



Creating Geometric and Dimensional Constraints Using the AutoCAD® .NET API

Stephen Preston – Autodesk, Inc.

CP316-5

In AutoCAD 2010, we introduced the Parametric Drawing feature that allows geometric and dimensional constraints to be added to AutoCAD drawing elements. AutoCAD 2011 introduces a .NET API to allow you to add, edit, and query these constraints from your add-in applications. In this class, we will explain the underlying architecture of this feature and demonstrate the API using some simple examples.

About the Speaker:

Stephen has been a member of the Autodesk Developer Technical Services (DevTech) team since 2000, first as a support engineer and now as manager of the EMEA (Europe, Middle East, and Africa) team. In those roles, his responsibilities included supporting the AutoCAD APIs, including ObjectARX and AutoCAD .NET, as well as AutoCAD OEM and RealDWG™ technologies. Currently, he manages the Developer Technical Services Team in the Americas and serves as Workgroup Lead, working closely with the AutoCAD engineering team on future improvements in the AutoCAD APIs.

Stephen started his career as a scientist, and has a Ph.D. in Atomic and Laser Physics from the University of Oxford.

stephen.preston@autodesk.com

Introduction

The Parametric Drawing feature was introduced in AutoCAD 2010 as the first feature to make use of the underlying Associative Framework¹ architected by Jiri Kripac, Senior Architect on the AutoCAD software development team. AutoCAD 2010 included an ObjectARX (C++) Parametric Drawing API. A .NET API was added in AutoCAD 2011², and is a thin wrapper on top of the C++ API.

The Parametric Drawing feature allows you to apply geometric and dimensional constraints to entities in an AutoCAD drawing:

- **Geometric constraints** allow you to define rules that govern the geometric relationships between entities in your drawing – for example, these two circles must always be concentric, or the end points of these two lines must always be coincident.
- **Dimensional constraints** allow you to extend the behavior of standard AutoCAD dimensions (linear dimensions, angular dimensions, etc.) to allow them to control the dimension they represent and not just report it – for example, a normal radial dimension will change its value to reflect the radius of a circle as the user edits the circle; a radial dimensional constraint will not allow the user to edit the circle's radius, except by editing the dimension. Dimensional constraints can also be dependent on other dimensions in the drawing, so (for example) a circle's radius can be set to always be the same as the length of a particular line. Extend the line, and the circle's radius changes accordingly.

In this handout we'll explain the basic architecture for the Parametric Drawing feature, describe the constraints you have available to you, and walk through some simple code examples.

All samples in the handout and in the class are in VB.NET. You can easily translate to C# using an online translator. I've found <http://www.developerfusion.com/tools/convert/vb-to-csharp/> to be very effective. The samples were created using Visual Studio 2008 Professional, but you can use any version of Visual Studio 2008 or 2010 – including the Express editions.

This handout (and the class) assumes that you are familiar with the .NET Framework and the AutoCAD .NET API; and that you understand how to create, build and run an AutoCAD .NET addin. If you're not familiar with these, then please work through some of the resources listed in the [Further Reading](#) section before the class.

¹ See the ObjectARX Developers Guide section 'Advanced Topics->Associative Framework' for a detailed conceptual overview of the framework.

² AutoCAD 2011 also included a second feature based on the Associative Framework – the Associative Surface API. See Philippe Leefsma's "AutoCAD® .NET: Exploring the Associative Surfaces API" Product Clinic (CP232-1C) at AU Virtual 2010 to find out more about the Associative Surfaces API.

Overview of the Parametric Drawing Architecture

The Parametric Drawing feature is implemented using the Associative Framework architecture. The same architecture is used for the Associative Surfaces feature. The building blocks of the Associative Framework (and therefore the Parametric Drawing feature) are very simple – you have Actions and Dependencies:

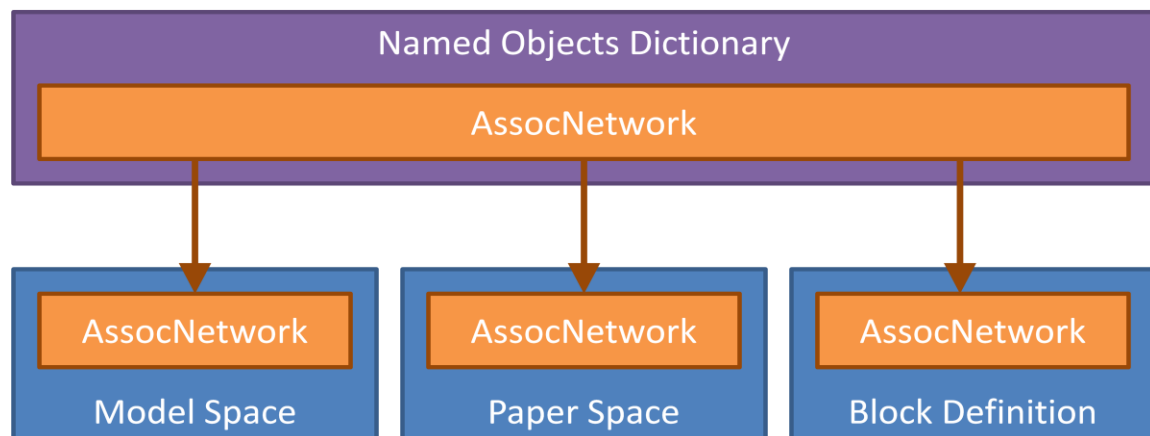
- **Actions** take some inputs, perform some calculations, and then create some output. The inputs are usually the properties of objects in the drawing. The outputs are usually to change the properties of objects in the drawing. An Action is represented in the API by the `AssocAction` class.
- **Dependencies** simply define the links between the Actions and the objects in the drawing the actions depend on. Dependencies are represented by the `AssocDependency` and `AssocDependencyBody` classes.

Keep these two definitions in mind when you're using the API, and everything else will fall into place for you.

