ObjectARX&Dummies 教程（一）——Overview

一个巴西人写的 ObjectARX 教程，其后面所讲到的自定义实体的教程，应该是这方面最好的教程了。

呵呵，没时间翻译，大家就将就着看英文吧。

Class 1 - Overview

Introduction

ObjectARX is an AutoCAD Runtime Extension.

With ObjectARX SDK you can build applications that will alow you to extend AutoCAD features like commands, dialog boxes, entities, objects and much more.

The ObjectARX application is actually a DLL that is loaded into AutoCAD environment and allows you to access new features as mentioned above. To be able to build these DLLs you need to follow some basic rules to setup Microsoft Visual Studio.NET environment and assert your application will respect AutoCAD requirements.

The performance of this application will be the same of native features. By the way, Autodesk uses ObjectARX to build vertical products you may already know like Autodesk MAP and Architectural Desktop, among many others.

User Requirements

Because ObjectARX is not a simple customization tool, some requirements must be observed to allow you to be able to proceed. If you don't match these requirements I would recommend you to first increase your skills and then go back to try this course.

The minimum requirements to learn at least basic ObjectARX are:

（1）Basic Visual Studio.NET concepts;

（2）Average C++ knowledge;

（3）Advanced AutoCAD knowledge;

（4）MFC (Microsoft Foundation Classes) concepts;

（5）Object Oriented Techniques;

As mentioned before, this course is not intended to be a complex guide or even cover advanced features. I believe you can go further by yourself after learn the basic features and mainly the basic concepts which are the secret of ObjectARX.

I will not cover .NET Framework capabilities to make the course as much simple as I can. Once you learn and figure out how AutoCAD works from inside you will open your mind to ObjectARX capabilities and will be able to build great application!

How to use ObjectARX SDK

Once you have downloaded your copy of ObjectARX SDK and extracted it to your hard drive, you will find the following folders inside it:

\arxlabs : This directory consists of a set of labs that shows some aspects of the ObjectARX.

\classmap : This directory contains an AutoCAD drawing with the ObjectARX class hierarchy tree.

\docs : This directory contains the ObjectARX online help files.

\inc : The inc directory contains the ObjectARX header files.

\lib : The lib directory contains the ObjectARX library files.

\redistrib : This directory contains DLLs that may be required for an ObjectARX application to run.

\samples : This directory contains examples of ObjectARX applications.

\utils : This directory contains other libraries like brep for boundary representation and ObjARXWiz for the ObjectARX wizards.

ObjectARX Classes Naming

ObjectARX classes names follow the following prefix standards:

AcRx : Classes for binding an application and for runtime class registration and identification.

AcEd : Classes for registering native AutoCAD commands and for AutoCAD event notification.

AcDb : AutoCAD database classes.

AcGi : Graphics classes for rendering AutoCAD entities.

AcGe : Utility classes for common linear algebra and geometric objects.

Depending on which set of features you use in your applications you will need to use the corresponding library as follows:

AcRx : acad.lib, rxapi.lib, acdb16.lib

AcEd : acad.lib, rxapi.lib, acedapi.lib, acdb16.lib

AcDb : acad.lib, rxapi.lib, acdb16.lib

AcGi : acad.lib, rxapi.lib, acdb16.lib

AcGe : acad.lib, rxapi.lib, acge16.lib, acdb16.lib

ObjectARX Wizard

I will bypass the Visual Studio environment configuration to build ObjectARX applications. You could refer to this information inside SDK documentation. We will use on this course the Wizard provided by ADN (Autodesk Developer Network) team. It is located inside ObjectARX directory called \utils\ObjARXWiz. Inside it you will find the installation package named ArxWizards.msi.

To install this Wizard, close your Visual Studio.NET and double click the above mentioned file. Follow the steps. When finished, open Visual Studio.NET again and you will see a new toolbar.

See you on Class 2!

ObjectARX&Dummies 教程（二）——AutoCAD's Database

Class 2 - AutoCAD's Database

Introduction

Each AutoCAD drawing represents a structured Database which stores several types of objects. When you just open a new drawing, AutoCAD creates behind the scenes a organized and efficient Database. This Database has a minimun data that allows you to make basic drawings.

This minimun data is represented basically by objects like layers, linetypes, text styles, etc. Due that you have layer 0, stadard text style, continuous linetype and others.

Since AutoCAD Release 2000 you can work with multiple drawings at the same time which is called MDI environment. This functionality brings great flexbility but also bring some extra complexity when dealing with more than one drawing. We won't discuss MDI aspects on this course but this will be probably a requirement for your future ObjectARX applications.

How data is stored

This Database mantains all sort of objects a drawing needs to exist. These objects are stored into appropriate containers which are special objects made to manage objects of the same type. This way we have appropriate methods and procedures to store entities, layers, text styles, etc. Each object stored into Database receives an identifier that is called ObjectId. This identifier is unique inside the same AutoCAD session and it is valid during the lifecycle of each object. The ObjectId is generated by its Database and you don't need to worry about how to create it.

- Inside ObjectARX we have basically 3 kind of objects:
- Entities: Objects with graphical representation (lines, arcs, texts, ...);
- Containers: Special objects to store and manage collections (layer table, linetype table, ...);
- Objects: Objects without any graphical representation (groups, layouts, ...)

AutoCAD's Database structure

Creating objects

To create an object through ObjectARX we have some kind of recipe depending on what type of object it is and where we would like to store it (most of time we need to store an objects inside its specific container). Basically, you will follow this sequence:

Declare a pointer to the object type you would like to create and call its new operator;

With this pointer, call appropriate methods of this object to change its features;

Get a pointer to the Database where you would like to create the object (most of time the current Database);

Open the appropriate container where it should be stored;

Call the specific container method to store your object passing its pointer;

Receive its ObjectId automatically generated by its container;

Finish the operation closing all opened objects including containers and the object you have just created.

Obviously you will create some handy classes to allow the automation of this processes because they are very similar and can be easily reused. The main idea is to create a sort of database utility funcions like: AddLayer, AddLine, AddCircle, AddTextStyle, etc.

* It is very important to not forget to close opened objects because this will cause AutoCAD to terminate.

Sample code to create a line (AcDbLine)

This code demonstrates how to create a simple line between two points. The process is simple and no error check is made. This code needs to be embedded inside an ObjectARX application structure to work. The main idea is to show you the concepts. Further we will create a working code. Pay attention to the order of opening and closing operations.

```
// We first need to declare a couple of points
```

```
    AcGePoint3d startPt(1.0, 1.0, 0.0);
    AcGePoint3d endPt(10.0, 10.0, 0.0);
    // Now we need to instantiate an AcDbLine pointer
    // In this case, its constructor allows me to pass the 2 points
    AcDbLine *pLine = new AcDbLine(startPt, endPt);
    // Now we need to open the appropriate container which is inside
BlockTable
    AcDbBlockTable *pBlockTable = NULL;
    // First, get the current database and then get the BlockTable
    AcDbDatabase* pDB = acdbHostApplicationServices()-
>workingDatabase();
    pDB->getSymbolTable(pBlockTable, AcDb::kForRead);
    // Inside BlockTable, open the ModelSpace
    AcDbBlockTableRecord* pBlockTableRecord = NULL;
    pBlockTable->getAt(ACDB_MODEL_SPACE, pBlockTableRecord,
AcDb::kForWrite);
    // After get ModelSpace we can close the BlockTable
    pBlockTable->close();
    // Using ModelSpace pointer we can add our brand new line
    AcDbObjectId lineId = AcDbObjectId::kNull;
    pBlockTableRecord->appendAcDbEntity(lineId, pLine);
```

```
// To finish the process we need to close ModelSpace and the entity
pBlockTableRecord->close();
pLine->close();
```

On the next class we will present the ObjectARX application structure and will build and compile a simple application using the above code. See you there!

ObjectARX&Dummies 教程（三）——Application Overview

Class 3 - Application Overview

Introduction

As I mentioned before, the ObjectARX is actually a DLL. It can be linked with or without MFC extensions. Most of times you would like to link with MFC. Autodesk provides a pretty handy Wizard to allow users to quickly create ObjectARX applications with minimun code effort.

Before we can proceed I would like to clarify the main differences between ObjectARX and ObjectDBX applications. The main idea is to separate Interface and Object Classes to allow the called "Object Enablers". This is not a mandatory but it is a good programming practice and Autodesk made this separation to provide great things like, for instance, open an ADT drawing inside AutoCAD, download the proper enabler and then show those ADT entities correctly inside AutoCAD.

Suppose you need to create an application with a bunch of custom entities and objects. The drawings created with this application will contains these custom objects and if another users try to open this drawing inside AutoCAD and without your applications they will see only Proxy entities. If you would like to allow these users to see your custom entities but don't perform any commands over them you just need to ship de DBX part of your application. This way the user will be able to see your custom entities and perform some limited operations.

When the drawing is saved AutoCAD preserve the custom entity information using the called Proxy entities. This will happen even if the DBX module is not available. The Proxy entities stores the binary data of your custom object and keep that until your application come back.

In other hand, the ARX module of your application will be responsible for the interface. There you should register your commands, create your dialogs, customize menus, etc.

Application Structure

Both ARX and DBX modules must implement an entry point function. This function is responsible to perform messages exchange between AutoCAD and your application. For those who are familiar with old C language it is the substitute of main() function.

This entry point function has a signature as follow:

```
extern "C" AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt);
```

A simple implementation of this function is:

```
extern "C" AcRx::AppRetCode
acrxEntryPoint(AcRx::AppMsgCode msg, void* pkt)
{
    switch(msg) {
        case AcRx::kInitAppMsg:
            break;
        case AcRx::kUnloadAppMsg:
            break;
        default:
            break;
}
    return AcRx::kRetOK;
}
```

This function is automatically implemented by the Wizard as you will see later. The first parameter (msg) is the message sent from AutoCAD to your application telling you what is happening. You may receive a "new drawing" message, a "init application" message among many others. These messages are very important to your application and to allow you to react to each desired event to monitor. The second parameter (pkt) is a data package that can be useful in some situations that I would avoid to discuss now (remeber, our course is for Dummies). This function must return a value to AutoCAD using AppRetCode which can be kRetOK (common value) or even kRetError which will force AutoCAD to unload your application.

The most relevant point here is to remember that this function is very important and is where your application will begin to execute.

Registering Commands

Probably your application will implement several commands. You can register your commands from the acrxEntryPoint() function when receiving the kInitAppMsg message that represents the event fired by AutoCAD when it loads your application (this can be done by several ways that we will discuss later). Once this message is received, you can call the appropriate methods to register each desired command.

Registered commands must have a Group Name, a Global Name, a Local Name, some flags, a void function pointer and, optionally, some other parameters. Basically the registered command will fire the function you specified. This function must be a void function without any parameters. When the user inside AutoCAD call your command, AutoCAD seek its command stack, find your command and fire your function. That's it!

It is very important that you preceed your commands with a 3 or 4 letters prefix to avoid command colision with other third-party applications. Regarding to the main parameters:

Group Name: Allows you to group your common commands to make easier to unload and manage them;

Global Name: Your command untranslated name. You should use english names here that is the most used language;

Local Name: Your localized command name which represents the translated name;

Flags: Can be a combination of several types. The two most important flags are ACRX_CMD_TRANSPARENT and ACRX_CMD_MODAL. They establishes your command behavior that can be transparent (like ZOOM) or modal (like major commands);

void function pointer: Here you pass the name of the void function you would like to link with the command. This function will be fired by AutoCAD when the command is invoked.

Once you register your commands you need obviously to unregister when leaving AutoCAD or when your application is unloaded. This can be easily done unregistering all commands by the Group Name. When you recieve the kUnloadAppMsg message is time to remove your commands.

Running your application

Supposing you already compiled your application successfully it is time to load it inside AutoCAD and test your commands. My preferable method to load / unload ObjectARX applications is through the APPLOAD command. It opens a very handy dialog that allows you to browse for the application and load it. It comes also with a Startup Suite briefcase that allows you to automatically load a list of applications when

AutoCAD starts.

Once your application is loaded, just fire your commands and enjoy!

I have posted a simple application made by ARXWizard and Visual Studio.NET 2002 (7.0) and ObjectARX 2004. It is called SimpleLine and it was posted to our file sharing web site as mentioned before. I would like you to download it and pay attention to the code that we have discussed on Class 2. See that I have mapped a command to a function and inside this function the routing to create a line is performed. Go ahead, give a try! Compile this code and open the result ARX file inside AutoCAD.

Next class I will show how to build this application from scratch using the ARXWizard.

Stay tuned!

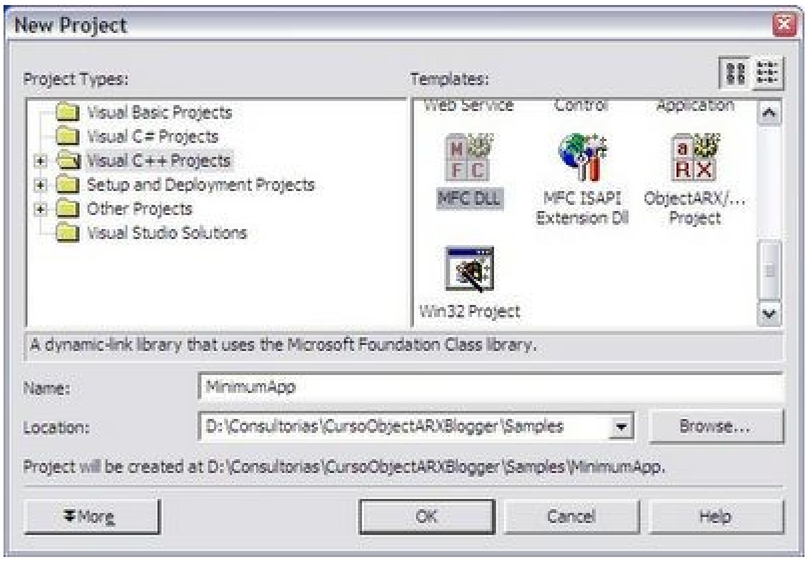
ObjectARX&Dummies 教程（三 a）——Minimum application

Class 3a - Minimum application

Hello

On this class we will implement the minimum ObjectARX application without use the ARXWizard. To do that we will need to create the Visual C++ project from scratch and perform some tuning on project settings.

To begin, open your Microsoft Visual C++ .NET 2002. Open File menu, then choose New and New Project. The following dialog will appear (note that this dialog can change a little bit depending on what add-on your have installed):



A dynamic-link library that uses the Microsoft Foundation Class library.

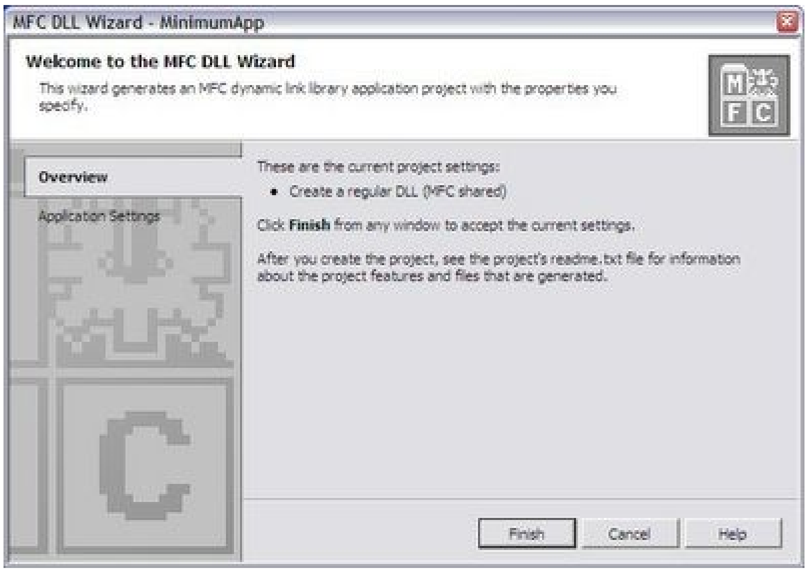
Name: MinimumApp

Location: D:\Consultorias\CursoObjectARXBlogger\Samples

Project will be created at D:\Consultorias\CursoObjectARXBlogger\Samples\MinimumApp.

On this dialog, choose Visual C++ Projects tree node and, on the right portion select MFC DLL template. After that, specify the name and location you would like to use for this project. Click OK to continue.

After click OK the following dialog will be displayed. This dialog has two steps (in this case). The first step, called Overview, just confirm what you have entered before.



Welcome to the MFC DLL Wizard

This wizard generates an MFC dynamic link library application project with the properties you specify.

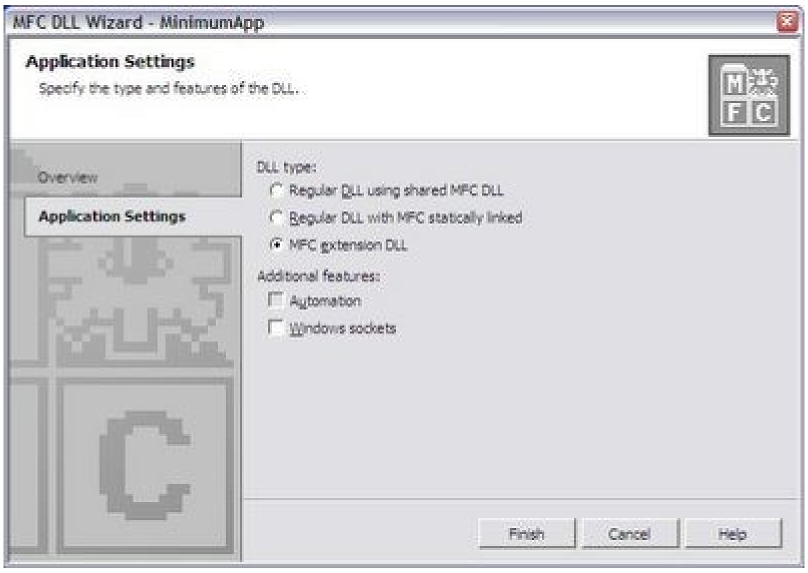
These are the current project settings:

- Create a regular DLL (MFC shared)

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

Clicking on the Application Settings step, the following page will appear inside this dialog:



Application Settings

Specify the type and features of the DLL.

