SD468553

A Practical Sample of Building a Custom User Interactive AutoCAD Command

Norman Yuan

Independent IT Consultant

Learning Objectives

- Create custom command
- Analyze information/data needed for the command execution and possible user interaction
- Design the code structure

Description

Writing code to define a custom command is as easy as decorating class/method with *[CommandClass]/[CommandMethod]* attributes. But making the command user-friendly and error-proof need to take into many factors into account while actually implementing the process of the command execution, which may warrant far more code that originally expected. This instruction demo shows the process of developing a custom “FILLET” command in detail with AutoCAD .NET API code.

Speaker(s)

Norman Yuan started using AutoCAD in his work as civil engineer in early 1990s and soon realized how much a few lines of AutoLisp code could make AutoCAD so much easier to use and make CAD much more efficient. That was when his programming life began: AutoLisp/ADE -> AutoCAD VBA -> generic enterprise system development, which covers all kinds of programming aspects. He has worked in engineering companies of all sizes, from a score of employees to over tens of thousands. He is currently working as independent IT/CAD technology consultant, based in Edmonton, AB, Canada. He has 25+ years of experience of CAD programming, not only with plain AutoCAD, but also with AutoCAD verticals, such as Map 3D, Civil 3D, Plant 3D, and other Autodesk Applications, such as Navisworks, Revit.

Contact:

Email: norman.yuan@live.com
Blog: Drive AutoCAD With Code

Contents

Create Custom Command

With AutoCAD .NET API, creating a custom command is as easy as decorate a public class as [CommandClass] and decorate a public method, which does not return anything and does not accept argument, with [CommandMethod].

A typical .NET API class that defines custom command(s) would look like:

```
using Autodesk.AutoCAD.Runtime;
using CadApp = Autodesk.AutoCAD.ApplicationServices.Application;

[assembly: CommandClass(typeof(NormCommandsForAU2020.Au2020Commands))]
[assembly: ExtensionApplication(typeof(NormCommandsForAU2020.Au2020Commands))]

namespace NormCommandsForAU2020
{
    public class Au2020Commands : IExtensionApplication
    {
        #region IExtensionApplication implementing

        public void Initialize()
        {
            var dwg = CadApp.DocumentManager.MdiActiveDocument;
            var ed = dwg.Editor;

            try
            {
                ed.WriteMessage(
                    $"\nInitializing custom add-in \"{this.GetType().Name}\"...");
                // To DO here
                ed.WriteMessage($"\nIntializing done.\n");
            }
            catch (System.Exception ex)
            {
                ed.WriteMessage($"\nInitializing error:\n{ex.Message}\n");
            }
        }

        public void Terminate()
        {

        }

        #endregion

        [CommandMethod("Hello")]
        public static void RunMyCommand()
        {
            var dwg = CadApp.DocumentManager.MdiActiveDocument;
            var ed = dwg.Editor;
            ed.WriteMessage($"\nHello world!\n");
        }
    }
}
```

Something to Know about Creating Custom Command

When creating custom command, there are something one should know

Specifying CommandClass and ExtensionApplication

A class can be explicitly specified as CommandClass and/or ExtensionApplication, as shown below:

```
[assembly: CommandClass(typeof(NormCommandsForAU2020.Au2020Commands))]
[assembly: ExtensionApplication(typeof(NormCommandsForAU2020.Au2020Commands))]
```

However, these 2 lines of code are not mandatory. When this statement is not used, AutoCAD would search all classes in the loaded .NET assembly to find CommandMethod; or run the Initialize() method if the implemented IExtensionApplication when the assembly is loaded. If it is used, AutoCAD only search the classes specified as CommandClassm and/or ExtensionApplication. If multiple CommandMethods and/or ExtensionApplication are defined in multiple classes, but some of the class are specified as CommandClass and/or ExtensionApplication, while other classes are not, then the CommandMethod and/or ExtensionApplication in those other classes would be ignored by AutoCAD.