In the Parametric Drawing API, we have some specific classes to consider³:

AssocNetwork

The `AssocNetwork` class is a type of (is derived from) `AssocAction`. The `AssocNetwork` contains all the `AssocActions` that make up the Parametric Drawing API. There is a master `AssocNetwork` in the Named Objects Dictionary (in a sub-dictionary named `ACAD_ASSOCNETWORK`). This master `AssocNetwork` references child `AssocNetwork`s – one for each `BlockTableRecord` you've added geometric or dimensional constraints to. The `AssocNetwork` for a `BlockTableRecord` is stored in the `BlockTableRecord`'s extension dictionary in a sub dictionary named `ACAD_ASSOCNETWORK`.

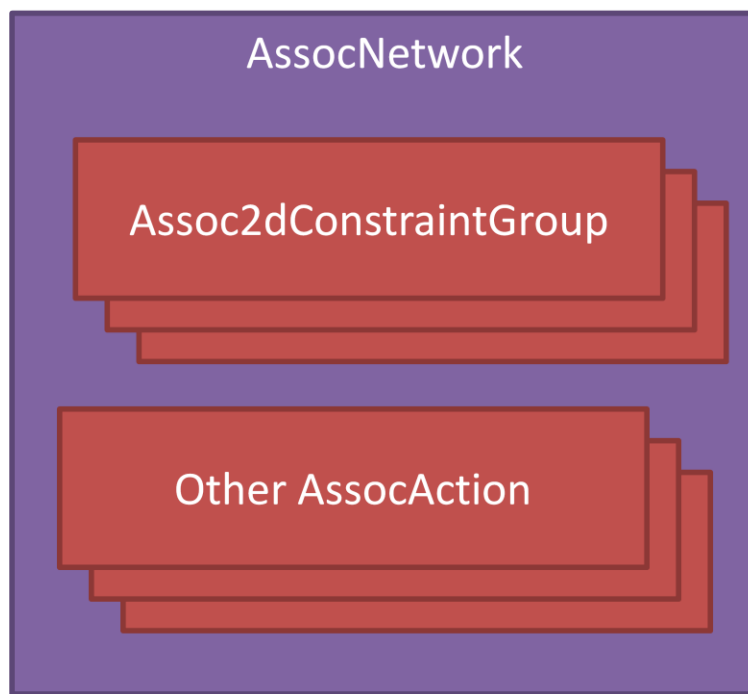


³ These classes are very well documented in the ObjectARX Reference Guide – both the unmanaged and managed references.

The AssocNetwork is not specific to the Parametric Drawing API. It holds the actions and dependencies for all Associative Framework features –that’s Parametric Drawing and Associative Surfaces. in AutoCAD 2011

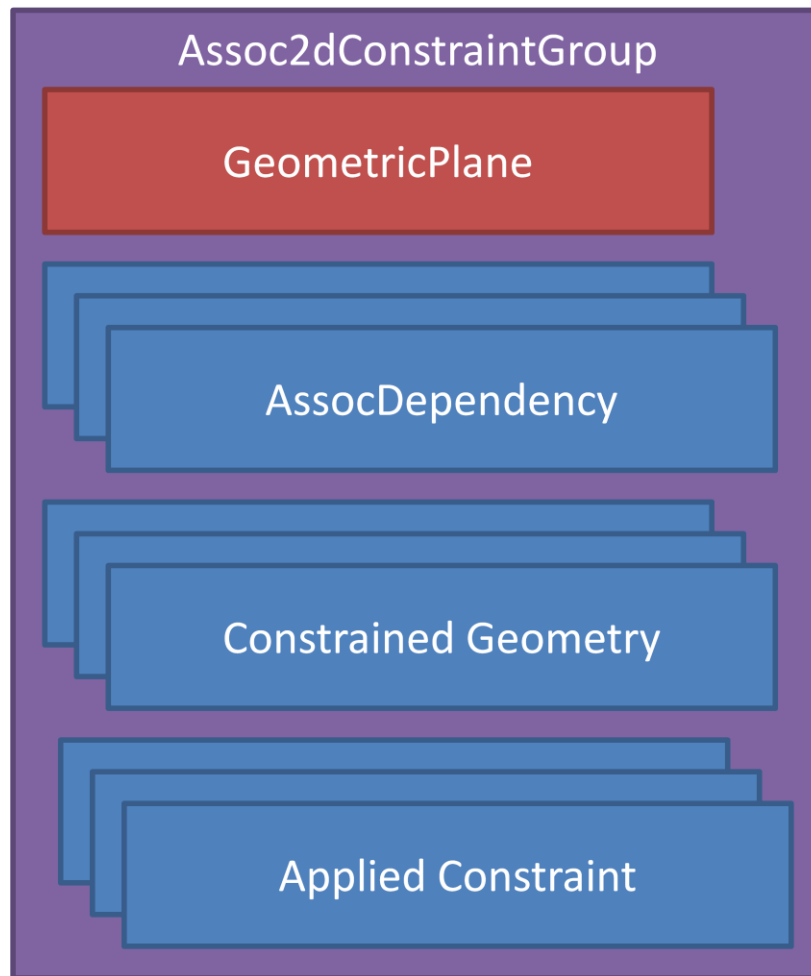
Assoc2dConstraintGroup

The Assoc2dConstraintGroup class is also a type of AssocAction. Assoc2dConstraintGroups are stored in the BlockTableRecord’s AssocNetwork, along with all the other AssocActions. Because the Parametric Drawing feature is for two-dimensional geometry only, a new Assoc2dConstraintGroup is created to store the constraints that apply to a specific geometric plane. This means that you may find several Assoc2dConstraintGroups within a single AssocNetwork, and you have to query each Assoc2dConstraintGroup to determine the plane it is defined for⁴.



Each Assoc2dConstraintGroup stores a list of AssocDependencies (the links to all geometry being constrained on its geometric plane, and a list of the constraints being applied to that geometry. The Assoc2dConstraintGroup won’t allow you to add geometry that is not on the correct plane.