DLL type:

- ○ Regular DLL using shared MFC DLL
- ○ Regular DLL with MFC statically linked
- ● MFC extension DLL

Additional features:

- ☐ Automation
- ☐ Windows sockets

Now we will choose the MFC extension DLL that is the most adequate DLL type to build ObjectARX applications. Note that the use of MFC is not an obligation. You can build ObjectARX applications without MFC but I strongly recommend you to always use MFC because if you don't need MFC now you probably will need it in a near future.

I would like to avoid more technical discussions on this subject because this is not our focus on this course. MFC is a huge and rich library that will avoid several lines of code and will make your application safe and easy to manage. Click Finish to continue.

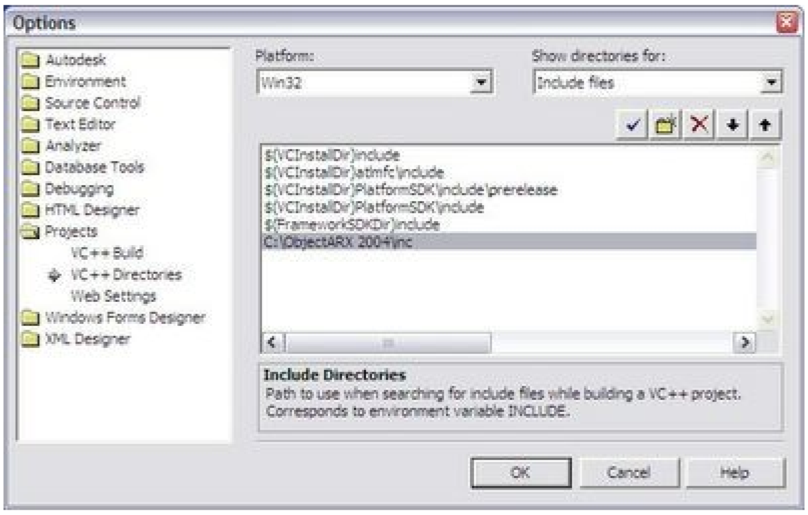
After clicking Finish Visual C++ will create the MFC DLL project files for you with basic implementation of some features. Remember, this is just a minimum application and we will only make a few things to turn it ready to compile, build and load into AutoCAD.

The Visual C++ environment is very intuitive and I guess you will not face much trouble to learn how to use it. Basically it has a project management area (default placed at left side), an editor area (placed at the right portion) and the command / monitor area which is placed below. The project management area uses a tab dialog bar to provide tools like Solution Explorer, Resource View and Class View among many others.

Select the Solution Explorer tab and you will see your project files, organized using a tree. This explorer can handle multiple projects but only one can be the default which has its name in bold font.

Now we will need to change some basic settings on our Visual Studio environment to allow us to compile and link ObjectARX application. As I said on previous classes, to build the application, which is a DLL, we will need to compile using the provided ObjectARX headers (.H files) and to link with ObjectARX libraries (.LIB files). The easiest way to do this is to change the global Options of Visual Studio. This will affect all projects and you won't need to do this again for new projects.

Open the Tools menu and select Options (the last entry). The following dialog will appear. Select Projects and then VC++ Directories. On the Show directories for field, select Include files. Below will appear a list of the include directories that Visual Studio already has and we will add our ObjectARX inc path which contains the desired .H files. Click on the folder icon and click on the ellipsis button and search for the inc path (in my case, it is placed at C:\ObjectARX 2004\inc).



Platform: Win32

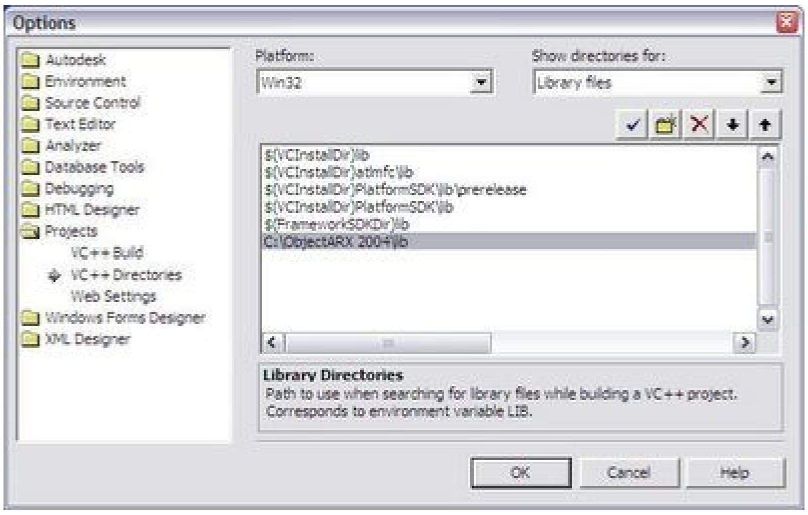
Show directories for: Include files

```
$(VCInstallDir)include
$(VCInstallDir)atlmfc\include
$(VCInstallDir)PlatformSDK\include\prerelease
$(VCInstallDir)PlatformSDK\include
$(FrameworkSDKDir)include
C:\ObjectARX 2004\inc
```

Include Directories
Path to use when searching for include files while building a VC++ project. Corresponds to environment variable INCLUDE.

Don't click OK. Now we need to add the library files directory. Select

Library files on the Show directories for field. Repeat the above procedure to add a path but this time you will add the lib path (in my case, C:\ObjectARX 2004\lib). Click OK to finish.



Platform: Win32

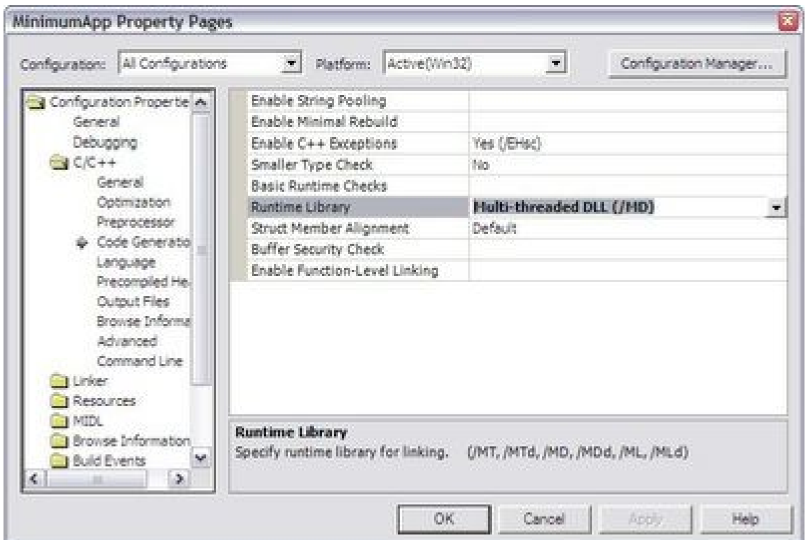
Show directories for: Library files

```
$(VCInstallDir)lib
$(VCInstallDir)atlmfc\lib
$(VCInstallDir)PlatformSDK\lib\prerelease
$(VCInstallDir)PlatformSDK\lib
$(FrameworkSDKDir)lib
C:\ObjectARX 2004\lib
```

Library Directories
Path to use when searching for library files while building a VC++ project.
Corresponds to environment variable LIB.

Now we need to configure our project. This will require some project settings change and some code typing. First, we will change the project settings. Right click the project name inside Solution Explorer and select Properties. The following dialog will appear. Select, on the Configuration field, All Configurations. This will allow us to change both Debug* and Release* settings at the same time.

On the Configuration Properties node, select C/C++ and Code Generation. The right portion of this dialog will display a list with several properties. Select the Runtime Library entry and chance its value to Multi-threaded DLL. This is a requirement to make our DLL compatible with AutoCAD environment.

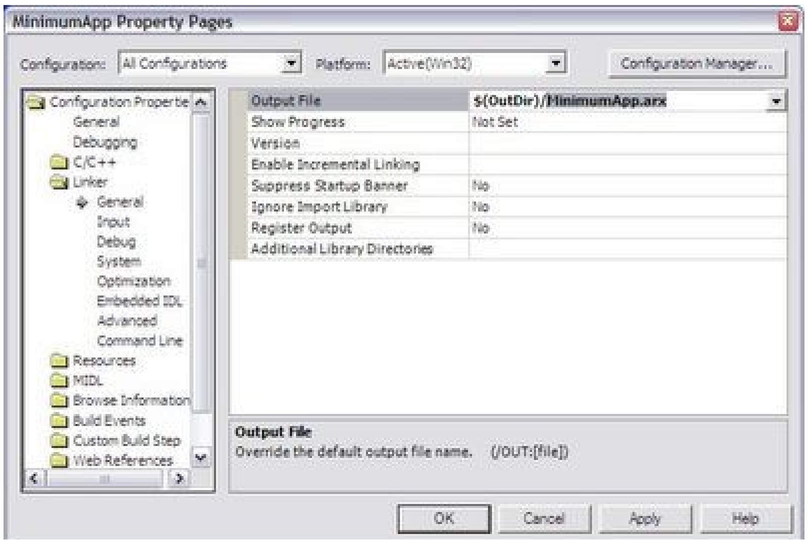

MinimumApp Property Pages

Configuration: All Configurations Platform: Active(Win32)

Enable String Pooling	
Enable Minimal Rebuild	
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	
Runtime Library	Multi-threaded DLL (/MD)
Struct Member Alignment	Default
Buffer Security Check	
Enable Function-Level Linking	

Runtime Library
Specify runtime library for linking. (/MT, /MTd, /MD, /MDd, /ML, /MLd)

Now, select the Linker node and General. On the Output File entry change the extension name from DLL to ARX. Note that Visual Studio use several macros (those names with a $ at beginning) to allow easy and flexible path configuration.


MinimumApp Property Pages

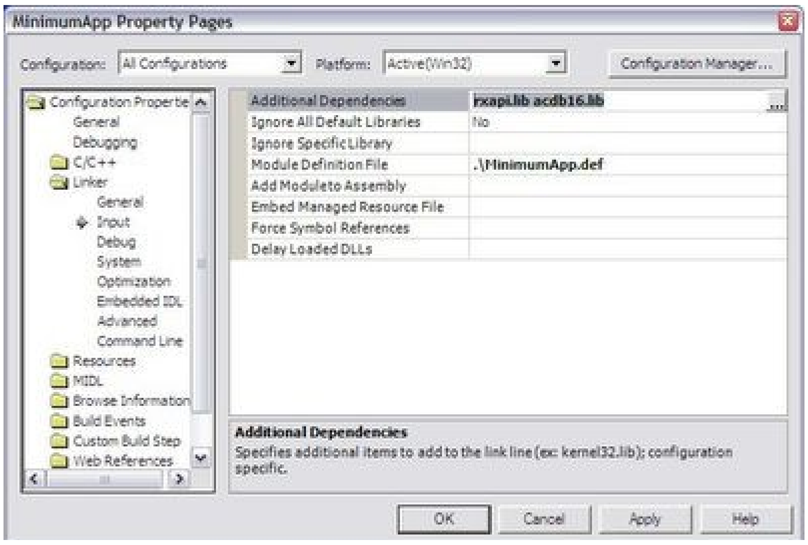
Configuration: All Configurations Platform: Active(Win32)

Output File	$(OutDir)/MinimumApp.arx
Show Progress	Not Set
Version	
Enable Incremental Linking	
Suppress Startup Banner	No
Ignore Import Library	No
Register Output	No
Additional Library Directories	

Output File
Override the default output file name. (/OUT:[file])

Still inside Linker node, select Input. Here we will add those libraries our application will use. This will depend on what features you are using inside your ObjectARX application. In this case, we will add just the basic two libraries called rxapi.lib and acdb16.lib.

Select the Additional Dependencies entry and click on ellipsis button. Type the two previously mentioned files. These libraries are located at lib folder. Remember that on previous classes I have talked about the features each library has built in.



Configuration: All Configurations
Platform: Active(Win32)

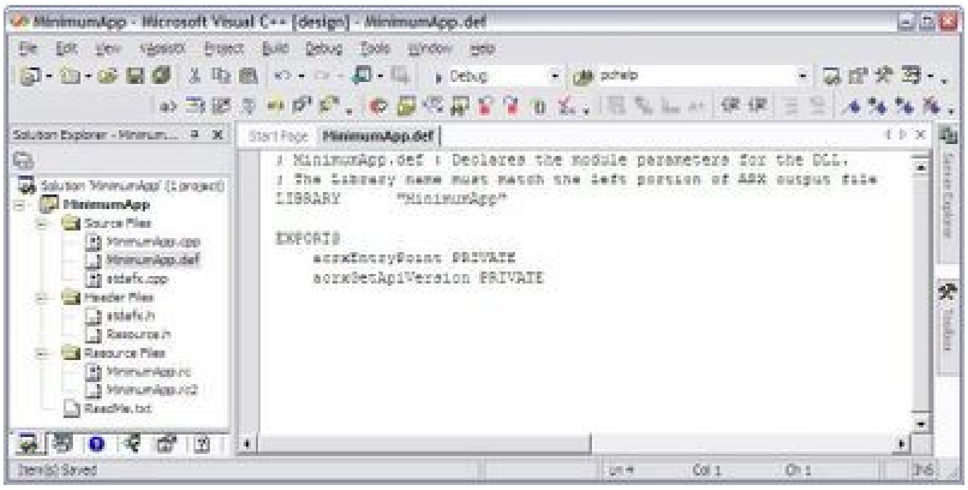
Additional Dependencies	rxapi.lib acdb16.lib
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	.\MinimumApp.def
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	

Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

Click OK to close Project Properties dialog. Now we still need to do some code typing. The first step is to edit the DEF file which is placed at Source Files folder of your project at Solution Explorer. Double click the DEF file and it will appear at the right portion of Visual Studio window. We will need to add the following lines, under the EXPORTS section of this file:

```
acrxEntryPoint PRIVATE
acrxGetApiVersion PRIVATE
```

Pay attention to the name between quotes in front of LIBRARY section inside this file. This name must be the same name you have entered on the output file. In other words, if your project generates a ABCD.arx file you need to have LIBRARY "ABCD" inside the DEF file.



```
; MinimumApp.def : Declares the module parameters for the DLL.
; The Library name must match the left portion of ARX output file
LIBRARY      "MinimumApp"

EXPORTS
     acrxEntryPoint PRIVATE
     acrxGetApiVersion PRIVATE
```

The next step is to change our StdAfx.h file which is the key compilation file. We will need to inform Visual Studio to use Release version of MFC libraries when our project is being compiled using the DEBUG directive.