Implementing IExtensionApplication

Often when creating custom command, or in general, when running .NET code, some kind of initializing work is needed to make sure custom commands execute properly. Thus, it is common that the CommandClass also implements IExtensionApplication, so that code runs when the assembly is loaded into AutoCAD. There are somethings to take into account:

- When implementing IExtensionApplication.Initialize(), make sure to wrap code in *try…catch…* block. If code execution in the Initialize() method raises exception that is not handled, the assembly loading will be aborted (thus the code in this assembly will not available for execution). In general, the code should be like:

  ```
  public void Initialize()
  {
      var dwg = CadApp.DocumentManager.MdiActiveDocument;
      var ed = dwg.Editor;

      try
      {
          ed.WriteMessage(
              $"\nInitializing custom add-in \"{this.GetType().Name}\"...");
          // To DO whatever is needed here
          ed.WriteMessage($"\nIntializing done.\n");
      }
      catch (System.Exception ex)
      {
          ed.WriteMessage($"\nInitializing error:\n{ex.Message}\n");
      }
  ```

```
}
```

- In most cases it is not necessary to place any code in IExtensionApplication.Terminate(), unless the ExtensionApplication class has to clean up resources used outside AutoCAD, such as an open database connection (which is bad practice, though, if the code runs inside AutoCAD maintain an open connection to external database).
- One needs to be careful as what kind of work to be done in the IExtensionApplication.Initialize(), especially if the work may take long time to run, which would increase AutoCAD startup time, to user, the prolonged startup time may look like AutoCAD being frozen/hang. Also, depending on how the ExtensionApplication class is loaded (manually with "NETLOAD" command, or with some sort of auto-load mechanism, such as "Auto-loader", or Acad.lsp), if the code in Initialize() may rely on other AutoCAD's resources being available. For example, if you want to load your own Ribbon items in Initialize(), AutoCAD may not have its ribbon menu fully loaded when your DLL is loaded and the Initialize() executed. One may consider in the Initailize() method to hook up an even handler to Application_Idle event, and have the custom initialization work done when AutoCAD finish its startup and becomes idle.

Organizing Custom Command Code and Process Execution Code

While all the working code can be contained in a single class that is specified as CommandClass and possibly a single DLL project, it would much better to place all custom commands in a CommandClass of DLL project, and all the custom command functionalities in separated different projects. That is, the CommandClass project has references to all other projects. This way, only one assembly is to be loaded into AutoCAD explicitly, which would make the auto-loading configuration much easier: there is only one DLL file to be loaded, as long as other referenced DLL files in placed in the same location.

CommandMethod Being Static or Not

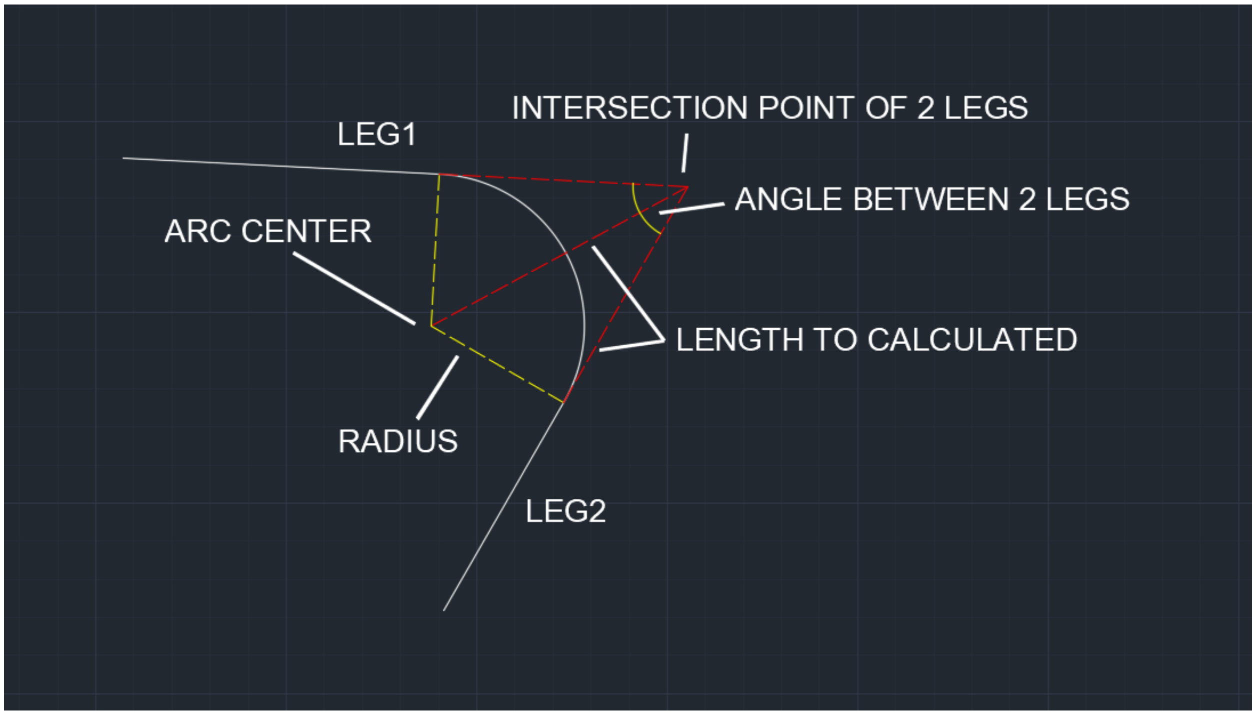
CommandMethod being static or not results in different way for AutocCAD to create the CommandClass instance when the custom command defined by CommandMethod is executed. I had an old article on this topic:

https://drive-cad-with-code.blogspot.com/2010/09/command-method-static-or-not-static.html

Design A Custom “Fillet” Command

AutoCAD comes command “Fillet”. It has been improved in later AutoCAD versions by showing the effect of the command executing before it actually finished: after selecting the first fillet leg, the fillet result would show when the mouse cursor hovers on second fillet leg.

Form user’s point of view, running command “FILLET” requires user to select 2 straight segments of line/polyline that do not parallel to each other. Once 2 legs are selected, as long as the radius of the fillet operation is known, AutoCAD should be able to calculate where the fillet arc meets the 2 legs. See the picture below:



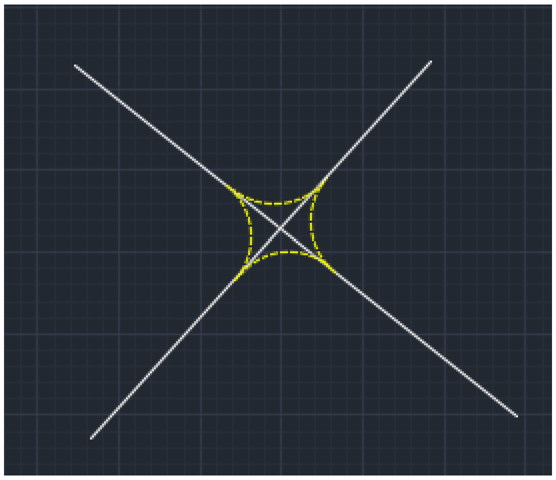
So, if we program our own custom FILLET command, our code needs to calculate where the arc’s center point is and the arc’s start angle and end angle, so that we could programmatically draw the fillet arc. Looking at the picture, one might think it would be quite easy to do the calculation: based on the angle formed by the 2 legs and the arc’s radius, we can calculate the points where the arc meets the 2 legs and the center point of the arc.

However, in reality, there are more factors that should be take into account.

Some Factors to Take into Account

1. The angle of the 2 legs (lines or straight segments of polylines) could be counted differently, depending on start point and end point of the 2 legs.

2. The drawing's unit setting could count the angle clockwise or counter-clockwise, or even the start direction of 0 degree could be different.
3. If the 2 legs are crossing each other, as shown in picture below, which side of the legs to be filleted.



4. The 2 legs could be different types of entity (i.e. Line or Polyline). The combinations include: 2 Lines; 1 Line and 1 Polyline, 2 Polylines, or 2 segments of the same Polyline.

Define Scope of Operation Functionality of the Custom FILLET Command

The custom FILLET command could be the exact replica of built-in FILLET command. But for the purpose of this class, I want to make it slightly different from the built-in one with 2 options: fillet arc's radius being static or dynamic. Static means that user enter a radius, the command show the fillet with an arc of that radius, while dynamic means the fillet arc's radius changes while user drag the mouse cursor. With this in mind, following conditions are applied to make the custom command a bit easier to build (it is certainly possible to enhance the custom command to remove some of the limiting conditions):

1. If a leg is polyline, only straight segment is allowed to be picked as fillet leg.
2. If the 2 legs are 2 entities and one of them is, or both of them are Polyline, then only the start segment or end segment of the Polyline is allowed to be picked.
3. If the 2 legs are from the same polyline, the 2 legs must be the connected segments.
4. The 2 legs must not parallel to each other. Therefore, the 2 legs' intersecting angle must be less than 180˚
5. During the command execution, set start angle (0˚) to point to east, and angle direction to counter-clockwise;
6. During the command execution, create Line entities as visual graphics to be presented as 2 legs. The 2 Lines all starts at the intersecting point of the 2 legs; the intersecting angle of the 2 legs always starts from leg 1 and rotates counter-clockwise to leg 2.
7. If 2 legs are different entities and one of it is a Polyline, the command joins the 2 entities into single Polyline