⁴ See the section [Code Walkthrough – Creating a Constraint Group](#) for a code sample showing how to walkthrough the actions in an AssocNetwork to find all the Assoc2dConstraintGroups, and how to query the plane the constraint group is defined for.



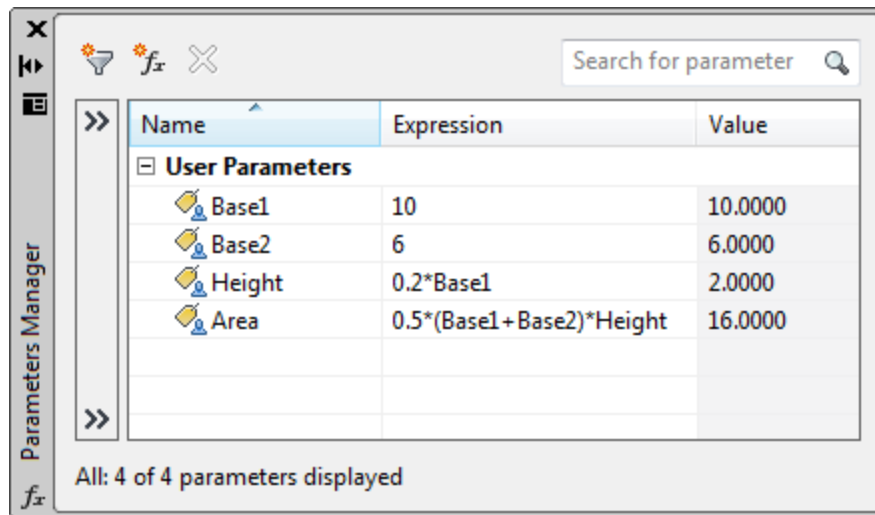
The most important functions you'll use in this class are the 'add' functions:

- `AddConstrainedGeometry`
- `Add3PointAngularConstraint`
- `AddAngularConstraint`
- `AddDistanceConstraint`
- `AddGeometricConstraint`
- `AddRadiusDiameterConstraint`

AssocVariable

An `AssocVariable` is another type of `AssocAction`. It stores User Parameters - the variables you see when you open the Parameter Manager. Each `AssocVariable` has a name, an expression (which can reference other `AssocVariables` by name), and a value (calculated from the name).

For example, I could have four `AssocVariables` defined like this:



AssocDependency

An AssocDependency represents the dependency (or link) we mentioned earlier between an action and a database resident object⁵. There are three main types of AssocDependency in the Parametric Drawing API:

AssocGeomDependency

The AssocGeomDependency class stores the dependency of an action on specific subentities (i.e. the edges and vertices) of a geometric entity. You don't have to deal with these when adding geometric constraints.

AssocValueDependency

The AssocValueDependency class stores the dependency of an action on a value (the value of an AssocVariable). You will use these when you create dimensional constraints.

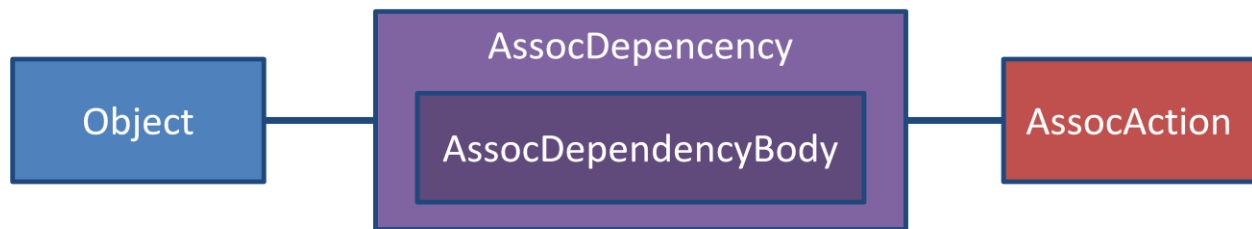
AssocDimDependencyBody

The AssocGeomDependency and AssocValueDependency classes are concrete classes derived from AssocDependency. Dimensional dependencies are implemented slightly differently. If (in ObjectARX) you want to create a custom dependency, you do so by deriving from AcDbAssocDependencyBody⁶. The custom AcDbAssocDependencyBody class is owned by an AcDbAssocDependency class, which refers calls to the 'body' class. This is how dimensional dependencies are implemented - as an AssocDimDependencyBody in .NET.

You'll use this class when you create a dimensional dependency. It links the action to the (database resident) Dimension entity.

⁵ Which may be an AssocAction.

⁶ The API won't allow you to derive from AcDbAssocDependency.



So now you understand the basic architecture of the Parametric Drawing API, let's take a look at what it can actually do.

Available Constraints

The API provides access to all the constraints you can add through the user interface. These are documented in the ObjectARX helpfiles, but I'm reproducing that information here to bring it all together in one place.

Geometric Constraints

You can create any of the following constraints using the `AddGeometricalConstraint` method of the `Assoc2dConstraintGroup` and the **`GeometricalConstraint.ConstraintType`** enum. The enum allows you to define the following constraint types (the names are self-explanatory):

Horizontal - Can be applied to a line or two points.

Vertical - Can be applied to a line or two points.

Parallel - Can be applied between two lines.

Perpendicular - Can be applied between two lines.

Normal - Currently can only be applied between a line and circle (or arc).

Collinear - Can be applied between two lines.

Coincident - Can be applied between two points, a point and a curve.

Concentric - Can be applied between any two of circles, ellipses, arcs or bounded ellipses.

Tangent - Can be applied between two of curves (except two lines). This constraint is not applicable for closed splines; for bounded splines, the tangent point can only be the start or end point which is coincident with start or end point of the other bounded curve.

EqualRadius - Can be applied between any two of circles or arcs.

EqualLength - Can be applied between two bounded lines (not rays).

Symmetric - Can be applied between two same type geometries. Circle and arc are considered to be the same type of geometry. The symmetry axis is a line.

Smooth (G2Smooth constraint) - Can be applied between a bounded spline and a bounded curve (including spline).

Fix - Can be applied on any supported geometry. [That's normally fixing an edge, or fixing an edge at its vertex].