To do that, open the StdAfx.h file which is located at Header Files folder inside Solution Explorer. Before the #include line, insert the following code:

```
#if defined(_DEBUG) && !defined(_FULLDEBUG_)
#define _DEBUG_WAS_DEFINED
#undef _DEBUG
#pragma message (" Compiling MFC header files in release mode.")
#endif
```

Now, scroll to the end of this file and add the following lines to manage the _DEBUG symbol back and to include basic .H files our application will need:

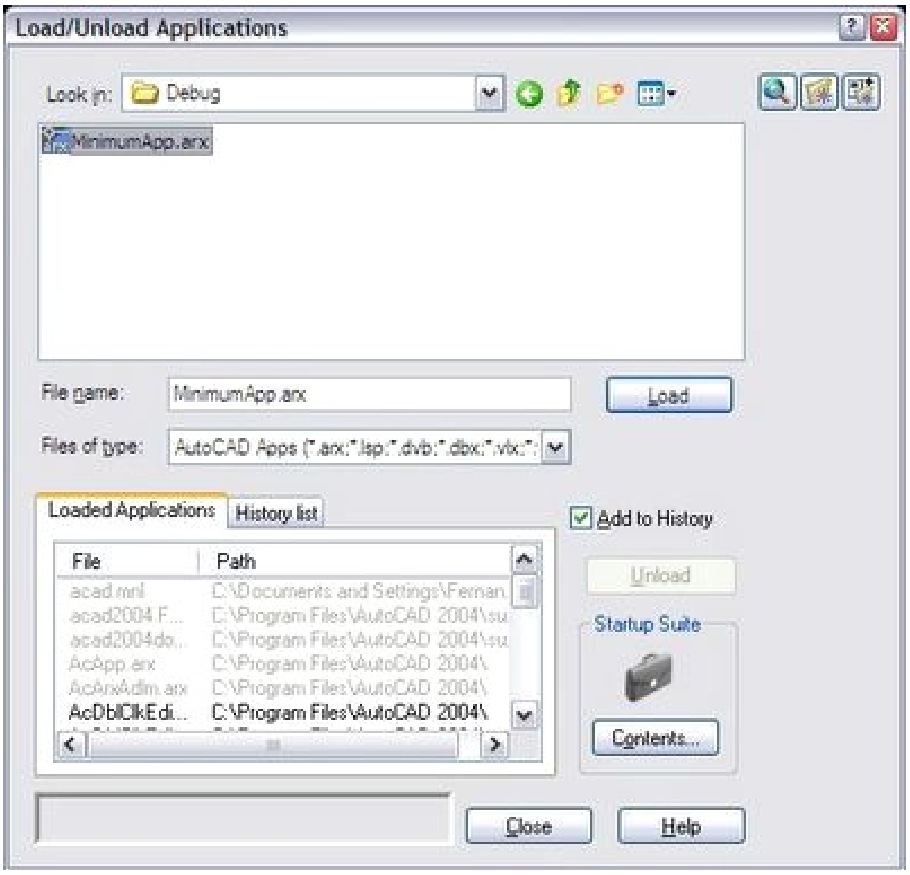
```
#ifdef _DEBUG_WAS_DEFINED
#define _DEBUG
#undef _DEBUG_WAS_DEFINED
#endif


// ObjectARX Includes
#include "rxregsvc.h"
#include "acutads.h"
```

The last step is to add our acrxEntryPoint method which is our application start point. Open the CPP file of your application which has the same name that you have set to your project plus the CPP extension. It is placed inside Source Files folder. Open if and scroll down to the end. Add the following lines:

```
// ObjectARX ENTRYPOINT
extern "C" AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode
msg, void* appId){
    switch(msg) {
        case AcRx::kInitAppMsg:
            acrxUnlockApplication(appId);
            acrxRegisterAppMDIAware(appId);
            acutPrintf("\nMinimum ObjectARX application loaded!");
            break;
        case AcRx::kUnloadAppMsg:
            acutPrintf("\nMinimum ObjectARX application
            unloaded!");
            break;
    }
    return AcRx::kRetOK;
}
```

Now we are ready to build our application. Open Build menu and select Build Solution (or F7 key). Visual Studio will compile and link and build your project. If you have followed all above steps carefully it will generate the application without any error.

Start AutoCAD and run APPLOAD command which will show the following dialog. Browse to your project and you will find the application inside the Debug folder which is the default compilation type. Select it and click Load button. A message will appear at the bottom of this dialog telling you that your ARX was successfully loaded or not!



Look in: Debug

MinimumApp.arx

File name: MinimumApp.arx

Files of type: AutoCAD Apps (*.arx;*.lsp;*.dvb;*.dbx;*.vlx;*.

[✓ Add to History]

File	Path
acad.mnl	C:\Documents and Settings\Fernan
acad2004.F...	C:\Program Files\AutoCAD 2004\su
acad2004do...	C:\Program Files\AutoCAD 2004\su
AcApp.arx	C:\Program Files\AutoCAD 2004\
AcArxAdm.arx	C:\Program Files\AutoCAD 2004\
AcDblClkEdi...	C:\Program Files\AutoCAD 2004\

That's it, your first ObjectARX application is loaded and is running inside AutoCAD!

Next class we will do the same using the ARXWizard. Stay tuned!

ObjectARX&Dummies 教程（三 b）——Minimum application using ARXWizard

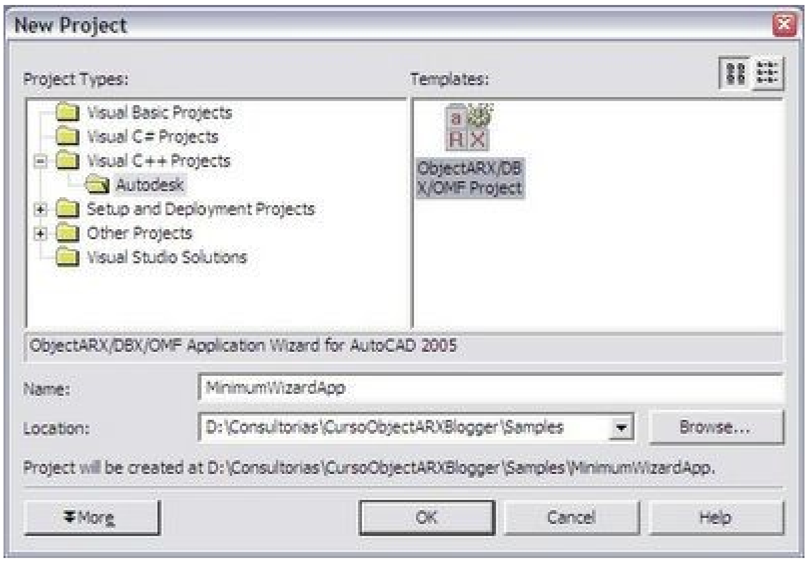
Class 3b - Minimum application using ARXWizard

Hello

On previous class I have presented the step by step way to create a minimum application by hand. Of course there is a easier way to do that but I think is very important that you understand correctly things that are behind the scenes.

This time we will use the ARXWizard tool which is provided by Autodesk through ObjectARX SDK. If you go to the \Utils folder you will find the install program. Go ahead, install it and allow Live Update to run at the first time. Do this with your Visual Studio .NET closed.

After you install it, open Visual Studio.NET, open File menu and start a New Project. The following dialog will appear and you will find a new node inside Visual C++ Projects folder which is called Autodesk. Select this node and the ObjectARX/DBX/OMF Project icon will appear at the right side as following:



New Project

Project Types:

- Visual Basic Projects
- Visual C# Projects
- Visual C++ Projects
  - Autodesk
- Setup and Deployment Projects
- Other Projects
- Visual Studio Solutions

Templates:

ObjectARX/DBX/OMF Project

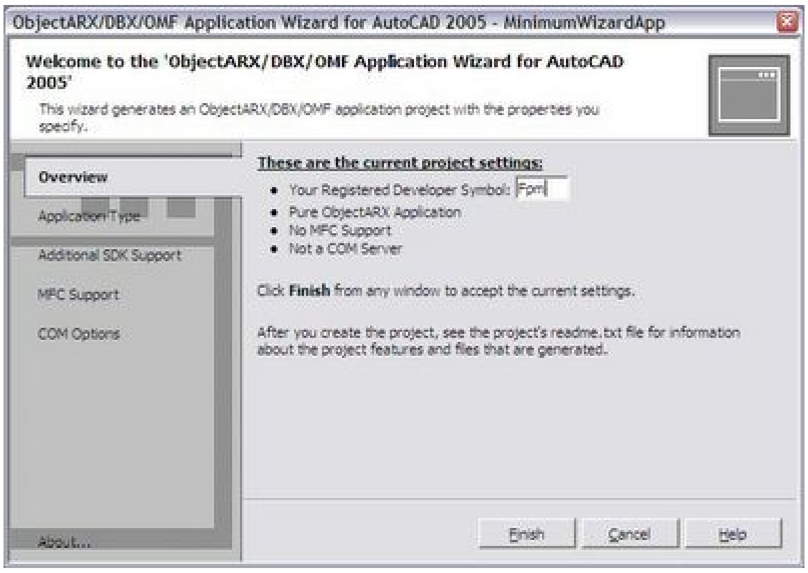
ObjectARX/DBX/OMF Application Wizard for AutoCAD 2005

Name: MinimumWizardApp

Location: D:\Consultorias\CursoObjectARXBlogger\Samples

Project will be created at D:\Consultorias\CursoObjectARXBlogger\Samples\MinimumWizardApp.

Fill out the Name field and specify the desired location to create the new project. Click OK to continue. The following dialog will appear:



ObjectARX/DBX/OMF Application Wizard for AutoCAD 2005 - MinimumWizardApp

Welcome to the 'ObjectARX/DBX/OMF Application Wizard for AutoCAD 2005'

This wizard generates an ObjectARX/DBX/OMF application project with the properties you specify.

These are the current project settings:

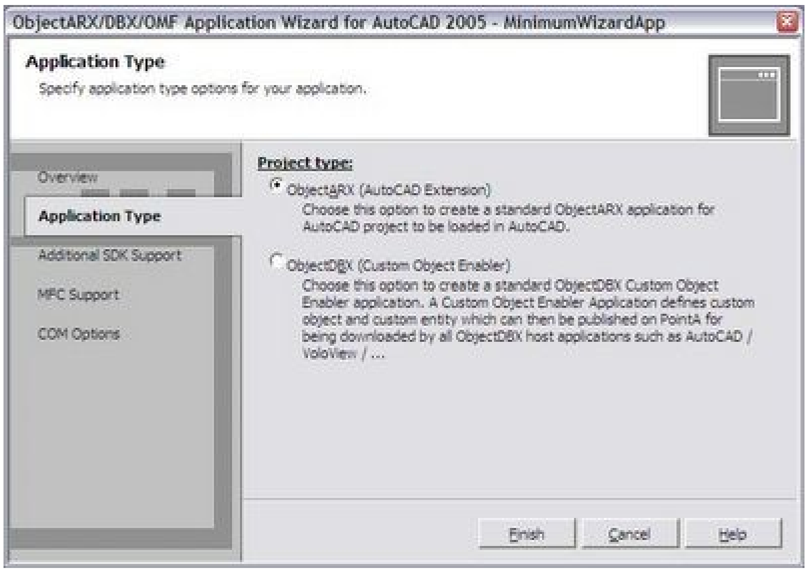
- Your Registered Developer Symbol: Fom
- Pure ObjectARX Application
- No MFC Support
- Not a COM Server

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

This dialog presents the steps to setup your new project. The first page, called Overview, shows some information and give you the opportunity to inform your RDS (Registered Developer Symbol). This label will be used to prefix anything that your code could implement and could conflict with other third-party applications. To allow this prefix to be unique, Autodesk provides (through ADN subscriptions) a way to register your prefix and inform other ADN members. Even you are note an ADN member you should create your own RDS. Use your initials, your 3 name first chars or any other name you find clear and useful.

The next step is to choose your desired Application Type. As I have mentioned before, ARXWizard suggests the ARX / DBX types which are basically the separation of Interfaces and Custom Classes. More details about the main differences between ARX and DBX can be found at ObjectARX documentation. This time we will choose the ObjectARX option as follows:


ObjectARX/DBX/OMF Application Wizard for AutoCAD 2005 - MinimumWizardApp

Application Type

Specify application type options for your application.

Project type:

[✓ ObjectARX (AutoCAD Extension)]
Choose this option to create a standard ObjectARX application for AutoCAD project to be loaded in AutoCAD.

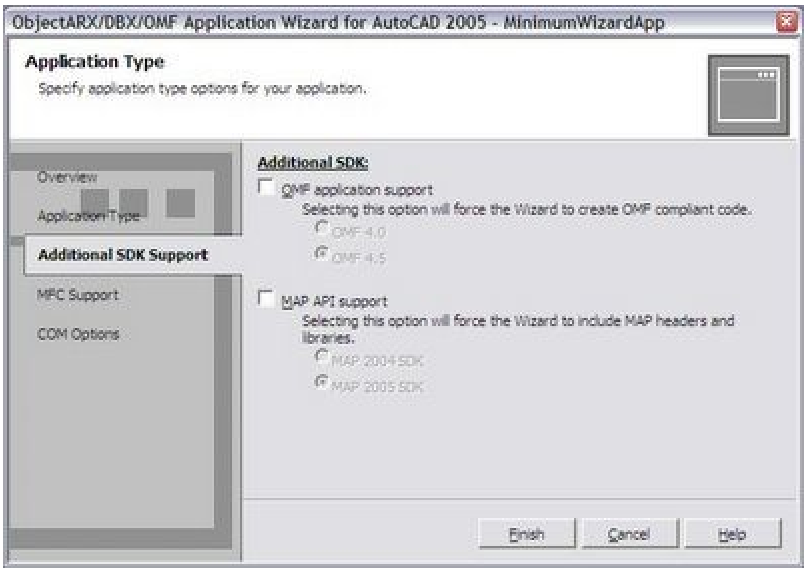
ObjectDBX (Custom Object Enabler)
Choose this option to create a standard ObjectDBX Custom Object Enabler application. A Custom Object Enabler Application defines custom object and custom entity which can then be published on PointA for being downloaded by all ObjectDBX host applications such as AutoCAD / VoloView / ...

The next step is about Additional SDK Support which allows you to extend basic ObjectARX features to an specific Autodesk vertical. There are two options:

- OMF Support: This is the SDK extension for Autodesk Architectural Desktop (aka ADT) which contains specific features that could be used if you plan to develop an ObjectARX application to run inside ADT;

- MAP API Support: This is the SDK extension for Autodesk MAP which contains extra features to be used if you plan to develop a MAP ObjectARX application.

In our case, we will develop an ObjectARX application targeting plain (or vanilla) AutoCAD so leave both blank.


ObjectARX/DBX/OMF Application Wizard for AutoCAD 2005 - MinimumWizardApp

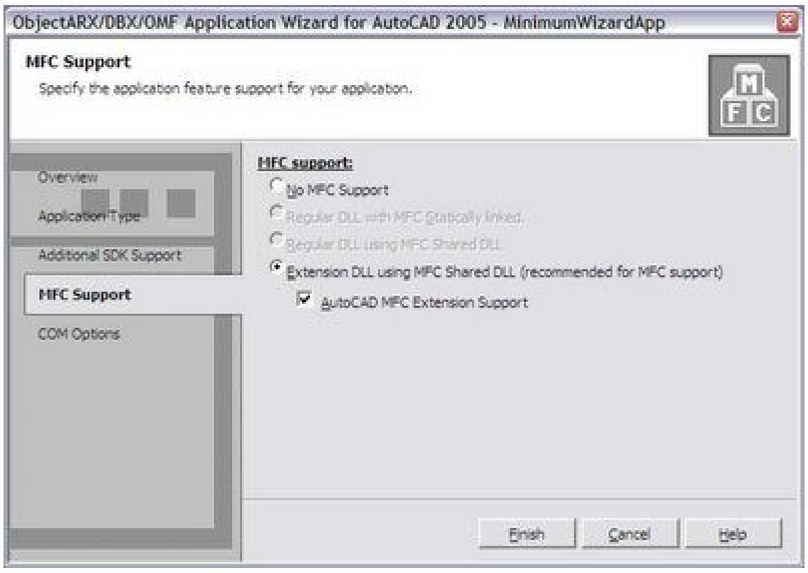
Application Type

Specify application type options for your application.

Additional SDK:

- [ ] OMF application support
  Selecting this option will force the Wizard to create OMF compliant code.
  - OMF 4.0
  - [✓ OMF 4.5]
- [ ] MAP API support
  Selecting this option will force the Wizard to include MAP headers and libraries.
  - MAP 2004 SDK
  - [✓ MAP 2005 SDK]

The next step is to specify MFC Support. As I mentioned before we will use MFC Extension DLL project type. This dialog also offers the option to enable AutoCAD MFC Extension Support which will allow you to use specific AutoCAD controls like LineType combo boxes, Color combo boxes, Dockable dialogs, etc. This is pretty handy once those controls are not so simple to implement from scratch. Select Extension DLL and enabled AutoCAD MFC Support:


ObjectARX/DBX/OMF Application Wizard for AutoCAD 2005 - MinimumWizardApp

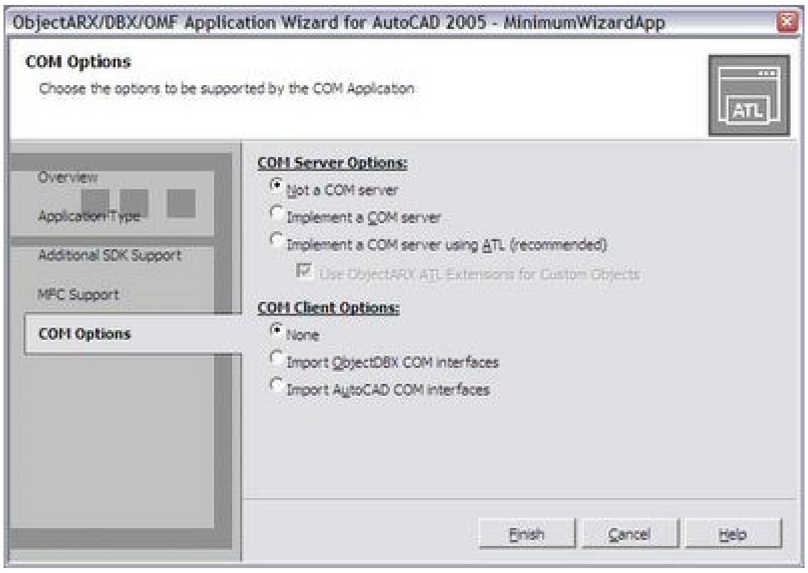
MFC Support

Specify the application feature support for your application.

MFC support:

- No MFC Support
- Regular DLL with MFC Statically linked
- Regular DLL using MFC Shared DLL
- [✓ Extension DLL using MFC Shared DLL (recommended for MFC support)]
  - [✓ AutoCAD MFC Extension Support]

The last step is dedicated to COM related stuff. ObjectARX supports COM implementations on both Server and Client sides. As COM programming is very complex and is beyond this course scope so I will not cover it.


ObjectARX/DBX/OMF Application Wizard for AutoCAD 2005 - MinimumWizardApp

COM Options

Choose the options to be supported by the COM Application

COM Server Options:

- [✓ Not a COM server]
- Implement a COM server
- Implement a COM server using ATL (recommended)
  - [✓ Use ObjectARX ATL Extensions for Custom Objects]

COM Client Options:

- [✓ None]
- Import ObjectDBX COM interfaces
- Import AutoCAD COM interfaces

Leave the options on the Not a COM Server and None. Click Finish to proceed.