Implement the Custom Command in Code

From above analysis, the workflow of this custom FILLET command should be:

1. User starts the command;
2. Select the first leg;
3. Select the second leg;
4. Instead of select leg, user can choose keyword “Radius” to enter radius of the fillet arc;
5. Once 2 valid legs are selected, a temporary fillet arc is drawn to show how the fillet would be look like;
6. user press Enter to accept the fillet arc, command completes;
7. or user can choose dragging the mouse to dynamically change the fillet arc’s radius, and click the mouse to confirm the dragged arc, command complete.

With this workflow, the code could be divided into 3 functioning portions

1. Code to guide user to select 2 legs. The code should properly validate the picked entity as required (class *FilletTargetSelector*);
2. Code to generate auxiliary/temporary entities (the 2 legs, the arc, and the center line that divides the legs’ intersection angle evenly). These temporary entities are used as visual aid to show how the fillet arc looks like before the actual fillet is done (class *FilletGhosts*);
3. Code to actually complete the fillet operation - adding arc, trimming the 2 legs as needed, joining involved legs as Polyline (class *FilletTool*)

Class *FilletTargetSelector*

In this class the code execution guides user to pick the 2 legs to be filleted. It will validate the picked entities. Besides the very basic requirement of the legs being either Line or Polyline, the other validation rules occur when the second leg is selected, including:

- If the 2 legs are in parallel, exception will be raised and will be caught in calling class *FilletTool*;
- If the leg is polyline, the picked segment must be straight one;
- If one leg is a polyline, or both legs are polylines, the picked point must be on the start or end segment of the polyline;
- If the 2 legs on the same polyline, the 2 legs must be connected to each other.

After 2 valid legs are selected, an instance of *FilletGhosts* class is generated. Because the main purpose of this class is to generate required data for actually doing the fillet work, and the data involves creating non-database-residing entities (ghost graphics), it implements *IDisposable* interface, so its instance can be wrapped in *using(…){…}* block to make it easier to dispose those ghost entities.

This class also creates an instance of class *AngleSettings*, which saves current drawing’s “ANGBASE” and “ANGDIR” system variables and then set them to required value, described previously (0˚ starts in X direction and angle is counted counter-clockwise). The saved angle settings are restored back when this class is disposed (at the end of *using(…){…}* block).

Class *FilletGhosts*

This class contains 4 non-database-residing drawables (2 Lines as legs, 1 Arc and 1 Line as center line dividing the intersecting angle of the 2 legs, on which the arc's center point locates). These drawables are used to generate TransientGraphics to show temporary graphics of the fillet arc/legs/center line for user to preview the command's outcome. Obviously, as long as the 2 legs are selected properly, the fillet arc's center point will be located on the calculated center line according to the radius of the arc. Because our custom Fillet command expect to allow user to drag mouse to dynamically change the fillet arc during preview, this class also handles Editor.PointMonitor event to draw the changed fillet arc during preview. Finally, since the temporary ghost drawables are non-database-residing, this class implements IDisposable interface, to make disposing them easier by calling its Dispose() method.