The Parametric_API_Demo_VB sample demonstrates adding all these geometric constraint types.

Dimensional Constraints

Assoc2dConstraintGroup has four methods for adding dimensional constraints, each with an associated enum:

- Add3PointAngularConstraint and AddAngularConstraint use the AngularConstraint.AngularSectorType enum.
- AddDistanceConstraint uses the DistanceConstraint.DistanceDirectionType enum.
- AddRadiusDiameterConstraint uses the RadiusDiameterConstraint.RadDiaConstrType enum.

Here are the values those enums can take:

AngularConstraint.AngularSectorType enum

ParallelAntiClockwise - The angle measured from the forward direction of line 1 to the forward direction of line 2 anticlockwise.

AntiParallelClockwise - The angle measured from the forward direction of line 1 to the non forward direction of line 2 clockwise.

ParallelClockwise - The angle measured from the forward direction of line 1 to the forward direction of line 2 clockwise.

AntiParallelAntiClockwise - The angle measured from the forward direction of line 1 to the non forward direction of line 2 anticlockwise.

DistanceConstraint.DistanceDirectionType enum

NotDirected - Not directed distance. The minimum distance between the two geometries is measured.

FixedDirection - Directed distance with fixed direction. The distance between the two geometries is measured along the fixed direction.

PerpendicularToLine - Directed distance with relative direction. The distance between the two geometries is measured along the direction which is perpendicular to an existing constraint line.

ParallelToLine - Directed distance with relative direction. The distance between the two geometries is measured along the direction which is parallel to an existing constraint line.

RadiusDiameterConstraint.RadDiaConstrType enum

CircleRadius - The radius of a constrained circle or arc is measured.

CircleDiameter - The diameter of a constrained circle or arc is measured.

MinorRadius - The minor radius of a constrained (bounded) ellipse is measured.

MajorRadius - The major radius of a constrained (bounded) ellipse is measured.

The Parametric_API_Demo_VB sample project demonstrates all four dimensional constraint methods, but doesn't cover every enum value.

Code Walkthrough – Adding a Fixed Constraint

So now you understand the basic structure of the Parametric Drawing API and the range of constraints you have available, let's take a look at the code you need to write to create some of these constraints. The Parametric API is a relatively low level API. This gives you a lot of power and flexibility, but it also means that even adding the simplest constraint requires quite a lot of code. Fortunately, once you've worked out how to add one constraint type, it's pretty easy to work out how to add the others.

In this first example, we're simply going to add a 'Fix' constraint to a line's start point. I'll list the code in full first and then explain what's happening step-by-step after wards. Here's the code⁷:

```
<CommandMethod("TESTFIXED")> _
Public Shared Sub testFixedCommand()
    Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
    Dim tm As Autodesk.AutoCAD.DatabaseServices.TransactionManager =
db.TransactionManager
    Using myT As Transaction = tm.StartTransaction()
```

⁷ This code is part of the Parametric_API_Demo_VB sample. The same sample contains examples of adding many other constraint types. Also, check the sample for the latest version of this code – if the code differs, the sample should be the correct version.

```

'Create DB resident line
Dim bt As BlockTable = DirectCast(myT.GetObject(db.BlockTableId,
OpenMode.ForRead, False), BlockTable)
Dim btr As BlockTableRecord =
DirectCast(myT.GetObject(bt(BlockTableRecord.ModelSpace), OpenMode.ForWrite, False),
BlockTableRecord)
Dim entity As Entity = New Line(New Point3d(12, 5, 0), New Point3d(15, 12, 0))
btr.AppendEntity(entity)
myT.AddNewlyCreatedDBObject(entity, True)

'To query the subentities of the line, we create and use a protocol extension
(PE) provided by the associativity API
Dim subentityIdPE As AssocPersSubentityIdPE
Dim peCls As RXClass =
AssocPersSubentityIdPE.GetClass(GetType(AssocPersSubentityIdPE))
Dim pSubentityIdPE As IntPtr = entity.QueryX(peCls)
If pSubentityIdPE = IntPtr.Zero Then
    System.Windows.MessageBox.Show("cannot get pSubentityIdPE")
    Exit Sub
End If
subentityIdPE = TryCast(AssocPersSubentityIdPE.Create(pSubentityIdPE, False),
AssocPersSubentityIdPE)
If subentityIdPE Is Nothing Then
    System.Windows.MessageBox.Show("cannot get subentityIdPE")
    Exit Sub
End If
'Now we have the PE, we query the subentities
Dim edgeSubentityIds As SubentityId() = Nothing
'First we retrieve a list of all edges (a line has one edge)
edgeSubentityIds = subentityIdPE.GetAllSubentities(entity, SubentityType.Edge)
'Now we retrieve the vertices associated with the first edge in our array.
'In this case we have one edge, and the edge has three vertices - start, end
and middle.
Dim startSID As SubentityId = SubentityId.Null, endSID As SubentityId =
SubentityId.Null
Dim other As SubentityId() = Nothing
subentityIdPE.GetEdgeVertexSubentities(entity, edgeSubentityIds(0), startSID,
endSID, other)
'The PE returns a SubEntId. We want a FullSubentityPath
Dim subentPathEdge As FullSubentityPath, subentPath1 As FullSubentityPath
subentPathEdge = New FullSubentityPath(New ObjectId(0) {entity.ObjectId},
edgeSubentityIds(0)) 'The line edge
subentPath1 = New FullSubentityPath(New ObjectId(0) {entity.ObjectId},
startSID) 'The edge's startpoint.

'We call a helper function to retrieve or create a constraints group
Dim consGrpId As ObjectId = GetConstraintGroup(True)
Using constGrp As Assoc2dConstraintGroup = DirectCast(myT.GetObject(consGrpId,
OpenMode.ForWrite, False), Assoc2dConstraintGroup)
    'Pass in geometry to constrain (the line edge)
    Dim consGeom As ConstrainedGeometry =
constGrp.AddConstrainedGeometry(subentPathEdge)
    'Now create the constraint, a Fixed constraint applied to the line's
startpoint.
    Dim paths As FullSubentityPath() = New FullSubentityPath(0) {subentPath1}
    Dim newConstraint As GeometricalConstraint =
constGrp.AddGeometricalConstraint(GeometricalConstraint.ConstraintType.Fix, paths)
End Using
myT.Commit()
End Using