Now you can open the project files and see what the ARXWizard has done for you. There are a lot of differences between the project we have created on previous class and the present project because ARXWizard use different implementations using handy classes. We will cover these features several times with our upcoming samples on the next classes.

Compile and Build the project and try to load the resulting ObjectARX application inside AutoCAD. You probably will be able to successfully load it

ObjectARX&Dummies 教程（四）——Object Lifecycle

Class 4 - Object Lifecycle

Some Additional Concepts

Now you are able to create a minimum ObjectARX project we will continue the course with some additional concepts.

As I have mentioned before, AutoCAD's database is well organized to allow simple and direct manipulation of its objects. Generally we have two basic types: Containers and Objects.

Containers are special objects that provide a simple and efficient mechanism to store, edit and persist objects or collections. They are optimized to allow quick access with minimum overhead. Each type of object has an appropriate container where you should always store your object. There are several containers inside AutoCAD's database but some of the most common are: LayerTable, LinetypeTable and BlockTable. Each container class has standard access methods and most of them also offer a companion class to iterate through its items. Every time you create an object and would like to store it inside AutoCAD's database you need to follow its container protocol to store and persist is as well.

In other hand, Objects (including entities) are the most basic types and represent each element inside AutoCAD. They are implemented through specific classes with standard and specific methods. Some of them could be derived inside your application to allow customization.

Every database resident object has an exclusive identification called ObjectId. This identification is the "name" of each object inside database and it is used to reference, open and manipulate objects.

This could be a difficult concept for those who are fluent in standard C++ language because inside ObjectARX you don't delete a pointer to a database resident object. What? Yes, this is a little bit strange but AutoCAD has several reasons to do that including performance, memory management and other aspects.

So, how can I manipulate objects?

Don't panic. You just need to keep in mind some basic but essential rules:

1) Database resident objects should never be deleted even you have erased them!

2) If you have allocated an object but did not added it to database yet, go ahead...delete the pointer!

3) If you need to get a pointer to an object to manipulate it, acquire its ObjectId and use the appropriate container method to get its pointer. That's it? NO. You need to CLOSE the objects just after you finish to use it. Call the close() method or end its transaction to inform AutoCAD that you are done! (DON'T FORGET THIS ONE BECAUSE AutoCAD

WILL TERMINATE)

Most of bugs you will face at your first application will be have something to do with the above rules. Trust me!

Object Ownership and Relationship

Objects can refer each other using their ObjectId. This can be an ownership relation or just a relationship. If you think about a layer you will understand what is involved with this concept. The LayerTable container owns its records which are objects (Layers in this case). Each Layer ObjectId is referred inside each entity. Exactly due that you can't remove a Layer from a DWG file until all entities that uses this layer are erased or has its associated layer changed.

There are several examples of ownerships and relationships inside AutoCAD. During our course you will get this concept easily when we have to manipulate basic objects.

Creating a Layer

On previous classes I have presented some code fragment to explain how to create simple entities. Now I will present a simple code to create a Layer just to allow you to feel how much protocol is involved in such operation:

```
    AcDbLayerTable* pLayerTbl = NULL;
    // Get the current Database
    AcDbDatabase*pDB=acdbHostApplicationServices()-
    >workingDatabase();
    // Get the LayerTable for write because we will create a new entry
    pDB->getSymbolTable(pLayerTbl,AcDb::kForWrite);
    // Check if the layer is already there
    if (!pLayerTbl->has("MYLAYER")) {
    // Instantiate a new object and set its properties
AcDbLayerTableRecord *pLayerTblRcd = new  AcDbLayerTableRecord;
    pLayerTblRcd->setName("MYLAYER");
    pLayerTblRcd->setIsFrozen(0); // layer set to THAWED
    pLayerTblRcd->setIsOff(0); // layer set to ON
    pLayerTblRcd->setIsLocked(0); // layer un-locked
    AcCmColor color;
    color.setColorIndex(7); // set layer color to white
    pLayerTblRcd->setColor(color);
    // Now, add the new layer to its container
    pLayerTbl->add(pLayerTblRcd);
    // Close the new layer (DON'T DELETE IT)
    pLayerTblRcd->close();
    // Close the container

    pLayerTbl->close();

    }

    else {

    // If our layer is already there, just close the container and continue

    pLayerTbl->close();

    acutPrintf("\nMYLAYER already exists");

    }
```

ObjectARX&Dummies 教程（五）——Object Management

Class 5 - Object Management

Introduction

On previous class we have talked about object lifecycle. On this class we will go further on how to manage AutoCAD objects. As I have mentioned every object has its own identification called ObjectId. This is the key for acquiring its pointer and perform read or write operations.

The standard access method is made by an OPEN operation (for write, for read or for notify), some operations and a CLOSE method. Another approach, much more efficient is through transactions. This mechanism is much more secure and efficient. Let's talk about both methods.

Using standard OPEN / CLOSE method

This is the most used method but, in other hand, is the most unsafe because you may forget to close the object and then cause access violations or even fatal errors that will cause AutoCAD to terminate.

The global standard function to open objects is called acdbOpenObject(). This function will open every object derived from AcDbObject class and will provide you a C++ pointer to access the object properties and methods. One of this function signatures is the following:

inline Acad::ErrorStatus acdbOpenObject(AcDbObject *& pObj, AcDbObjectId id, AcDb::OpenMode mode, bool openErased);

pObj	Output pointer to the opened object
id	Input the object ID of the object to open
mode	Input mode to open object
openErased	Input Boolean indicating whether it's OK to open an erased object

This function receives an empty pointer to AcDbObject by reference that will be filled out by AutoCAD if there is an object with the provided input variable id. Further you need to provide your action intention on the object which can be WRITE, READ or NOTIFY. The latest parameter indicate if you would like to open the object if it is on erased status. Remember we have talked that erased objects remain inside AutoCAD database until the next save operation.

The opening intention is very important because it will limit or not what you can do with the object. If you open an object for READ you will not be able to call methods that modify the object's state. In other hand, if you open the object for WRITE you will be able to both modify and read object data. Ah, so is better to always open the object for WRITE?

definitely NOT!

When you open an object for WRITE AutoCAD fires several pre and post procedures on it that causes some performance overhead. If your routine opens several objects and you use the WRITE flag you will certainly loose performance.

The same object can be open for READ up to 256 times without close but it is not recommended. You should always close the object as soon as possible. If an object is opened for WRITE you can't open it a second time for WRITE. Basically, you need to follow the following rules:

Opening objects in different modes			
Object opened for:	kForRead	kForWrite	kForNotify
openedForRead	eAtMaxReaders	eWasOpenForRead	(Succeeds)
openedForWrite	eWasOpenForWrite	eWasOpenForWrite	(Succeeds)
openedForNotify	eWasOpenForNotify	eWasOpenForNotify	eWasOpenForNotify
wasNotifying	(Succeeds)	eWasNotifying	eWasNotifying
Undo	eWasOpenForUndo	eWasOpenForUndo	(Succeeds)

The best performance approach is to always open the object for READ, analyze if you will modify it and only after this analysis, upgrade its open operation to WRITE using the upgradeOpen() method. It will switch the object's state from READ to WRITE. To get back to READ use the downgradeOpen() method. These methods are very useful. A simple operation to use these methods would be:

```
void changeColor(AcDbObjectId id) {
    AcDbEntity* pEnt = NULL;
    if (acdbOpenObject(pEnt, id, AcDb::kForRead) == Acad::eOk) {
        if (pEnt->colorIndex() != 3) {
            pEnt->upgradeOpen();
            pEnt->setColorIndex(3);
        }
        else {
                acutPrintf("\nEntity already has color=3");
            }
        pEnt->close();
    }
}
```

Using Transaction method

Transactions are a better and much more efficient method to manage objects. They can be nested and this allows you to perform long operations without the limitation of READ and WRITE states. Basically you need to open a transaction, perform your desired modifications, and at the end, perform an end or abort transaction operation.

Transactions are pretty good to use when your application uses dialog boxes that change objects. At your dialog opening you start a new transaction and, depending on user click on OK or CANCEL button, you call end or abort transaction methods.

When you abort a transaction the whole contained modification are cancelled. In fact, modifications are really applied only when you end a transaction. Another great feature is that you can open an object for WRITE several times as you can open for READ at the same time.

You don't need to close objects opened through a transaction. The end or abort method will close every object opened and will perform modifications as necessary. It is not recommended to mix standard OPEN / CLOSE approach with TRANSACTIONS due some design limitations. You could read more details about this inside SDK documentation. Now, let's the see the same operation we did above using transactions:

```
void changeColor(AcDbObjectId id) {
    AcDbEntity* pEnt = NULL;
    acdbTransactionManager->startTransaction();
    if (acdbTransactionManager->getObject((AcDbObject*&)pEnt, id,
    AcDb::kForRead) == Acad::eOk) {
        if (pEnt->colorIndex() != 3) {
            pEnt->upgradeOpen();
            pEnt->setColorIndex(3);
        }
    else {
            acutPrintf("\nEntity already has color=3");
        }


    }
    acdbTransactionManager->endTransaction();
}
```

This time you open the object using the getObject() method which is much like acdbOpenObject() but YOU DON'T NEED TO CLOSE the object. The whole process is ended at endTransaction() method call. At that time all operations are applied in one operation.

ObjectARX&Dummies 教程（六）——Entities

Class 6 - Entities

Introduction

Entities are objects that has a graphical representation. They could be simple or complex depending on how many features and functionalities they implement. Entities are stored into BlockTableRecord container objects. Each of these containers will keep its entities until they are erased or Database is destroyed. As any other database resident object, each entity can be accessed through its unique ObjectId. Using its ObjectId we can then acquire its pointer (for Read, Write or Notify as we saw on previous class) and then perform desired operations.

Some special entities contains also another objects to simplify implementation and management. A good example of this approach is AcDb3dPolyline which has a collection of AcDb3dPolylineVertex objects that represents each of its vertexes.

Entity Properties

AutoCAD entities has several properties and some of them are common to all kind of entity. These properties are stored into entity's base class called AcDbEntity. This class, also derived from AcDbObject implements several common functionalities that will be used by every derived class and related implemented entity.

If we create a circle (AcDbCircle), for example, it will contain some properties that came from AcDbEntity. These properties are:

- Color
- Linetype
- Linetype scale
- Visibility
- Layer
- Line weight
- Plot style name

These properties has specific access methods that will allow you to read or modify them accessing the AcDbEntity base class. So, if we get our AcDbCircle entity and would like to change its color we just need to open it for Write, access the proper method and then close the circle.

If we need to build an application that only access these properties we really don't need to know what kind of entity we are opening. In this situation we just need to open our entity, get its pointer as an AcDbEntity pointer and access the desired method.

Entity Transformations

Each AutoCAD entity is placed into a 3D space. You already know that we can move, rotate, scale, align and many other modifications over an entity. AutoCAD threats most of these operations using geometric algebra using matrixes. Remember that we have talked about ObjectARX classes and, specially in this case, about AcGe prefixed classes. The AcGe classes are geometric classes which will allow you to perform simple and complex geometric operations inside AutoCAD.

So let's suppose you need to perform a rotation over several entities (circles, polylines, lines, etc.) and need to do this with minimum effort and basic geometric knowledge. No big deal! We just need to build a transformation matrix and call the appropriate method called transformBy() implemented by AcDbEntity class. Yes, every entity could be potentially transformed!

This function receives an object of class AcGeMatrix3d which represents the matrix to be applied to the entity that will perform some geometric operation. This could be a transformation matrix, a rotation matrix and so on.

This class has wonderful utility functions that will reduce your work a lot! Among these functions I would like to quote the following:

setToRotation: You pass in the desired angle (in radians), the axis vector of rotation and the point of rotation. With this parameters this function will fill the AcGeMatrix3d with all elements. After that, just call your entity's transformBy() method passing this matrix in;

setToTranlation: You just pass in a 3D vector (AcGeVector3d) which represents the transformation you would like to perform. After that, do the same operation as mentioned above;

setToMirroring: This function has 3 versions that receives a point (AcGePoint3d), a line (AcGeLine3d) or a plane (AcGePlane). Depending on what type of parameter you pass in it will build the proper matrix to mirror your entity! Great hum?

Entity Intersection

Another important functionality implemented at AcDbEntity level is entity intersection. Probably one of your future products will need to analyze entities intersections. The method intersectWith() is the responsible to do this job for you.

The most common signature of this method receives the argument entity pointer (the entity you would like to test with yours), the intersection type, an array of 3D points to be filled out by this functions with intersection points found and, optionally the GS marker of both entities which represents the subentity index.

The intersection type must be one of the following operands:

kOnBothOperands: neither entity is extended;

- kExtendThis: extend this entity;
- kExtendArg: extend argument entity;
- kExtendBoth: extend both entities.

If these two entities intercept each other (obviously depending on which type of intersection you specify) the passed array will receive the intersection points. This function is very useful and uses the core geometric engine of AutoCAD which make it fast and reliable.

Our next class will be a demo example and I would give you a couple of days to accomplish it and then I will post my solution to it. Stay tuned!

ObjectARX&Dummies 教程（七）——Containers

Class 7 - Containers

Hello

On this class I will present the concepts and features of ObjectARX container objects. We have talked a little bit about them before but now we will go into further details.

Introduction

The container object purpose is to store and manage objects of the same type or class. There are two types of containers: Symbol Tables and Dictionaries. Each type of container has some specific functionalities that were designed to allow easy and efficient access methods.

Symbol Tables

This type of container is designed to store the so called records. Each Symbol Table object store its records using an unique entry name. Through this entry you can obtain the record pointer and read or write information. The container may also receive new entries or even has entries removed (in case they are not used by other objects).

To walk through an object container entries you will need to use a proper iterator which will allow you to get entries and access its objects. AutoCAD has some Symbol Tables to store layers, linetypes, text styles and other objects. As these containers work almost the same way, there is a common base class for each of Symbol Tables, its records and the proper iterators.

The Symbol Table class tree is as follows:

```
AcDbSymbolTable
    AcDbAbstractViewTable
        AcDbViewportTable
        AcDbViewTable
    AcDbBlockTable
    AcDbDimStyleTable
    AcDbLayerTable
    AcDbLinetypeTable
    AcDbRegAppTable
    AcDbTextStyleTable
    AcDbUCSTable
//--------------------------------
AcDbSymbolTableRecord
    AcDbAbstractViewTableRecord
        AcDbViewportTableRecord
        AcDbViewTableRecord
    AcDbBlockTableRecord
    AcDbDimStyleTableRecord
    AcDbLayerTableRecord
    AcDbLinetypeTableRecord
    AcDbRegAppTableRecord
    AcDbTextStyleTableRecord
    AcDbUCSTableRecord
//--------------------------------
AcDbSymbolTableIterator
    AcDbAbstractViewTableIterator
        AcDbViewportTableIterator
        AcDbViewTableIterator
    AcDbBlockTableIterator
    AcDbDimStyleTableIterator
    AcDbLayerTableIterator
    AcDbLinetypeTableIterator
    AcDbRegAppTableIterator
    AcDbTextStyleTableIterator
    AcDbUCSTableIterator
```

So, to create a layer, for instance, you will need to:

- Open current Database;
- Open AcDbLayerTable (for write);
- Create an AcDbLayerTableRecord (using new operator);
- Configure the AcDbLayerTableRecord;
- Add it to AcDbLayerTable which is its proper container;
- Close the record;
- Close the container.

```
void createLayer() {

    AcDbLayerTable *pLayerTbl = NULL;

    acdbHostApplicationServices()->workingDatabase()

    ->getSymbolTable(pLayerTbl, AcDb::kForWrite);

    if (!pLayerTbl->has("MYLAYER")) {

AcDbLayerTableRecord*pLayerTblRcd=newAcDbLayerTableRecord;

        pLayerTblRcd->setName("MYLAYER");

        AcCmColor color;

        color.setColorIndex(1); // red

        pLayerTblRcd->setColor(color);

        pLayerTbl->add(pLayerTblRcd);

        pLayerTblRcd->close();

    } else

        acutPrintf("\nLayer already exists");

    pLayerTbl->close();

}
```

To list all existing layers:

- Open current Database;
- Open AcDbLayerTable (for read);
- Create an AcDbLayerTableIterator;
- Perform a loop through container entries;
- Get the key name for each entry;
- Close the container.

```
void iterateLayers() {

    AcDbLayerTable* pLayerTbl = NULL;

    acdbHostApplicationServices()->workingDatabase()

    ->getSymbolTable(pLayerTbl, AcDb::kForRead);

    AcDbLayerTableIterator* pLayerIterator;

    pLayerTbl->newIterator(pLayerIterator);

    AcDbLayerTableRecord* pLayerTblRcd;

    char *pLName;

    for (; !pLayerIterator->done(); pLayerIterator->step()) {

        pLayerIterator->getRecord(pLayerTblRcd, AcDb::kForRead);

        pLayerTblRcd->getName(pLName);

        pLayerTblRcd->close();

    acutPrintf("\nLayer name: %s",pLName);

    acutDelString(pLName);

}

    delete pLayerIterator;

    pLayerTbl->close();

}
```

Dictionaries

This type of container is designed to store generic AcDbObject derived class objects. This container is very useful because we can also store our custom objects inside it. The dictionary structure is much like a tree structure where we have nodes and entries. Inside the same node, entries can not repeat its name because they need to be unique inside the same level. These are the so called Key entries and each Key entry will map to an AcDbObject pointer which can be retrieved directly or through an interator (AcDbDictionaryIterator).