Class *FilletTool*

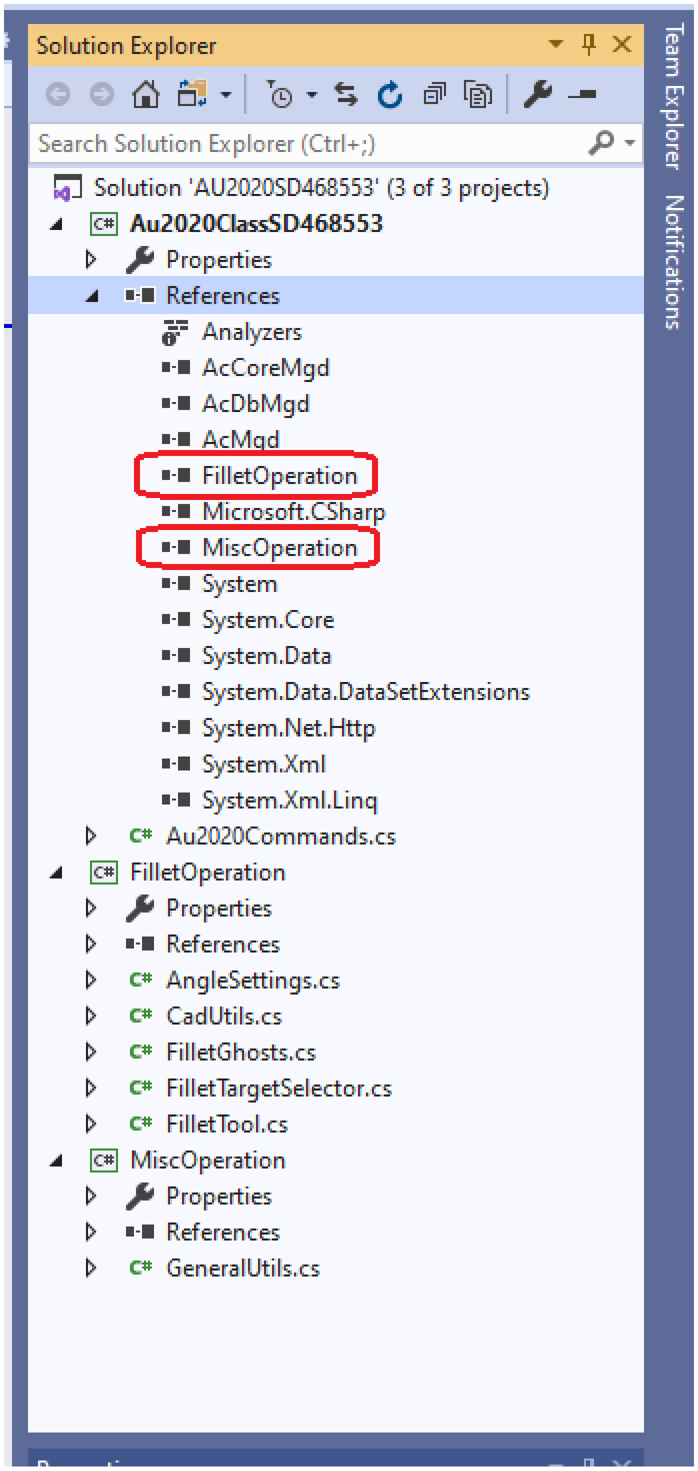
This class does the actual work:

- Create instance of FilletTargetSelector to let user select fillet legs;
- Draw fillet arc and update the 2 legs that connect to the arc;

This class is used in the separated custom command aggregation project.

Visual Studio Projects of This AU2020 Class

A Visual Studio Solution of 3 DLL projects is provided for this class:



```
Solution 'AU2020SD468553' (3 of 3 projects)
◢ Au2020ClassSD468553
    ▷ Properties
    ◢ References
        Analyzers
        AcCoreMgd
        AcDbMgd
        AcMgd
        FilletOperation
        Microsoft.CSharp
        MiscOperation
        System
        System.Core
        System.Data
        System.Data.DataSetExtensions
        System.Net.Http
        System.Xml
        System.Xml.Linq
    ▷ Au2020Commands.cs
◢ FilletOperation
    ▷ Properties
    ▷ References
    ▷ AngleSettings.cs
    ▷ CadUtils.cs
    ▷ FilletGhosts.cs
    ▷ FilletTargetSelector.cs
    ▷ FilletTool.cs
◢ MiscOperation
    ▷ Properties
    ▷ References
    ▷ GeneralUtils.cs
```

All projects are set to target .NET Framework 4.8 and compiled against AutoCAD2021 .NET assemblies. The code should work with older AutoCAD versions as long as the AutoCAD .NET assemblies (and the .NET Framework version, if necessary) corresponding to AutoCAD vesion is referenced.

Project *Au2020ClassSD468533*

This is the main project that aggregates all commands and is the one to be explicitly loaded into AutoCAD so that the custom commands can be executed by user. It has references to other 2 projects so that the custom commands can be built on the business operations programmed in that 2 projects

Project *FilletOperation*

This project contains all the code that implements the custom fillet operation logcs

Project *MiscOperation*

This project is meant to demonstrate how to separate business operation code into different projects and then to be referenced in command project for command building.