may also contact me by e-mail to solve any issue related to this course or related to ObjectARX. Further I would like to invite you to visit my **Blog** site at http://arxdummies.blogspot.com/ to find out more information.

Thank you!