To store an object we need to create an entry using the setAt() method passing also the object pointer which we already have instantiated with the new operator. After add this object we need to close() it. AcDbDictionary container will return the given AcDbObjectId for each entry.

This container is also used by some AutoCAD features like groups and multiline styles. We will cover more about Dictionaries on the Custom Objects chapter.

ObjectARX&Dummies教程（八）——Selection Sets

Class 8 - Selection Sets

Hello,

On this class we will cover the first ways we can interact with user to allow our application to get information from drawing screen You probably will need to use this method inside your application.

Introduction

This is one of the most important ways to interact with user because it will allow you to get information from drawing screen through selected entities. Some times you will request user to select entities individually and sometimes you will select them using a filter.

A selection set is a group of entities which are currently selected by an user or by an application. The most important concept involved when selecting entities from screen is that AutoCAD will return their names through a type called ads_name. This type contains the entity name (which is valid only on the current session) and it can be converted to ObjectId using the acdbGetObjectId() global function:

Acad::ErrorStatus acdbGetObjectId (AcDbObjectId& objId, const ads_name objName);

This function receives the ads_name and convert it to an AcDbObjectId. Most of selection set functions will still use the ads_name as parameters and on theses cases you don't need to convert it. The ads_name can store several entities or just one. This will depend on how you or the user has performed the selection.

The selection is made using a function called acedSSGet() which will apply a selection or prompt the user to do that. The function signature is:

```
int acedSSGet (const char *str, const void *pt1, const void *pt2,
const struct resbuf *entmask, ads_name ss);
```

How to use

It receives a selection option, two points, a mask and returns the resulting selection set. After use the selection set it needs to be released and this is done through the acedSSFree() function The selection option will instruct AutoCAD interface to do one of the following methods:

Selection Code	Description
NULL	Single-point selection (if pt1 is specified) or user selection (if pt1 is also NULL)
#	Nongeometric (all, last, previous)
:$	Prompts supplied
.	User pick
:?	Other callbacks
A	All
B	Box
C	Crossing
CP	Crossing Polygon
:D	Duplicates OK
:E	Everything in aperture
F	Fence

G	Groups
I	Implied
:K	Keyword callbacks
L	Last
M	Multiple
P	Previous
:S	Force single object selection only
W	Window
WP	Window Polygon
X	Extended search (search whole database)

This way we can perform the selection by several ways. Some examples are presented below:

```
ads_point pt1, pt2;
ads_name ssname;
pt1[X] = pt1[Y] = pt1[Z] = 0.0;
pt2[X] = pt2[Y] = 5.0; pt2[Z] = 0.0;
// Get the current PICKFIRST or ask user for a selection
acedSSGet(NULL, NULL, NULL, NULL, ssname);
// Get the current PICKFIRST set
acedSSGet("I", NULL, NULL, NULL, ssname);
// Repeat the  previous selection set
acedSSGet("P", NULL, NULL, NULL, ssname);
// Selects the last  created entity
acedSSGet("L", NULL, NULL, NULL, ssname);
// Selects entity passing through point (5,5)
acedSSGet(NULL, pt2, NULL, NULL, ssname);
// Selects entities inside the window from point (0,0) to (5,5)
```

```
acedSSGet("W", pt1, pt2, NULL, ssname);
```

Using Selection filters

Filters are a powerful way to speed up selection sets and avoid runtime operations to verify entities. You can use single filters or composed filters. Each filter is specified through a structure called resbuf. A resbuf is a linked list which store several types of information and may contains several items. To use a filter we need to construct it and pass it as a parameters of acedSSGet() method. The selection is performed but each selected entity will need to respect the filter. There are a lot of filters we can create and the SDK documentation cover all of them. The most used examples are presented below:

```
struct resbuf eb1, eb2;
char sbuf1[10], sbuf2[10];
ads_name ssname1, ssname2; eb1.restype = 0; // Entity name filter
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1;
eb1.rbnext = NULL;
// Retrieve all circles
acedSSGet("X", NULL, NULL, &eb1, ssname1); eb2.restype = 8;
//Layer name filter
strcpy(sbuf2, "0");
```

```
eb2.resval.rstring = sbuf2;
eb2.rbnext = NULL;
// Retrieve all entities on layer 0
acedSSGet("X", NULL, NULL, &eb2, ssname2);
```

Modifying entities through a selection set

To modify entities inside a selection set we need to walk through selection items, get each one, convert the ads_name to an ObjectId, open the entity for write, modify it and then close it. This operation can also be done using a transaction which is, in long operations, much better.

To show you how to walk through a selection set I will present a short code to select all CIRCLE entities inside the drawing and then change its color to red. The operation is pretty simple and is done this way:

```
// Construct the filter
struct resbuf eb1;
char sbuf1[10];
eb1.restype = 0; // Entity name
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1;
eb1.rbnext = NULL;
// Select All Circles
ads_name ss;
```

```
if (acedSSGet("X", NULL, NULL, &eb1, ss) != RTNORM){
    acutRelRb(&eb1);
    return;
}
// Free the resbuf
acutRelRb(&eb1);
// Get the length (how many entities were selected)
long length = 0;
if ((acedSSLength( ss, &length ) != RTNORM) || (length == 0)) {
    acedSSFree( ss );
    return;
}
ads_name ent;
AcDbObjectId id = AcDbObjectId::kNull;
// Walk through the selection set and open each entity
for (long i = 0; i < length; i++) {
    if (acedSSName(ss,i,ent) != RTNORM) continue;
    if (acdbGetObjectId(id,ent) != Acad::eOk) continue;
    AcDbEntity* pEnt = NULL;
if (acdbOpenAcDbEntity(pEnt,id,AcDb::kForWrite) != Acad::eOk)
    continue;
// Change color
pEnt->setColorIndex(1);
pEnt->close();
}
// Free selection
acedSSFree( ss );
```

I have used some new functions (like acdbOpenAcDbEntity) that are also part of ObjectARX SDK. Pay attention to the memory releases regarding to selection set and resbuf types. Note that I have used also a function called acedSSLength() to get the length of selection set.

The acedSSName() function get an at the passed index. If we have more than one entity selected this loop will get every single entity into this selection set.

See you next class.

ObjectARX&Dummies 教程（九）—— Interacting with AutoCAD

Class 9 - Interacting with AutoCAD

Hello

On the last class I have presented how to perform selection sets. This class I will show how can you interact with AutoCAD using global functions and acquiring information such as numbers, coordinates, system variables and much more.

Invoking Commands

ObjectARX provide us two global functions that allows us to invoke registered commands. This functionality is very handy and will help users to perform quick operations that don't require complex procedures. Even this method is quite simple you should avoid using it in complex and huge operations. This method may also create problems when dealing with events handling.

The two provided functions are acedCmd() and acedCommand(). The first one invokes the command through a passed in resbuf list which will inform all command parameters. The second function will receive a variable number of parameters which will reproduce the way you fire the command from the prompt interface. Below are these functions signature:

```
int acedCmd(const struct resbuf * rbp);
int acedCommand(int rtype, ... unnamed);
```

To build the resbuf list when using acedCmd() there is a utility function called acutBuildList() which constructs this linked list easily. You just need to pass paired values with codes that describe the types and end the list with a 0 or RTNONE value. Another good practice is to clear the command prompt, calling acedCommand(RTNONE) , after issued the command. Don't forget to free memory used, when using resbuf pointers, through the acutRelRb() utility function to avoid memory leaks. There are several ways to use theses functions and I will show some of them below:

acedCmd():

a) Moving the last created entity based on (0,0,0):

```
ads_point pt;
pt[0] = pt[1] = pt[2] = 0.0;
struct resbuf *Mv;
Mv = acutBuildList(RTSTR,"_MOVE",RTSTR,"_LAST",RTSTR,"",
RTPOINT,pt,RTSTR,PAUSE,0);
acedCmd(Mv);
acedCommand(RTNONE);
acutRelRb(Mv);
```

b) Calling a "redraw" native command:

```
struct resbuf *cmdlist;
cmdlist = acutBuildList(RTSTR, "_REDRAW", 0);
acedCmd(cmdlist);
acedCommand(RTNONE);
acutRelRb(cmdlist);
acedCommand():
```

a) Calling a ZOOM command and pausing for user input:

```
acedCommand(RTSTR, "Zoom", RTSTR, PAUSE, RTNONE);
```

b) Creating both a Circle and a Line entities:

```
acedCommand(RTSTR, "circle", RTSTR, "10,10",RTSTR, PAUSE,
RTSTR, "line", RTSTR, "10,10", RTSTR, "20,20", RTSTR, "", 0);
```

System Variables

Your application will probably need to access AutoCAD system variables that can be read or write. ObjectARX provide two functions to deal with these variables using the resbuf structure to access and/or modify values. The function are called acedGetVar() and acedSetVar() and below are their signatures:

```
int acedGetVar(const char * sym,struct resbuf * result);
int acedSetVar(const char * sym,const struct resbuf * val);
```

The first parameter is the variable name the second the resbuf pointer to set / get information. The following example show how to change the FILLET radius which is stored through a system variable:

```
struct resbuf rb, rb1;
acedGetVar("FILLETRAD", &rb);
rb1.restype = RTREAL;
rb1.resval.rreal = 1.0;
acedSetVar("FILLETRAD", &rb1);
```

It is very important that you specify the correct type of resbuf item acquired.

In this case, the FILLET radius is a real number which is RTREAL type.

User Input Functions

There are additional global functions to allow interaction with users via command prompt interface. Each of these functions could be used alone or with other ones. The following table shows what each function does:

acedGetInt	Gets an integer value
acedGetReal	Gets a real value
acedGetDist	Gets a distance
acedGetAngle	Gets an angle (oriented to 0 degrees specified by the ANGBASE)
acedGetOrient	Gets an angle (oriented to 0 degrees at the right)
acedGetPoint	Gets a point

acedGetCorner	Gets the corner of a rectangle
acedGetKword	Gets a keyword
acedGetString	Gets a string

Each of these functions returns a int number as a result code that could be one of the following:

RTNORM	User entered a valid value
RTERROR	The function call failed
RTCAN	User entered ESC
RTNONE	User entered only ENTER
RTREJ	AutoCAD rejected the request as invalid
RTKWORD	User entered a keyword or arbitrary text

AutoCAD allows you to prevent invalid values when user respond to your input functions. This feature can be made through the acedInitGet() function which can receive one or a combination of the following values:

RSG_NONULL	Disallow null input
RSG_NOZERO	Disallow zero values
RSG_NONEG	Disallow negative values
RSG_NOLIM	Do not check drawing limits, even if LIMCHECK is on
RSG_DASH	Use dashed lines when drawing rubber-band line or box
RSG_2D	Ignore Z coordinate of 3D points (acedGetDist() only)
RSG_OTHER	Allow arbitrary input—whatever the user enters

The following example shows how to acquire a value greater than zero:

```
int age = -1;

acedInitGet(RSG_NONULL  RSG_NOZERO  RSG_NONEG,
NULL);

acedGetInt("How old are you? ", &age);
```

ObjectARX&Dummies 教程（十）—— Using MFC

Class 10 - Using MFC

This class is not fully related to AutoCAD and ObjectARX but it covers an overview of MFC library. This library will allow you to create rich interfaces for your ObjectARX applications easily and with professional style.

Introduction

MFC (Microsoft Foundation Classes) is a powerful library made to allow windows programmers to easily create rich interfaces following the windows standard interface.

The MFC class tree is huge and obviously it is not the objective of this class to make you an MFC wizard. I will present here the main concepts involved when using MFC inside AutoCAD and how to take advantage of its power.

If you would like to use MFC inside your ObjectARX application you will need to link it with MFC libraries which can be done dynamically or statically. Currently ObjectARX only support dynamic linked MFC DLL applications because some of AutoCAD libraries were dynamic linked with MFC and you can't mix static and dynamic linked libraries. To make your application dynamic linked with MFC you need to select MFC Extension DLL option when creating your application.

Resource Management

One of the most important concepts when dealing with MFC libraries is that they are based on resources. Resources are rich information such as bitmaps, icons, string tables, dialog layouts, etc. Each DLL has its own resource package that could be used only by itself or shared with other DLLs (Resource only DLLs for example).

The problem is that your ObjectARX application is running into AutoCAD host application which has its own resources. To solve this problem you should switch the resource context to your DLL, use them, and then switch back to AutoCAD. This can be done manually but there is a couple of classes to help you to easily do that.

The first class, called CAcExtensionModule which will handle your module's own resource and the default resources. To make your application take advantage of this class you will need to create an instance of this class inside each module DllMain() function. This can be done using the following:

```
AC_IMPLEMENT_EXTENSION_MODULE(theArxDLL);
HINSTANCE _hdllInstance = NULL;
extern "C" int APIENTRY
```

```
  DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID
  lpReserved) {
  // Remove this if you use lpReserved
UNREFERENCED_PARAMETER(lpReserved);
if (dwReason == DLL_PROCESS_ATTACH) {
   theArxDLL.AttachInstance(hInstance);
   hdllInstance = hInstance;
}
else if (dwReason == DLL_PROCESS_DETACH) {
   theArxDLL.DetachInstance();
}
return 1;   // ok
}
```

The second class, called CAcModuleResourceOverride is a handy class that performs the switch from AutoCAD's resource to your application's resource when it is instantiated and then, when it is destroyed, switch back to AutoCAD. To use it, just declare an object of it before using your own resources. When it goes out of scope it will be destroyed and its destructor will switch back. So, this is the way to go:

```
CAcModuleResourceOverride res;
CMyDialo dlg;
```

```
dlg.DoModal();
```

This way you will successfully create your dialog and won't face strange errors like: "An unsupported operation was attempted". Another possible error happens with other dialog appearing instead of yours! Crazy hum? Remember that this applies to any type of resources including strings placed into string tables.

Using MFC Built-in support

Autodesk has also provided us some great classes to make our application looks like a native AutoCAD feature and work much more integrated with AutoCAD's interface. There are two collections of MFC built-in classes which are grouped by AcUi and AdUi prefixes. The AcUi prefixed classes are made to work inside AutoCAD environment and the AdUi classes are made to work outside AutoCAD but, due to license agreement, could be used only on applications that interact with AutoCAD.

Each of these two classes allow us to create buttons like those one we can see inside AutoCAD native dialogs, allow us to create smart edit controls modified to work with angles, points, texts, etc. There are also great combo boxes that allows us to create Layer like combos, linetype combos, color combos, etc.

These classes really provide you a professional appearance. There are also other advanced features like hack into AutoCAD Options dialog creating your own tab, create dockable dialogs, among many others.

Take a look inside ObjectARX documentation to see a complete list of those classes.

ObjectARX&Dummies 教程（十一）—— Custom ObjectARX Class

Class 11 - Custom ObjectARX Class

Introduction

In my personal opinion, the greatest feature of ObjectARX is the capability to develop your own objects and entities. This powerful feature will allow you to create complex applications and provide your users a unique experience using your software.

This feature is possible since ObjectARX beginning (officially at AutoCAD R13). Since Autodesk provided this feature, its own developers thought that they could develop their selves vertical solutions for the most interesting market areas. At that time, products like MAP, MCAD and ADT start to show up.

Today there are several vertical products based on AutoCAD made by Autodesk and much more developed by third-party companies.

How is this possible?

ObjectARX takes full advantage of C++ language features like inheritance, polymorphism and overriding among many others. This allows Autodesk to publish part of AutoCAD source code using a SDK like package with library and header files.

Beyond this point, ObjectARX exports some classes allowing you to derive from them and implement your own behavior taking advantage of all ready-to-use methods and overriding those ones you need.

This way, when your custom class is defined and implemented your application will be compiled and linked with AutoCAD native libraries and headers and will be possible to load your module (DLL) at runtime to use your own objects inside AutoCAD! Great hum?

When to use and when not to use custom objects

Even being a powerful feature of ObjectARX, custom objects are not the best solution for any kind of implementation. Sometimes is better to user another solution like XData or XRecords. This will totally depend on how much complex your application is and how much complex will be the way users will interact with your product.

You need to be sure when to use or not custom classes inside ObjectARX. Personally I perform some questions to myself that will help me to decide:

- My product's elements are simple or complex?
- My elements are only complex in terms of non-graphical data or they will require complex graphical representation?
- Do I need to protect my element's data when the drawing is out of my

company?

- Will my elements present a complex interaction with users very different from AutoCAD native entities?
- Do I need to share common information among my elements?

These questions will really help you to decide or not to user custom classes.

How to use a custom objects?

The first step is to choose your base class. This will depend on what type of custom object are you willing to implement. Basically you need to choose if it will represent an entity or a data object. If it will be an entity you will need to derive it from AcDbEntity or other of its derived classes. In other hand, if it will not have graphical appearance, you will derive it from AcDbObject. There are some ObjectARX classes that does not allow you to derive from. Take a look at ObjectARX documentation for a complete list.

It is very important to you clearly understand these differences between AcDbEntity and AcDbObject. Remember that every AcDbEntity if also an AcDbObject but NOT ALL AcDbObject is an AcDbEntity. This is because AcDbEntity is derived from AcDbObject.