```

```
End Sub
```

The first part of the code is simply adding a new line to the database so we can constrain it. The interesting part starts with:

```
'To query the subentities of the line, we create and use a protocol extension
(Pe) provided by the associativity API
Dim subentityIdPE As AssocPersSubentityIdPE
Dim peCls As RXClass =
AssocPersSubentityIdPE.GetClass(GetType(AssocPersSubentityIdPE))
Dim pSubentityIdPE As IntPtr = entity.QueryX(peCls)
If pSubentityIdPE = IntPtr.Zero Then
    System.Windows.MessageBox.Show("cannot get pSubentityIdPE")
    Exit Sub
End If
subentityIdPE = TryCast(AssocPersSubentityIdPE.Create(pSubentityIdPE, False),
AssocPersSubentityIdPE)
If subentityIdPE Is Nothing Then
    System.Windows.MessageBox.Show("cannot get subentityIdPE")
    Exit Sub
End If
```

The Parametric API is implemented using protocol extensions. If you're not an ObjectARX programmer, then this is probably the first time you've seen a protocol extension in the AutoCAD API. A protocol extension is a way to dynamically extend a class at runtime (i.e. add new methods and properties to the class without having to rewrite the class itself). You can find out more in the ObjectARX Developers Guide in the 'Advanced Topics->Protocol Extension' section. The code here is querying the entity (the line we created earlier) to see if a protocol extension class with name AssocPersSubentityIdPE has been registered for the entity's class. If it does, then the protocol extension class is instantiated using its Create method.

We now use the protocol extension to query the subentities (edges and vertices) we can use to constrain the entity. Normally, you will constrain an edge by its vertex, and the next section of code retrieves all the edges for this entity. Note that we specifically request edge subentities by passing SubentityType.Edge into GetAllSubentities:

```
'Now we have the PE, we query the subentities
Dim edgeSubentityIds As SubentityId() = Nothing
'First we retrieve a list of all edges (a line has one edge)
edgeSubentityIds = subentityIdPE.GetAllSubentities(entity, SubentityType.Edge)
```

Note: Some commands in the *Parametric_API_Demo_VB* sample project combine the code to create the Protocol Extension creation and return the FullSubentityPath of the first edge into a helper function – *GetFullSubentityPath*.

As this is a line we know it has just one edge - other entities may have more. We use the first edge from the array returned by our call to GetAllSubentities to retrieve the vertices for that edge (GetEdgeVertexSubentities):

```
'Now we retrieve the vertices associated with the first edge in our array.
```

```
'In this case we have one edge, and the edge has three vertices - start, end
and middle.
Dim startSID As SubentityId = SubentityId.Null, endSID As SubentityId =
SubentityId.Null
Dim other As SubentityId() = Nothing
subentityIdPE.GetEdgeVertexSubentities(entity, edgeSubentityIds(0), startSID,
endSID, other)
```

Next we convert the SubentityIds for the edge and the edge startpoint into FullSubentityPaths:

```
'The PE returns a SubEntId. We want a FullSubentityPath
Dim subentPathEdge As FullSubentityPath, subentPath1 As FullSubentityPath
subentPathEdge = New FullSubentityPath(New ObjectId(0) {entity.ObjectId},
edgeSubentityIds(0)) 'The line edge
subentPath1 = New FullSubentityPath(New ObjectId(0) {entity.ObjectId},
startSID) 'The edge's startpoint.
```

All constraints are added to an Assoc2dConstraintGroup. We're using a helper function (GetConstraintGroup⁸) to retrieve the ObjectId of the existing constraint group (or create a new one if the constraint group doesn't exist already). Then we open the constraint group in our transaction.

```
'We call a helper function to retrieve or create a constraints group
Dim consGrpId As ObjectId = GetConstraintGroup(True)
Using constGrp As Assoc2dConstraintGroup = DirectCast(myT.GetObject(consGrpId,
OpenMode.ForWrite, False), Assoc2dConstraintGroup)
```

Now we have our edge and vertex FullSubentityPaths, and we've opened our Assoc2dConstraintGroup for write, we're ready to create our constraint. We have to tell the constraint group the geometry we're constraining (the line's edge). We call the AddConstrainedGeometry function for this. Then we tell the constraint group what type of constraint we're adding (a Fix constraint) and what we're constraining the edge by (its start point) using the AddGeometricalGonstraint function:

```
'Pass in geometry to constrain (the line edge)
Dim consGeom As ConstrainedGeometry =
constGrp.AddConstrainedGeometry(subentPathEdge)
'Now create the constraint, a Fixed constraint applied to the line's
startpoint.
Dim paths As FullSubentityPath() = New FullSubentityPath(0) {subentPath1}
Dim newConstraint As GeometricalConstraint =
constGrp.AddGeometricalConstraint(GeometricalConstraint.ConstraintType.Fix, paths)
End Using
myT.Commit()
End Using
End Sub
```

Congratulations, you've created your first geometric constraint. When you run this code, you'll probably find that the constraint works, but you can't see the constraint glyph. Just invoke the _CONSTRAINTBAR _SHOWALL command, and it will be displayed.

⁸ We'll explain what the helper function does in the [next section](#).

The steps you'll take to add any geometric constraint are always the same as this:

1. Use the AssocPersSubentityIdPE protocol extension to retrieve the subentities of the entities you're interested in.
2. Get or create the Assoc2dConstraintGroup for the plane the entities lie in.
3. Add the geometry (subentities) you want to constrain to the Assoc2dConstraintGroup.
4. Create the constraint.