I really would like you to walk through the ObjectARX class hierarchy to locate yourself in that tree and see clearly what are we talking about. There is a DWG file, called classmap.dwg, inside your ObjectARX SDK folder called \classmap.

Runtime Identification

AutoCAD requires that every custom class has its own runtime type identification. This will be used by AutoCAD and by your own application. Basically you don't need to care about this because there are MACROS to do this job for you. The AcRxObject class is the responsible to perform this feature and exactly due that it is on the top of AcRx tree.

The runtime identification is made by some functions like the following:

desc(), a static member function that returns the class descriptor object of a particular (known) class.

cast(), a static member function that returns an object of the specified type, or NULL if the object is not of the required class (or a derived class).

isKindOf() returns whether an object belongs to the specified class (or a derived class).

isA() returns the class descriptor object of an object whose class is unknown.

These functions are pretty useful because they help you to get runtime important information from AutoCAD native objects and your own objects. A good example if when you have a pointer to an AcDbEntity and would like to know if it is an AcDbCircle or an AcDbLine. How? Just use the above functions to get the information or even try to cast the pointer.

To declare these functions you will need to use the following MACRO inside your class declaration:

ACRX_DECLARE_MEMBERS(CLASS_NAME);

To implement these functions, you will need to use one of the following MACROS:

ACRX_NO_CONS_DEFINE_MEMBERS(CLASS_NAME,

PARENT_CLASS):

Use for abstract classes and any other classes that should not be instantiated.

ACRX_CONS_DEFINE_MEMBERS(CLASS_NAME,

PARENT_CLASS, VERNO):

Use for transient classes that can be instantiated but are not written to file.

ACRX_DXF_DEFINE_MEMBERS(CLASS_NAME,PARENT_CLASS, DWG_VERSION, MAINTENANCE_VERSION, PROXY_FLAGS, DXF_NAME, APP):

Use for classes that can be written to, or read from, DWG and DXF files.

Additionally, you will need to initialize and delete your custom class from ObjectARX runtime tree which can be done, during your application's kInitAppMsg and kUnloadAppMsg using the following functions implemented by the above MACROS:

```
// Inside kInitAppMsg function handler
MyClass::rxInit();
// Call this only once for all of your custom classes
acrxBuildClassHierarchy();

// Inside kUnloadAppMsg function handler
deleteAcRxClass(MyClass::desc());
```

This will guarantee that when your application is loaded AutoCAD recognizes your class and when it was not present (unloaded) AutoCAD will transform your class instances into Proxy entities. Proxy entities are a binary package that protects and preserves your custom object during DWG roundtrip without your application.

ObjectARX&Dummies 教程（十二）—— Deriving from AcDbObject

Class 12 - Deriving from AcDbObject

Introduction

I will start with Custom Objects and then proceed with custom entities on the next class. Custom objects can be used for several purposes and they are very powerful. Once your application creates and manage a custom object you will be able to construct complex application structures as well as much more intelligent and efficient data storage.

Starting with a simple example, suppose that you will need to build an ObjectARX application which implements some bars that has a length property and several types of shapes. Is possible that more than one bar has the same shape and it would be nice if you can provide a single instance of shape's information and share it among all bars using this shape.

The first impulse is to repeat the information on each bar no matter you will duplicate information. This works but will generate additional problems beyond the first problem which is the unnecessary space used to store the same information. Suppose that you need to update the shape and you would affect all bars that are using this shape. If your information is repeated in all bars you will need to open each bar and update its information. In other hand, if shape's information is stored into one single place and bars reflect this information you will need only to update this information in one place and all bars using this shape will be updated as soon as you update the shape's information.

AutoCAD use this technique on several features like layers, text styles, groups, etc. You will use exactly a custom object to store this information and share it, through its ObjectId, among all "clients" of this object.

How to begin

As discussed on previous class, you will need to derive from AcDbObject to be able to build your own custom object. This can be easily done using the ARXWizard or you can do by yourself creating the class by hand.

After create the class you need build some methods to create, store and acquire those objects instances from its container. Well, but where you should store your custom objects? AutoCAD provides a general purpose container called Named Object Dictionary (NOD). NOD is capable to store and persist any custom object derived from AcDbObject. It uses a dictionary like storage structure where you put a unique key (at the same level) and an object instance through its pointer and ObjectId. There are other custom object containers like Extension Dictionary that I will avoid due our course audience.

The NOD container could (and should) be organized by folders to make your dictionary as much organized as you can. The first node should be your application name to avoid conflict with other third-party ObjectARX applications that could use the NOD at the same time as you. The second level should contain all your business groups of objects. This will really depend on how many and the number of custom object types you have. NOD does not prohibit you to stored different classes at the same level but I really recommend you to avoid this except in case you need to stored generic objects together like on a Preferences group of objects.

You don't need to always Open and Close the NOD and go deep to find where are your desired object every time you need to access it. You can build some kind of cache of the most used Objects through its ObjectId and manage this cache to be updated for every single opened drawing. Remember that NOD is part of AcDbDatabase object and it is per document. So, you need to care about to build and fill your dictionary for every brand new drawing.

How to persist your custom objects

As I said before, the most used place to store custom objects is the NOD which is an AcDbDictionary. NOD takes care of its child objects because it is a container. So, when the AcDbDatabase object is issued to save its data by AutoCAD it also pass this message to its child objects and NOD is one of them. Once NOD receives this message it walks through its structure and call dwgOutFields() for every object stored there. The same process occurs when you open the drawing and the dwgInFields() is called by AcDbDatabase on NOD and consequently on its children. Exactly due that you will need to override the DWG filling methods to make possible to persist your custom objects among DWG open/close sessions.

Essential functions to override in your custom object class are:

```
virtual Acad::ErrorStatus dwgInFields(AcDbDwgFiler* filer);
virtual Acad::ErrorStatus dwgOutFields(AcDbDwgFiler* filer) const;
virtual Acad::ErrorStatus dxfInFields(AcDbDxfFiler* filer);
virtual Acad::ErrorStatus dxfOutFields(AcDbDxfFiler* filer) const;
```

If you don't plan to support DXF interface to your custom object you could avoid them.

Object's state management

On class 5 we have talked about object states when opening objects. Inside your custom object class you need to pay attention to call the proper assert method to make sure that all proper events and processes are fired when your object's state has changed. This is very important!

Those functions who change your object's data state must first call the assertWriteEnabled() function and then apply the required modifications. Functions who only read information from your object and does not affect its data state must call assertReadEnabled() function and also I really recommend that you make all these as const functions. This will avoid you to accidentally change the object's state when it is opened for read what will cause an assert error message. If you forget to call the proper assert method strange things may occur like call UNDO and your object stay unchanged and a lot other bizarre things.

How to create a custom object

To implement your custom object you will need to do the following:

1- Derive from AcDbObject;

2- Implement your data;

3- Implement access functions (read/write) with proper assert calls;

4- Implement the filling methods persisting and reading your data;

As a baseline, I will present a short example here:

```
// ------------------------------------------
// Class declaration
// ------------------------------------------
class MyClass : public AcDbObject {
public:
ACRX_DECLARE_MEMBERS(MyClass);
MyClass() {};
virtual ~MyClass() {};
Acad::ErrorStatus getVal (int& val) const;
Acad::ErrorStatus setVal (int val);
Acad::ErrorStatus getString (CString& str) const;
Acad::ErrorStatus setString (LPCTSTR str);

virtual Acad::ErrorStatus dwgInFields(AcDbDwgFiler*);
virtual Acad::ErrorStatus dwgOutFields(AcDbDwgFiler*) const;

private:
int m_Val;
CString m_Str;
};
// ------------------------------------------
// Class Definition
// ------------------------------------------
ACRX_DXF_DEFINE_MEMBERS(MyClass,
AcDbObject, AcDb::kDHL_CURRENT,
AcDb::kMReleaseCurrent, 0, MYCLASS, MYSAMP);
// ------------------------------------------
Acad::ErrorStatus MyClass::getVal (int& val) const {
    assertReadEnabled();
    val = m_Val;
    return Acad::eOk;
}
// ------------------------------------------
Acad::ErrorStatus MyClass::setVal (int val) {
    assertWriteEnabled();
    m_Val = val;
    return Acad::eOk;
}
// ------------------------------------------
Acad::ErrorStatus MyClass::getString (CString& str) const {
    assertReadEnabled();
    str.Format("%s",m_Str);
    return Acad::eOk;
}
```

```
// ------------------------------------------
Acad::ErrorStatus MyClass::setString (LPCTSTR str) {
    assertWriteEnabled();
    m_Str.Format("%s",str);
    return Acad::eOk;
}
// ------------------------------------------
Acad::ErrorStatus MyClass::dwgInFields(AcDbDwgFiler* pFiler) {
    assertWriteEnabled();
    AcDbObject::dwgInFields(pFiler);

    Adesk::Int16 _val = 0;
    pFiler->readInt16(&_val);
    m_Val = _val;
    char* _temp = NULL;
    pFiler->readString(&_temp);
    m_Str.Format("%s",_temp);
    acutDelString(_temp);

    return pFiler->filerStatus();
}
// ------------------------------------------
```

```
Acad::ErrorStatus MyClass::dwgOutFields(AcDbDwgFiler* pFiler) const
{
    assertReadEnabled();
    AcDbObject::dwgOutFields(pFiler);

    pFiler->writeInt16(m_Val);
    pFiler->writeString(static_cast<const char*>(m_Str));

    return pFiler->filerStatus();
}
// ------------------------------------------
// ------------------------------------------
// Entry Point
// ------------------------------------------
AcRx::AppRetCode    acrxEntryPoint(AcRx::AppMsgCode    msg,    void*
appId) {
    switch (msg) {
        case AcRx::kInitAppMsg:
            acrxDynamicLinker->unlockApplication(appId);
            acrxDynamicLinker->registerAppMDIAware(appId);
            MyClass::rxInit();
            acrxBuildClassHierarchy();
            break;
        case AcRx::kUnloadAppMsg:
            deleteAcRxClass(MyClass::desc());
            break;
}
    return AcRx::kRetOK;
}
```

How to create and store your custom object

The NOD container is based on AcDbDictionary class which has several methods to read, write and erase entries. Your application needs to take care of NOD entries and be responsible to create instances of your custom class and store these objects inside the NOD. Each object stored must have a key defined or a generic key using the star * as its name.

```
void createMyObjects() {
    AcDbDictionary *pNamedobj = NULL;
    acdbHostApplicationServices()->workingDatabase()->
    getNamedObjectsDictionary(pNamedobj, AcDb::kForWrite);
    AcDbDictionary *pDict = NULL;
    if (pNamedobj->getAt("MYDICT",(AcDbObject*&) pDict,
        AcDb::kForWrite) == Acad::eKeyNotFound) {
    pDict = new AcDbDictionary;
    AcDbObjectId DictId;
    pNamedobj->setAt("MYDICT", pDict, DictId);
}
    pNamedobj->close();
    if (pDict) {
        MyClass *pObj1 = new MyClass();
        pObj1->setVal(1);
        pObj1->setString("String1");
        MyClass *pObj2 = new MyClass();
        pObj2->setVal(2);
        pObj2->setString("String2");
        AcDbObjectId rId1, rId2;
        pDict->setAt("*M",pObj1, rId1);
        pDict->setAt("*M",pObj2, rId2);
        pObj1->close();
        pObj2->close();
        pDict->close();
}
}
```

How to verify if my objects are really stored inside NOD?

You will need to iterate the NOD entries to find your dictionary and then perform an iteration over its entries. The process should be something like this:

```
void listMyObjects() {
    AcDbDictionary *pNamedobj = NULL;
    acdbHostApplicationServices()->workingDatabase()
    ->getNamedObjectsDictionary(pNamedobj, AcDb::kForRead);
    AcDbDictionary *pDict = NULL;
    pNamedobj->getAt("MYDICT",
    (AcDbObject*&)pDict,AcDb::kForRead);
    pNamedobj->close();
    if (pDict == NULL) {
        acutPrintf("\nThe dictionary MYDICT does not exist. Please
        create it first!");
        return;
}
    AcDbDictionaryIterator* pDictIter= pDict->newIterator();
    MyClass *pMyClass;
    int _val;
    CString _str;
    for (; !pDictIter->done(); pDictIter->next()) {
        pMyClass = NULL;

        pDictIter-

        >getObject((AcDbObject*&)pMyClass,AcDb::kForRead);

    if (pMyClass != NULL) {

        pMyClass->getVal(_val);

        pMyClass->getString(_str);

        pMyClass->close();

        acutPrintf("\nMyClass: val=%d, str=%s",_val,_str);

}

}

    delete pDictIter;

    pDict->close();

}
```

Stay tuned for the next Lab which will require you to build a custom object. See you there!

ObjectARX&Dummies 教程（十三）—— Deriving from AcDbEntity

Class 13 - Deriving from AcDbEntity

Introduction

On previous class you saw that we can derive from AcDbObject to create powerful custom objects and store them inside NOD. On this class you will see that we can also create custom entities, deriving from AcDbEntity or one of its derived classes, which will present you a new way of thinking about how much powerful an ObjectARX application can be.

At the launch of AutoCAD 13 ObjectARX SDK opened a new world for developers and for Autodesk itself. Custom entities allows developers to build rich graphical applications and present several new features not included into native AutoCAD entities. Autodesk also start to develop other applications based on new entities that brought great functionalities and unique experience.

When you think about a custom entity you need to first think on how it will be used, handled, edited and all features it will present to users. This will allow you to outline the custom entity behavior and list all tasks it will need to support and perform. This step is very important to decide if is better to develop a custom entity or if is better to use a native AutoCAD entity adding some data using XData or XRecords.

In other hand, when you use standard AutoCAD entities you will need also to handle its behavior to support and manage all things users may perform with it. This is not quite simple and depending on how creative are your users it will take you much time. When you adopt the custom entity solution it will give you much advantage on handling user operations but it will be more complex to implement.

Advantages on using custom entities

Several advantages will point you to the custom entities approach. I would like to list some of these advantages just to allow you to perceive how powerful it is:

Custom graphics: When you create your own custom entity you are responsible for its graphics. ObjectARX provides you some primitive drawing functions that allow you to draw your custom entity using everything you need. No matter if your entity is too simple or too complex, ObjectARX provides tools to do that.

Custom Grips and OSNAPs: It is also up to you implement or not Grips and Object Snap functionalities to your custom entities. You may need some specific OSNAP points and advanced Grip functions, go ahead, use them!

Custom transformation: Your custom entity can be transformed using your own criteria. This takes full advantage of powerful AcGe library included into ObjectARX.

Embedded native entities: If you would like to build an entity that looks like some native entities you can embed those entities into your custom entity and take advantage of all already implemented code. Suppose you need to build a custom entity that is much like a Polyline with a hatch inside. This can be easily done by embedding an AcDbHatch entity inside your AcDbPolyline derived custom entity.

Custom entity graphics

The AcGi library provides all you need to construct your custom entity's graphics. Basically your custom entity will be presented on AutoCAD screen using one or both of the following methods:

```
virtual Adesk::Boolean
AcDbEntity::worldDraw (AcGiWorldDraw * pWd);
virtual void
AcDbEntity::viewportDraw (AcGiViewportDraw * pVd);
```

The first function, called worldDraw(), is responsible to draw standard graphics for your entity. The second function, called viewportDraw(), is optional and it allows you to draw viewport dependent graphics. These functions receive a drawing context pointer that will allow you to perform your drawing as well.

These functions are called several times due several reasons and you need to provide the code inside them as faster and efficient as you can. Don't perform heavy calculations, long loops and other time consuming tasks there. If you are using native entities to draw you custom entity does not declare them inside these functions. Declare them as members of your class and just forward the calls inside worldDraw() or viewportDraw() to these embedded entities.

AutoCAD performs the drawing process walking through all database entities and calling first the worldDraw() method. If the worldDraw() method returns Adesk::kFalse, AutoCAD also walks through each viewport and call entity's viewportDraw() method. Exactly at this point you may draw a different graphic depending on each viewport configuration.

The provided drawing primitives are quite simple and I will just list them here: Circle, Circular arc, Polyline, Polygon, Mesh, Shell, Text, Xline and Ray. Please refer to the AcGi documentation inside ObjectARX SDK for instructions and detailed information on how to use them. Primitives are called using a geometry() function.

Custom entities allows you to also subdivide them. This feature is done using the AcGiSubEntityTraits object. The AcGiSubEntityTraits object sets graphical attribute values using the following traits functions:

- Color
- Layer
- Linetype
- Polygon fill type
- Selection marker

This way you can separate logical information through its graphics. For instance, if you entity has some texts and some lines you may separate them into two different subentities. Further, if you would like to draw part of your entity using a specific color or linetype, this is also done using subentities. Each subentity could have its own mark. This will allow you do perform advanced interation with users by discovering on which subentity user has clicked. If you would like to do that, before draw your subentity graphics, give a call to setSelectionMarker() passing an incremental index.

Inside viewportDraw() function you will also have access to caller viewport information through AcGiViewport object inside the AcGiViewportDraw object passed in. The viewport geometry object provides the same primitives world geometry object plus the following polygon and polyline primitives, which use eye and display space coordinates: polylineEye, polygonEye, polylineDc and polygonDc. Some examples of both worldDraw() and viewportDraw() methods are presented below:

```
Adesk::Boolean MyEnt::worldDraw(AcGiWorldDraw *pW) {
    AcGePoint3d verts[5];
    // Create some random points
    verts[0] = verts[4] = AcGePoint3d(-0.5, -0.5, 0.0);
    verts[1] = AcGePoint3d( 0.5, -0.5, 0.0);
    verts[2] = AcGePoint3d( 0.5, 0.5, 0.0);
    verts[3] = AcGePoint3d(-0.5, 0.5, 0.0);
    // Set the subentity color as 3
    pW->subEntityTraits().setColor(3);
    // Draw the polyline primitive
    pW->geometry().polyline(5, verts);
    return Adesk::kTrue;
}

void MyEnt::viewportDraw(AcGiViewportDraw* pV){
    AcGePoint2d lleft, uright;
    // Get viewport's DC coordinates
    pV->viewport().getViewportDcCorners(lleft,uright);
    // Perform some math here
    double xsize = uright.x - lleft.x;
    double ysize = uright.y - lleft.y;
    xsize /= 10.0;
    ysize /= 10.0;
    double xcenter = uright.x - xsize;
    double ycenter = uright.y - ysize;
    double hxsize = xsize / 2.0;
    double hysize = ysize / 2.0;
    AcGePoint3d verts[5];
    // Set vertex initial value
    for (int i=0; i<5; i++) {
        verts.x = xcenter;
        verts.y = ycenter;
        verts.z = 0.0;
    }
    // Perform some adjustments
    verts[0].x -= hxsize;
    verts[0].y += hysize;
    verts[1].x += hxsize;
    verts[1].y += hysize;
    verts[2].x += hxsize;
    verts[2].y -= hysize;
    verts[3].x -= hxsize;
```

```
    verts[3].y -= hysize;
    verts[4] = verts[0];

    // Set the subentity color as 3
    pV->subEntityTraits().setColor(3);
    // Draw the polyline on DC context
    pV->geometry().polylineDc(5, verts);
}
```

Implementing Object Snap (OSNAP)

Your custom entity probably will need to provide some precision points through Object Snap feature. Depending on how much complex your custom entity is you will need to implement several OSNAP points and several types like EndPoint, Center, etc. To add OSNAP features to your custom entity you will need to add the following method to your class (there are other signatures):

```
virtual Acad::ErrorStatus AcDbEntity::getOsnapPoints(
AcDb::OsnapMode osnapMode,
int gsSelectionMark,
const AcGePoint3d& pickPoint,
const AcGePoint3d& lastPoint,
const AcGeMatrix3d& viewXform,
AcGePoint3dArray& snapPoints,
```

AcDbIntArray& geomIds) const;

This function will allow you to fill the passed in AcGePoint3dArray with those points that match with the provided osnapMode. Possible values to osnapMode are:

AcDb::kOsModeEnd: endpoint on the entity that is nearest to the pickPoint.

AcDb::kOsModeMid: midpoint (of any line, arc, etc., subentity) that is nearest to the pickPoint.

AcDb::kOsModeCen: center point (of any circle or arc subentity) that is nearest to the pickPoint.

AcDb::kOsModeNode: node point that is nearest to the pickPoint.

AcDb::kOsModeQuad: quad point that's nearest to pickPoint.

AcDb::kOsModeIns: insertion point of the entity (the insertion of a BlockReference or an MText object).

AcDb::kOsModePerp: intersection point of the entity and a line perpendicular to it passing through lastPoint.

AcDb::kOsModeTan: point on the entity where a line passing through lastPoint will be tangent to the entity

AcDb::kOsModeNear: Find the point on the entity that's nearest to pickPoint.

Imagine your custom entity is a rectangle and the user is running a LINE command and hover your entity with the EndPoint OSNAP enabled. Your entity will need to respond AutoCAD providing those points that could be used as EndPoints of your entity. In this case, inside your getOsnapPoints() function, you will need to fill the AcGePoint3dArray with the points of the rectangle corner. AutoCAD choose which of these points are inside the aperture box and the closest point to the cursor. So, your function will be something like:

```
Acad::ErrorStatus MyEnt::getOsnapPoints(
    AcDb::OsnapMode osnapMode,
    int gsSelectionMark,
    const AcGePoint3d& pickPoint,
    const AcGePoint3d& lastPoint,
    const AcGeMatrix3d& viewXform,
    AcGePoint3dArray& snapPoints,
    AcDbIntArray& geomIds) const {
        assertReadEnabled();
        switch (osnapMode) {
            case AcDb::kOsModeEnd:
                snapPoints.append(pt[0]);
                snapPoints.append(pt[1]);
                snapPoints.append(pt[2]);
                snapPoints.append(pt[3]);
                break;
        }
        return Acad::eOk;
}
```

The intersection OSNAP is not implemented through getOsnapPoints() method. As it is much more complex there is a special function called intersectWith() to handle that. I won't present details about this here but you may read and find further information inside ObjectARX SDK documentation.

Implementing GRIP and Stretch points

Grip points provides a great and simple way to user edit and transform entities. You will probably want to implement this feature for your entity. Further, stretch points will allow users to stretch your entity as well. These two features are very simple to implement and make your entity much more flexible and powerful.

Basically you just need to inform which are your key points to Grip and stretch. Other functions will be responsible on set the behavior of your entity when each Grip and stretch point are used. Depending on the complexity of your entity these functions may become a little bit complex.

For the Grip points feature you will need to implement a couple of functions. The first function, called getGripPoints(), will return those points where you would like to enable a Grip. The second function, called moveGripPointsAt() will perform the action when Grips are fired:

```
virtual Acad::ErrorStatus
AcDbEntity::getGripPoints (AcGePoint3dArray& gripPoints,
AcDbIntArray& osnapModes,AcDbIntArray& geomIds) const;

virtual Acad::ErrorStatus
AcDbEntity::moveGripPointsAt (const AcDbIntArray& indices,
const AcGeVector3d& offset);
```

Remember that Grip points don't need to be created only where there is a part of your graphics. You can, for instance, create a Grip at the center of a rectangle and there is nothing drawn there.

The getGripPoints() function receive 3 arguments. Currently only the first argument is used. It is an array of 3D points passed in by AutoCAD. This array contains all points involved into the current Grip operation. As other entities may be involved into this operation this array may already be filled. You just will append your desired points to this array. The points you have appended inside getGripPoints() will be identified by an index from the order they were appended to the point array.

The moveGripPointsAt() function will receive the index array and a 3D vector sent by AutoCAD with the current transformation

(AcGeVector3d) being applied. At this time you just need to loop the index array, get each of its values (the index) and, depending on this value fire the transformation at the desired point. Imagine again your entity is a rectangle and you have 5 Grip points, one of each corner and one at the center. For each corner you will apply only the transformation to its grip and for the center grip you will apply the transformation to all of your points. The corner grip operation will result into a stretch at that corner and the center grip operation will result into a move of the whole entity. To apply the transformation to each point just call the transformBy() method passing in the AcGeVector3d received from AutoCAD.

In other hand, stretch points are defined and controlled by another two functions. They are much like the Grip functions and sometimes you just return a call to the Grip functions inside respective Stretch functions:

```
virtual Acad::ErrorStatus
AcDbEntity::getStretchPoints(
AcGePoint3dArray& stretchPoints) const;


virtual Acad::ErrorStatus
AcDbEntity::moveStretchPointsAt(
const AcDbIntArray& indices, const AcGeVector3d& offset);
```

The behavior of stretch functions are almost the same as the Grip functions and if your stretch points will behavior like Grip points you may just forward the call as below:

```
Acad::ErrorStatus MyEnt::getStretchPoints(
AcGePoint3dArray& stretchPoints) const {
    AcDbIntArray osnapModes,geomIds;
    return MyEnt::getGripPoints(stretchPoints,osnapModes,geomIds);
}
Acad::ErrorStatus MyEnt::moveStretchPointsAt(
const AcDbIntArray& indices, const AcGeVector3d& offset) {
    return MyEnt::moveGripPointsAt(indices,offset);
}
```

The same concept of worldDraw() and viewportDraw() functions also applies here to the moveGripPointsAt() and moveStretchPointsAt() functions. They are called several times and they need to be as faster as you can. When you click at an entity's grip the moveGripPointsAt() function is called for every small mouse movement.

Implementing Transformation

Your custom entity needs to support transformation if you would like to allow users to perform commands like MOVE, ROTATE and SCALE. This feature is implemented through transformBy() method which receives a transformation matrix representing the current transformation being applied to your entity. Inside this function you will apply this matrix to your entity's data to reflect the modifications. The AcGeMatrix3d class support all types of transformations and encapsulate them through a matrix:

```
virtual Acad::ErrorStatus
AcDbEntity::transformBy(const AcGeMatrix3d& xform);
```

A typical implementation of transformBy() function could be:

```
Acad::ErrorStatus MyEnt::transformBy(
const AcGeMatrix3d& xform) {
    pt[0].transformBy(xform);
    pt[1].transformBy(xform);
    pt[2].transformBy(xform);
    pt[3].transformBy(xform);
}
```

In some special cases that I won't present here, you may need to apply the transformation to a clone or copy of your original entity. This is done using the getTransformedCopy() method which receives the transformation matrix and a pointer to be filled with the entity's transformed copy.

Too much information? Next class I will present a short and practical example with a custom entity. Stay tuned!

ObjectARX&Dummies 教程（十四）—— Creating a Custom Entity (step by step)

Class 14 - Creating a Custom Entity (step by step)

Hello

On this class I will complete the Class 13 with a tutorial to show you how to create a simple custom entity step by step. This entity has some features that will help you to see what is possible to do using ObjectARX. Please read carefully the following instructions.

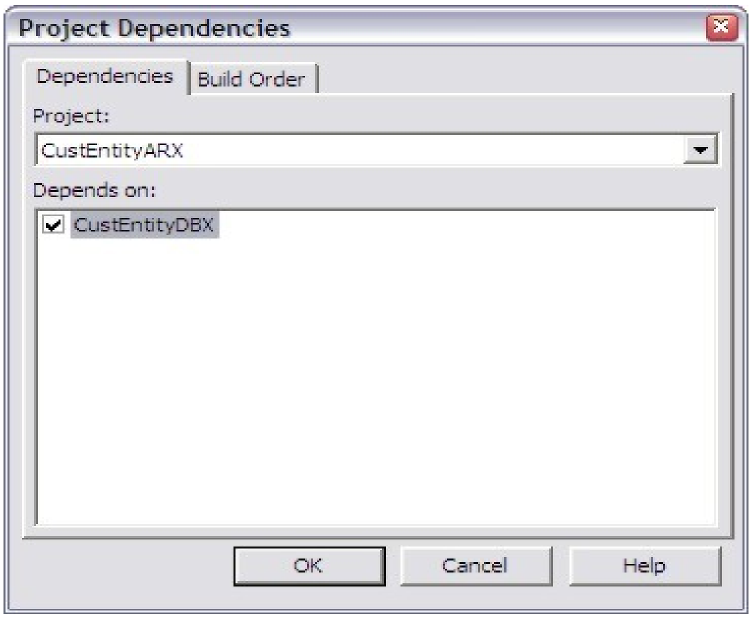
-Create a Blank Solution called CustomEntitySample;

-Add a new DBX project called CustEntityDBX;

-Add a new ARX project called CustEntityARX;

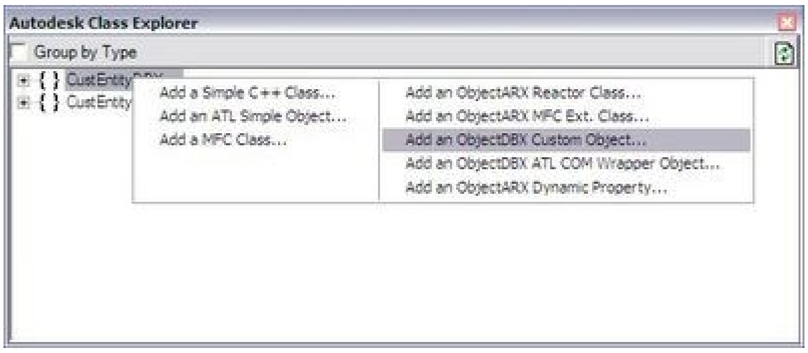
-Remember to enable MFC Extension on both projects;

-Create a dependency from ARX project to DBX:



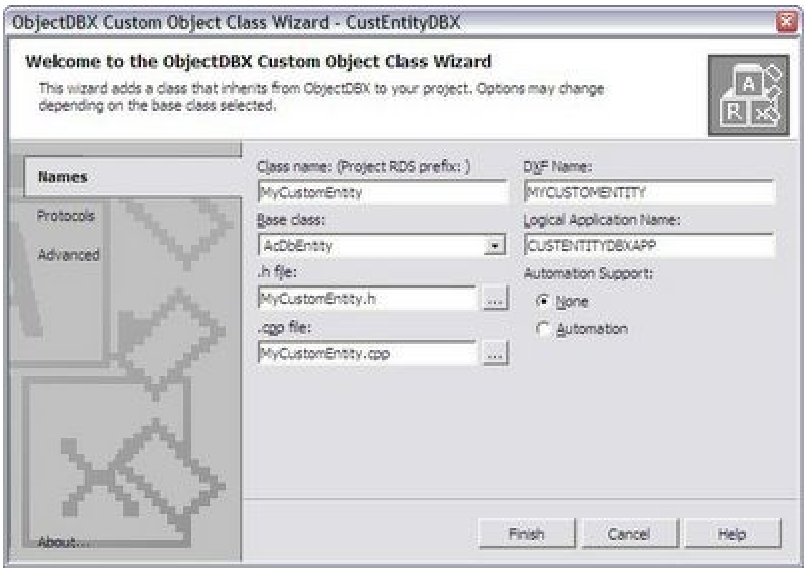
-Go to DBX project and open Autodesk Class Explorer;

-Right click on CustEntityDBX node and select "Add ObjectDBX Custom Object...":


Autodesk Class Explorer
Group by Type
CustEntityDBX
CustEntity
Add a Simple C++ Class...
Add an ATL Simple Object...
Add a MFC Class...
Add an ObjectARX Reactor Class...
Add an ObjectARX MFC Ext. Class...
Add an ObjectDBX Custom Object...
Add an ObjectDBX ATL COM Wrapper Object...
Add an ObjectARX Dynamic Property...

-Choose MyCustomEntity as class name and derive it from AcDbEntity;

-Other fields will be filled automatically (you may change these values);



ObjectDBX Custom Object Class Wizard - CustEntityDBX

Welcome to the ObjectDBX Custom Object Class Wizard

This wizard adds a class that inherits from ObjectDBX to your project. Options may change depending on the base class selected.

Class name: (Project RDS prefix:) MyCustomEntity

DXF Name: MYCUSTOMENTITY

Base class: AcDbEntity

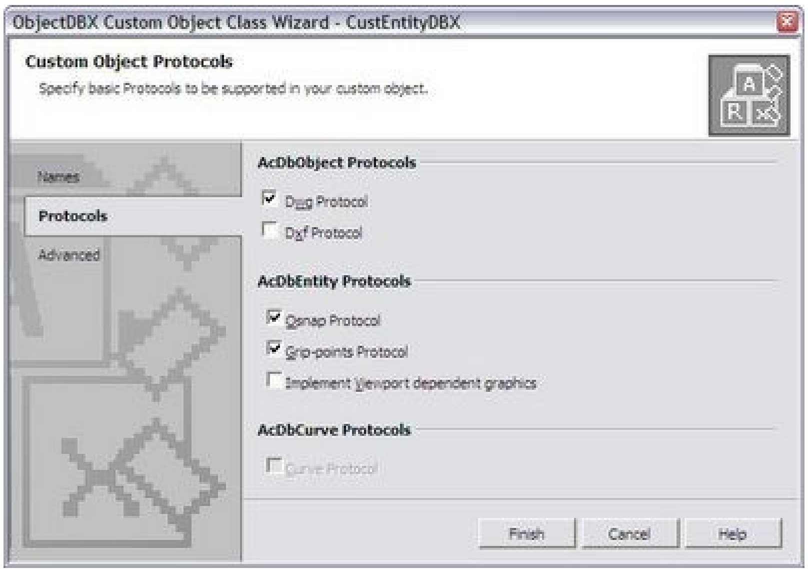
Logical Application Name: CUSTENTITYDBXAPP

.h file: MyCustomEntity.h

Automation Support: [✓ None] Automation

.cpp file: MyCustomEntity.cpp

-Go to the Protocols TAB and enable both Osnap and Grip-points protocols:



ObjectDBX Custom Object Class Wizard - CustEntityDBX

Custom Object Protocols

Specify basic Protocols to be supported in your custom object.