Code walkthrough – Creating a Constraint Group

When adding our Fix constraint, we used a helper function – GetConstraintGroup – to create a constraint group. Here is that function:

```
' Helper function to retrieve (or create) constraint group
Private Shared Function GetConstraintGroup(ByVal createIfDoesNotExist As Boolean)
As ObjectId
    ' Calculate the current plane on which new entities are added by the editor
    ' (A combination of UCS and ELEVATION sysvar).
    Dim ed As Autodesk.AutoCAD.EditorInput.Editor =
Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDocument.Edi
tor
    Dim ucsMatrix As Matrix3d = ed.CurrentUserCoordinateSystem
    Dim origin As Point3d = ucsMatrix.CoordinateSystem3d.Origin
    Dim xAxis As Vector3d = ucsMatrix.CoordinateSystem3d.Xaxis
    Dim yAxis As Vector3d = ucsMatrix.CoordinateSystem3d.Yaxis
    Dim zAxis As Vector3d = ucsMatrix.CoordinateSystem3d.Zaxis
    origin = origin + CDBl(Application.GetSystemVariable("ELEVATION")) * zAxis
    Dim currentPlane As New Plane(origin, xAxis, yAxis)

    ' get the constraint group from block table record
    Dim idConstrGroup As ObjectId = ObjectId.Null
    Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
    Dim networkId As ObjectId =
AssocNetwork.GetInstanceFromObject(SymbolUtilityServices.GetBlockModelSpaceId(db),
createIfDoesNotExist, True, "")
    If networkId.IsNull Then
        System.Windows.MessageBox.Show("network id is null")
        Return idConstrGroup
    End If

    ' Try to find the constraint group in the associative network
    Dim tm As Autodesk.AutoCAD.DatabaseServices.TransactionManager =
db.TransactionManager
    Using myT As Transaction = tm.StartTransaction()
        Using network As AssocNetwork = DirectCast(myT.GetObject(networkId,
OpenMode.ForRead, False), AssocNetwork)
            If network Is Nothing Then
                Return idConstrGroup
            End If
            ' Iterate all actions in network to find Assoc2dConstraintGroups
            Dim actionsInNetwork As ObjectIdCollection = network.GetActions
            For nCount As Integer = 0 To actionsInNetwork.Count - 1
                Dim idAction As ObjectId = actionsInNetwork(nCount)
                If idAction = ObjectId.Null Then
                    Continue For
                End If
            End For
        End Using
    End Using
End Function
```

```

        End If
        ' Is this action a type of Assoc2dConstraintGroup?
        If
            idAction.ObjectClass.IsDerivedFrom(RXObject.GetClass(GetType(Autodesk.AutoCAD.Database
            Services.Assoc2dConstraintGroup))) Then
                Using action As AssocAction = DirectCast(myT.GetObject(idAction,
                OpenMode.ForRead, False), AssocAction)
                    If action Is Nothing Then
                        Continue For
                    End If
                    Dim constGrp As Assoc2dConstraintGroup = DirectCast(action,
                    Assoc2dConstraintGroup)
                    ' Is this the Assoc2dConstraintGroup for our plane of interest?
                    If constGrp.GetWorkPlane.IsCoplanarTo(currentPlane) Then
                        ' If it is then we've found an existing constraint group we can use.
                        Return idAction
                    End If
                End Using
            End If
        Next
    End Using

    ' If we get to here, a suitable constraint group doesn't exist, create a new
    one if that's what calling fn wanted.
    If idConstrGroup.IsNull AndAlso createIfDoesNotExist Then
        Using network As AssocNetwork = DirectCast(myT.GetObject(networkId,
        OpenMode.ForWrite, False), AssocNetwork)
            ' Create construction plane
            Dim constraintPlane As New Plane(currentPlane)
            ' If model extent is far far away from origin then we need to shift
            ' construction plane origin within the model extent.
            ' (Use Pextmin, Pextmax in paper space)
            Dim extmin As Point3d = db.Extmin
            Dim extmax As Point3d = db.Extmax
            If extmin.GetAsVector().Length > 100000000.0 Then
                Dim originL As Point3d = extmin + (extmax - extmin) / 2.0
                Dim result As PointOnSurface = currentPlane.GetClosestPointTo(originL)
                constraintPlane.[Set](result.GetPoint(), currentPlane.Normal)
            End If
            ' Create the new constraint group and add it to the associative network.
            Using constGrp As New Assoc2dConstraintGroup(constraintPlane)
                idConstrGroup = db.AddDBObject(constGrp)
            End Using
            network.AddAction(idConstrGroup, True)
        End Using
    End If
    myT.Commit()
End Using
Return idConstrGroup
End Function

```

The Parametric Drawing feature is a 2d constraints system, so we have to create a new Assoc2dConstraintGroup for each unique plane within the BlockTableRecord in which we're working (normally that would be model space). We can have multiple constraint groups associated with a single block, but each will be for a different geometric plane. We assume the user wants to use a constraint group defined for the current default entity insertion point for a

drawing, which is defined by the current UCS with a possible offset due to the ELEVATION sysvar. The first part of our code calculates that plane:

```
' Calculate the current plane on which new entities are added by the editor
' (A combination of UCS and ELEVATION sysvar).
Dim ed As Autodesk.AutoCAD.EditorInput.Editor =
Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDocument.Editor
Dim ucsMatrix As Matrix3d = ed.CurrentUserCoordinateSystem
Dim origin As Point3d = ucsMatrix.CoordinateSystem3d.Origin
Dim xAxis As Vector3d = ucsMatrix.CoordinateSystem3d.Xaxis
Dim yAxis As Vector3d = ucsMatrix.CoordinateSystem3d.Yaxis
Dim zAxis As Vector3d = ucsMatrix.CoordinateSystem3d.Zaxis
origin = origin + CDBl(Application.GetSystemVariable("ELEVATION")) * zAxis
Dim currentPlane As New Plane(origin, xAxis, yAxis)
```

An important note about this sample code – All the sample commands in the *Parametric_API_Demo_VB* sample create entities using the WCS, but this helper function creates a constraint group using the UCS. Therefore, it's possible that if you run this code with the UCS set to something other than World, you will find an *eNonCoplanarGeometry* exception being thrown by calls to *Assoc2dConstraintGroup.AddConstrainedGeometry*. These examples could have used a much simpler version of the *GetConstraintGroup* helper function. I've included this more complex version to highlight that constraint groups are plane dependent.

Now we've calculated our working plane, we can check if there's already a constraint group defined for this plane. To do this we must first open the associative network (*AssocNetwork*) object:

```
' get the constraint group from block table record
Dim idConstrGroup As ObjectId = ObjectId.Null
Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
Dim networkId As ObjectId =
AssocNetwork.GetInstanceFromObject(SymbolUtilityServices.GetBlockModelSpaceId(db),
createIfDoesNotExist, True, "")
If networkId.IsNull Then
    System.Windows.MessageBox.Show("network id is null")
    Return idConstrGroup
End If
```

Each *BlockTableRecord* can have up to one *AssocNetwork*. In the unlikely event that we can't find or create one (if *networkId* is null), then something has gone wrong and we return a null *ObjectId* from the function to indicate a problem.

Each *AssocNetwork* holds a collection of *Actions*, so we next open the *AssocNetwork* for read and iterate through all its actions looking for *AssocActions* that are also *Assoc2dConstraintGroups* (and doing a little error checking along the way):

```
' Try to find the constraint group in the associative network
Dim tm As Autodesk.AutoCAD.DatabaseServices.TransactionManager =
db.TransactionManager
Using myT As Transaction = tm.StartTransaction()
```



```

Using network As AssocNetwork = DirectCast(myT.GetObject(networkId,
OpenMode.ForRead, False), AssocNetwork)
If network Is Nothing Then
    Return idConstrGroup
End If
' Iterate all actions in network to find Assoc2dConstraintGroups
Dim actionsInNetwork As ObjectIdCollection = network.GetActions
For nCount As Integer = 0 To actionsInNetwork.Count - 1
    Dim idAction As ObjectId = actionsInNetwork(nCount)
    If idAction = ObjectId.Null Then
        Continue For
    End If
    ' Is this action a type of Assoc2dConstraintGroup?
    If
idAction.ObjectClass.IsDerivedFrom(RXObject.GetClass(GetType(Autodesk.AutoCAD.Database
Services.Assoc2dConstraintGroup))) Then
        Using action As AssocAction = DirectCast(myT.GetObject(idAction,
OpenMode.ForRead, False), AssocAction)
            If action Is Nothing Then
                Continue For
            End If

```

Once we've found a constraint group, we open it for read and check if it is the group for our working plane. If it is, we return its ObjectId to the calling function

```

        Dim constGrp As Assoc2dConstraintGroup = DirectCast(action,
Assoc2dConstraintGroup)
        ' Is this the Assoc2dConstraintGroup for our plane of interest?
        If constGrp.GetWorkPlane.IsCoplanarTo(currentPlane) Then
            ' If it is then we've found an existing constraint group we can use.
            Return idAction

```

If we get to the end of the loop and haven't found a constraint group for our working plane, then we create one (if the caller specified that we should). Note how we add the constraint group to the database and also to the AssocNetwork:

```

If idConstrGroup.IsNull AndAlso createIfDoesNotExist Then
    Using network As AssocNetwork = DirectCast(myT.GetObject(networkId,
OpenMode.ForWrite, False), AssocNetwork)
        ' Create construction plane
        Dim constraintPlane As New Plane(currentPlane)
        ' If model extent is far far away from origin then we need to shift
        ' construction plane origin within the model extent.
        ' (Use Pextmin, PExtmax in paper space)
        Dim extmin As Point3d = db.Extmin
        Dim extmax As Point3d = db.Extmax
        If extmin.GetAsVector().Length > 100000000.0 Then
            Dim originL As Point3d = extmin + (extmax - extmin) / 2.0
            Dim result As PointOnSurface = currentPlane.GetClosestPointTo(originL)
            constraintPlane.[Set](result.GetPoint(), currentPlane.Normal)
        End If
        ' Create the new constraint group and add it to the associative network.
        Using constGrp As New Assoc2dConstraintGroup(constraintPlane)
            idConstrGroup = db.AddDBObject(constGrp)
        End Using
        network.AddAction(idConstrGroup, True)

```



```
End Using
End If
```

Then we return the ObjectId of the constraint group we created and we're done.

Code Walkthrough – Defining a User Parameter

We've seen how to create geometric constraints, so now it's time to move on to dimensional constraints. But before we do, let's take a helpful detour and look at how to add user parameters (AssocVariables) to our network.

Dimensional constraints are fairly limited unless we can also define user parameters (or variables) to include in the expressions governing those constraints. The Parametric_API_Demo_VB samples includes a helper function for this – AddOrModifyVariable – that is used by all the functions that add dimensional constraints. We'll examine this code now:

```
' Add a new variable to the associative network or modify expression of the
existing one
Public Shared Function AddOrModifyVariable(ByVal varName As String, ByVal
varExpression As String) As ObjectId
    Dim varId As ObjectId = ObjectId.Null
    Dim doc As Document = Application.DocumentManager.MdiActiveDocument
    Dim db As Database = doc.Database
    Using myT As Transaction = db.TransactionManager.StartTransaction()
        ' Open the AssocNetwork
        Dim networkId As ObjectId =
AssocNetwork.GetInstanceFromObject(SymbolUtilityServices.GetBlockModelSpaceId(db),
True, True, "")
        Dim network As AssocNetwork = DirectCast(myT.GetObject(networkId,
OpenMode.ForWrite), AssocNetwork)

        Dim var As AssocVariable = Nothing
        ' Iterate through all actions in the network
        Dim actionIds As ObjectIdCollection = network.GetActions
        For Each actionId As ObjectId In network.GetActions
            ' Is this action an AssocVariable?
            If
actionId.ObjectClass.IsDerivedFrom(RXObject.GetClass(GetType(Autodesk.AutoCAD.Database
Services.AssocVariable))) Then
                ' If so, we check if it has the name we're looking for.
                var = DirectCast(myT.GetObject(actionId, OpenMode.ForWrite),
AssocVariable)
                If var IsNot Nothing Then
                    ' If name matches, then we exit loop and set its expression
                    If var.Name = varName Then
                        Exit For
                    Else
                        var = Nothing
                    End If
                End If
            End If
        Next
        ' If variable with correct name wasn't found, we create it.
        If var Is Nothing Then
```

```

        var = New AssocVariable()
        varId = network.Database.AddDBObject(var)
        network.AddAction(varId, True)
        myT.AddNewlyCreatedDBObject(var, True)
        var.SetName(varName, True)
    End If
    ' Finally we set its expression to the new value.
    Dim errMsg As String = ""
    var.SetExpression(varExpression, "", True, True, errMsg, False)
    Dim rb As New ResultBuffer()
    errMsg = var.EvaluateExpression(rb)
    var.Value = rb
    myT.Commit()
End Using
Return varId
End Function