AcDbObject Protocols

- ✓ Dwg Protocol
- ☐ Dxf Protocol

AcDbEntity Protocols

- ✓ Osnap Protocol
- ✓ Grip-points Protocol
- ☐ Implement Viewport dependent graphics

AcDbCurve Protocols

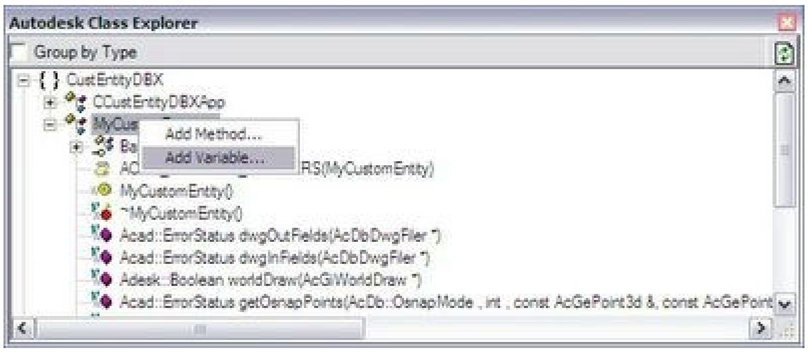
- ☐ Curve Protocol

-Rebuild your project to see if everything is ok;

-Now, open again the Autodesk Class Explorer, go to CustEntityDBX project;

-Select the MyCustomEntity node (if it is not there, click on Refresh icon);

-Right click on it and select "Add Variable...":


Autodesk Class Explorer

Group by Type

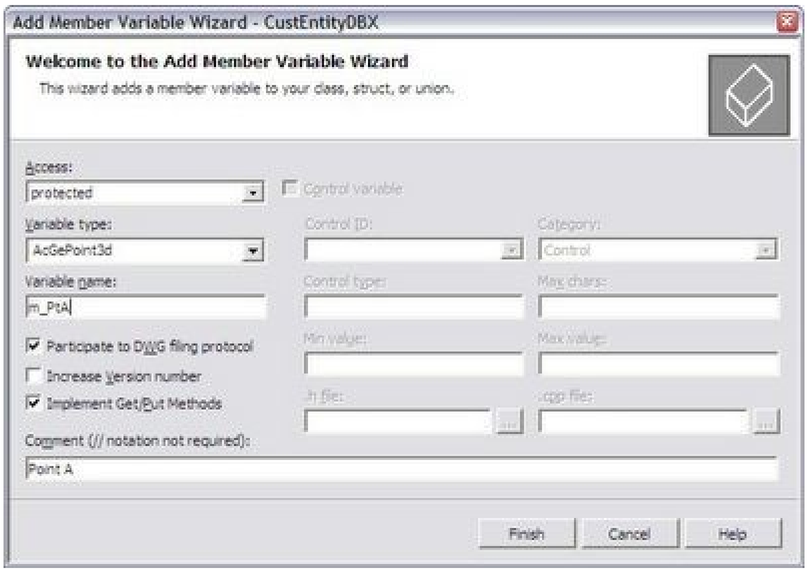
```
{ } CustEntityDBX
    CCustEntityDBXApp
    MyCus
        Ba
        AC                    RS(MyCustomEntity)
        MyCustomEntity()
        ~MyCustomEntity()
        Acad::ErrorStatus dwgOutFields(AcDbDwgFiler *)
        Acad::ErrorStatus dwgInFields(AcDbDwgFiler *)
        Adesk::Boolean worldDraw(AcGiWorldDraw *)
        Acad::ErrorStatus getOsnapPoints(AcDb::OsnapMode , int , const AcGePoint3d &, const AcGePoint
```

-Name it as "m_PtA", type will be an AcGePoint3d and its access will be protected;

-Enable "Participate to DWG filing protocol", disable "Increase Version number";

-Enable "Implement Get/Put methods" and fill desired comments for it;

-Repeat the process and create : "m_PtB", "m_PtAB" and "m_PtBA":


Add Member Variable Wizard - CustEntityDBX

Welcome to the Add Member Variable Wizard

This wizard adds a member variable to your class, struct, or union.

Access: protected

Control variable [ ]

Variable type: AcGePoint3d

Control ID:

Category: Control

Variable name: m_PtA

Control type:

Max chars:

[✓] Participate to DWG filing protocol

[ ] Increase Version number

[✓] Implement Get/Put Methods

Min value:

Max value:

.h file:

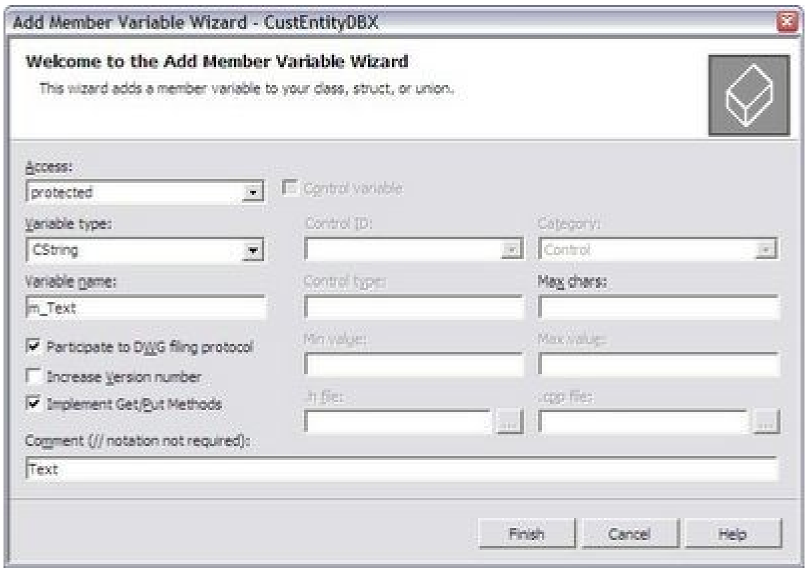
.cpp file:

Comment (// notation not required): Point A

-Now, add a CString variable called "m_Text";

-Remember to correct the CString manipulation and filing as we have done in Lab3;

-Now you should compile your code ok;


Add Member Variable Wizard - CustEntityDBX

Welcome to the Add Member Variable Wizard

This wizard adds a member variable to your class, struct, or union.

Access: protected

[ ] Control variable

Variable type: CString

Control ID:

Category: Control

Variable name: m_Text

Control type:

Max chars:

[✓] Participate to DWG filing protocol

[ ] Increase Version number

[✓] Implement Get/Put Methods

Min value:

Max value:

.h file:

.cpp file:

Comment (// notation not required): Text

Before continue, we will need to remove some functions that we will not use. First remove the following from .H file:

```
virtual Acad::ErrorStatus getGripPoints (
AcDbGripDataPtrArray &grips, const double curViewUnitSize, const int gripSize,
const AcGeVector3d &curViewDir, const int bitflags) const;

virtual Acad::ErrorStatus moveGripPointsAt (
const AcDbVoidPtrArray &gripAppData,
const AcGeVector3d &offset, const int bitflags);
```

There are 4 methods getOsnapPoints(). We will only use the following method:

```
virtual Acad::ErrorStatus getOsnapPoints (
AcDb::OsnapMode osnapMode, int gsSelectionMark, const
AcGePoint3d &pickPoint,
const AcGePoint3d &lastPoint, const AcGeMatrix3d &viewXform,
AcGePoint3dArray &snapPoints, AcDbIntArray &geomIds) const ;
```

Now, you will need to delete the body of these functions from cpp file.

WORLDDRAW

-To draw our custom entity, we will need to add the following code to worldDraw() method:

```
assertReadEnabled();
// Bounding Polyline
AcGePoint3d pts[4];
pts[0] = m_PtA;
pts[1] = m_PtAB;
pts[2] = m_PtB;
pts[3] = m_PtBA;
mode->subEntityTraits().setSelectionMarker(1); // Mark 1
mode->subEntityTraits().setColor(1); // Red
mode->geometry().polygon(4,pts);
// Entity's Text
mode->subEntityTraits().setSelectionMarker(2); // Mark 2
```

```
    mode->subEntityTraits().setColor(256); // ByLayer
    AcGiTextStyle style;
    style.setFileName("txt.shx");
    style.setBigFontFileName("");
    style.setTextSize(25);
    style.loadStyleRec();
    AcGePoint3d txtPt((m_PtB.x+m_PtA.x)/2.0,(m_PtB.y+m_PtA.y)/2.0,
    m_PtA.z);

    mode->geometry().text(txtPt, AcGeVector3d::kZAxis,
    (m_PtAB-m_PtA),m_Text,m_Text.GetLength(),Adesk::kFalse, style);

    return Adesk::kTrue;
```

GRIP Points

-Open remaining getGripPoints() method inside MyCustomEntity implementation (cpp);

-We would like to enable 5 grips for this entity. One at each point (A,B,AB and BA) plus one at the center;

-Inside the getGripPoints(), add the following code:

```
gripPoints.append(m_PtA);
gripPoints.append(m_PtAB);
gripPoints.append(m_PtB);
gripPoints.append(m_PtBA);
gripPoints.append(AcGePoint3d((m_PtB.x+m_PtA.x)/2.0,
(m_PtB.y+m_PtA.y)/2.0,m_PtA.z));
```

Now, when user clicks on each Grip we would like to perform an action. This is done using the moveGripPointsAt() method;

We need to check the index of each fired grip accordingly to the getGripPoints() method and then apply the transformation;

Inside remaining moveGripPointsAt() method, add the following code:

```
assertWriteEnabled();
for(int i=0; i<indices.length(); i++) {
    int idx = indices.at(i);
    // For A and center point
    if (idx==0 idx==4) m_PtA += offset;
    // For AB and center point
    if (idx==1 idx==4) m_PtAB += offset;
    // For B and center point
    if (idx==2 idx==4) m_PtB += offset;
```

```
        // For BA and center point
        if (idx==3 idx==4) m_PtBA += offset;
    }
    return (Acad::eOk);
```

OSNAP Points

Open the getOsnapPoints() method. We will add 3 Osnap modes: EndPoint, MidPoint and Center:

```
    assertReadEnabled();
    switch (osnapMode) {
        case AcDb::kOsModeEnd:
            snapPoints.append(m_PtA);
            snapPoints.append(m_PtAB);
            snapPoints.append(m_PtB);
            snapPoints.append(m_PtBA);
            break;
        case AcDb::kOsModeMid:
            snapPoints.append(m_PtA+((m_PtAB-
    m_PtA).length()/2.0)*((m_PtAB-m_PtA).normalize()));
            snapPoints.append(m_PtAB+((m_PtB-
    m_PtAB).length()/2.0)*((m_PtB-m_PtAB).normalize()));
            snapPoints.append(m_PtB+((m_PtBA-
    m_PtB).length()/2.0)*((m_PtBA-m_PtB).normalize()));
            snapPoints.append(m_PtBA+((m_PtA-
    m_PtBA).length()/2.0)*((m_PtA-m_PtBA).normalize()));
            break;
    case AcDb::kOsModeCen:
            snapPoints.append(AcGePoint3d((m_PtB.x+m_PtA.x)/2.0,
        (m_PtB.y+m_PtA.y)/2.0, m_PtA.z));
            break;
    }
    return (Acad::eOk);
```

TRANSFORMATION

The custom entity transformation is done through transformBy() method;

Open the Autodesk Class Explorer, expand the MyCustomEntity node, expand Base Classes node and then expand AcDbEntity node;

-Scroll down the list and select:

Acad::ErrorStatus transformBy(const AcGeMatrix3d &);

-Right click on this method and select "Implement Base Class Method";

-This function logic is very simple, just add the following code:

```
    assertWriteEnabled();
    m_PtA.transformBy(xform);
    m_PtAB.transformBy(xform);
    m_PtB.transformBy(xform);
    m_PtBA.transformBy(xform);
    return (Acad::eOk);
```

-This function will allow the custom entity to be transformed;

ARX PROJECT

-First we will need to add a new command called "MyCustEnt";

-This will create a method inside acxrEntryPoint.cpp file;

-Now, add the necessary include instruction after the #include "StdAfx.h":

```
#include "..\CustEntityDBX\MyCustomEntity.h"
```

-Inside the method we will create our entity:

```
    // Input information
    ads_point pt1,pt2;
    if (acedGetPoint(NULL,"Set the first point:\n",pt1) != RTNORM)
        return;
    if (acedGetCorner(pt1,"Set the second point:\n",pt2) != RTNORM)
        return;
    char buffer[512];
    if (acedGetString(0,"Enter the text:\n",buffer) != RTNORM)
        return;
    // Setup entity
    MyCustomEntity *pEnt = new MyCustomEntity();
    pEnt->put_m_PtA(asPnt3d(pt1));
    pEnt->put_m_PtAB(AcGePoint3d(pt2[X],pt1[Y],pt1[Z]));
    pEnt->put_m_PtB(asPnt3d(pt2));
    pEnt->put_m_PtBA(AcGePoint3d(pt1[X],pt2[Y],pt2[Z]));
    pEnt->put_m_Text(buffer);
    // Post to Database
    AcDbBlockTable *pBlockTable;
    acdbHostApplicationServices()->workingDatabase()-
    >getSymbolTable(pBlockTable,
    AcDb::kForRead);
    AcDbBlockTableRecord *pBlockTableRecord;
    pBlockTable->getAt(ACDB_MODEL_SPACE,
    pBlockTableRecord,AcDb::kForWrite);
pBlockTable->close();
    AcDbObjectId retId = AcDbObjectId::kNull;
    pBlockTableRecord->appendAcDbEntity(retId, pEnt);
    pBlockTableRecord->close();
    pEnt->close();
```

TEST

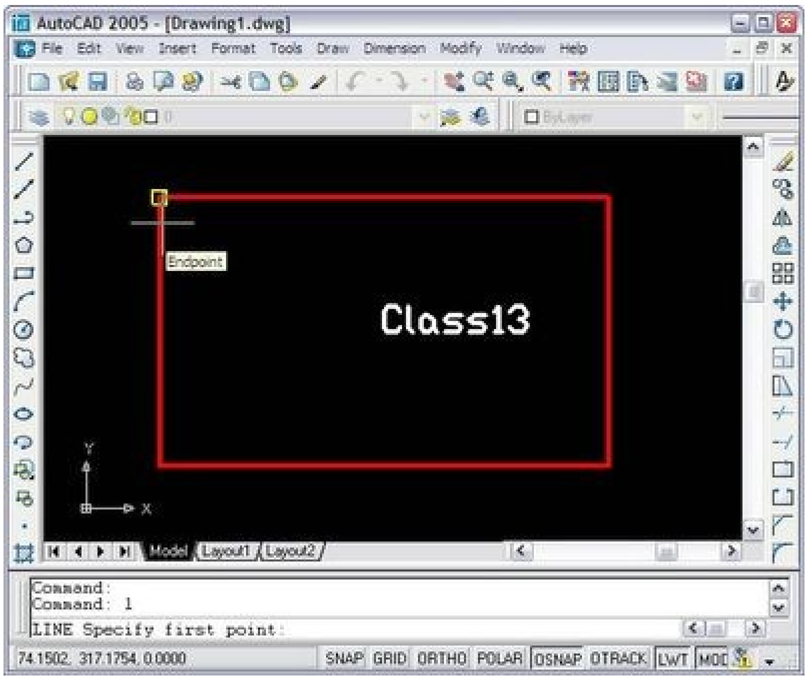
-Rebuild your project;

-Open AutoCAD and load DBX module first and then load the ARX;

-Fire command MYCUSTENT and create as many entities as you want;

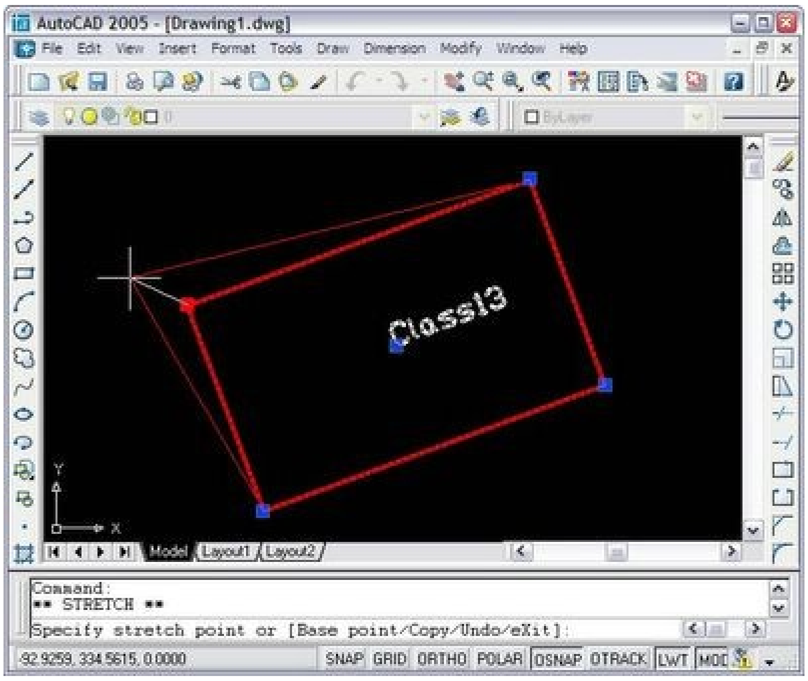
-Test it against GRIP edit, MOVE, ROTATE, SCALE;

-Fire a LINE command and try to get precision points (EndPoint, MidPoint and Center) over your entities;


AutoCAD 2005 - [Drawing1.dwg]
File Edit View Insert Format Tools Draw Dimension Modify Window Help
ByLayer
Endpoint
Class13
Y
X
Model Layout1 Layout2
SNAP GRID ORTHO POLAR OSNAP OTRACK LWT MODEL

```
Command:
Command: 1
LINE Specify first point:
```

74.1502, 317.1754, 0.0000


AutoCAD 2005 - [Drawing1.dwg]
File Edit View Insert Format Tools Draw Dimension Modify Window Help
ByLayer
Class13
Model
Layout1
Layout2
SNAP GRID ORTHO POLAR OSNAP OTRACK LWT MODEL

```
Command:
** STRETCH **
Specify stretch point or [Base point/Copy/Undo/eXit]:
```

-92.9259, 334.5615, 0.0000