```

The code to access the AssocNetwork should be familiar by now:

```

Dim networkId As ObjectId =
AssocNetwork.GetInstanceFromObject(SymbolUtilityServices.GetBlockModelSpaceId(db),
True, True, "")
Dim network As AssocNetwork = DirectCast(myT.GetObject(networkId,
OpenMode.ForWrite), AssocNetwork)

```

Next we iterate all actions in the network, looking for AssocVariables. If we find one, then we check if it has the name we need:

```

Dim var As AssocVariable = Nothing
' Iterate through all actions in the network
Dim actionIds As ObjectIdCollection = network.GetActions
For Each actionId As ObjectId In network.GetActions
    ' Is this action an AssocVariable?
    If
actionId.ObjectClass.IsDerivedFrom(RXObject.GetClass(GetType(Autodesk.AutoCAD.Database
Services.AssocVariable))) Then
        ' If so, we check if it has the name we're looking for.
        var = DirectCast(myT.GetObject(actionId, OpenMode.ForWrite),
AssocVariable)
        If var IsNot Nothing Then
            ' If name matches, then we exit loop and set its expression
            If var.Name = varName Then
                Exit For
            Else
                var = Nothing
            End If
        End If
    End If
End If
Next

```

When we exit this loop, 'var' is either pointing to the AssocVariable with the correct name or it is Nothing. A value of Nothing indicates that no variable with that name was found. If we didn't find the variable we wanted, then we create a new one:

```

If var Is Nothing Then

```

```
var = New AssocVariable()
varId = network.Database.AddDBObject(var)
network.AddAction(varId, True)
myT.AddNewlyCreatedDBObject(var, True)
var.SetName(varName, True)
End If
```

Finally, whether it was an existing variable or one we just created, we set its value and return the variable's ObjectId to indicate success. Variable expressions are strings, because an expression can be a value or an equation (the TESTVARIABLECREATE command in the sample demonstrates setting the expression to an equation); variable values are result buffers, because the values could have different types:

```
' Finally we set its expression to the new value.
Dim errMsg As String = ""
var.SetExpression(varExpression, "", True, True, errMsg, False)
Dim rb As New ResultBuffer()
errMsg = var.EvaluateExpression(rb)
var.Value = rb
myT.Commit()
End Using
Return varId
```

Code Walkthrough – Adding a Radial Dimension Constraint

Now we're ready to create our dimensional constraint – a simple radial dimension in this case. Creating a dimensional constraint requires a little more work than creating a geometric constraint: you still have to query the entity you're constraining for its subentities (via its Protocol Extension), but you also have to create a Dimension entity, define the user parameters that drive the dimension, and link them all together via 'dependency' objects. Here's the code for constraining a Circle using a RadialDimension:

```
' Add a radial dimensional constraint to a circle
<CommandMethod("TESTRADIUS")> _
Public Shared Sub testRadiusCommand()
    Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
    Dim tm As Autodesk.AutoCAD.DatabaseServices.TransactionManager =
db.TransactionManager
    Dim circle As Circle
    Using myT As Transaction = tm.StartTransaction()
        ' Create new circle entity
        Dim bt As BlockTable = DirectCast(myT.GetObject(db.BlockTableId,
OpenMode.ForRead, False), BlockTable)
        Dim btr As BlockTableRecord =
DirectCast(myT.GetObject(bt(BlockTableRecord.ModelSpace), OpenMode.ForWrite, False),
BlockTableRecord)
        circle = New Circle(New Point3d(25, 25, 0), New Vector3d(0, 0, 1), 3)
        btr.AppendEntity(circle)
        myT.AddNewlyCreatedDBObject(circle, True)
        ' Create new radial dimension entity
        Dim chordPoint As New Point3d(circle.Center.X + circle.Radius,
circle.Center.Y, circle.Center.Z)
```

```

    Dim dimRadius As New RadialDimension(circle.Center, chordPoint, circle.Radius,
    "", db.DimStyleTableId)
    btr.AppendEntity(dimRadius)
    myT.AddNewlyCreatedDBObject(dimRadius, True)
    ' Get FullSubentityPath of circle edge.
    Dim subentPath As FullSubentityPath = GetFullSubentityPath(circle,
SubentityType.Edge)
    Dim varId As ObjectId, valDepId As ObjectId
    'create user parameter
    varId = AddOrModifyVariable("Radius", "2.0")
    ' Create value dependency (the relationship between the AssocAction and a
scalar value - the variable value)
    Using valDep As New AssocValueDependency()
        valDepId = db.AddDBObject(valDep)
        Dim compoundId As New CompoundObjectId(varId, db)
        valDep.AttachToObject(compoundId)
    End Using
    ' Create dim dependency (the relationship between the AssocAction and an
Entity)
    Dim dimDepId As ObjectId = ObjectId.Null
    Dim dimDepBodyId As ObjectId = ObjectId.Null
    AssocDimDependencyBody.CreateAndPostToDatabase(dimRadius.ObjectId, dimDepId,
dimDepBodyId)
    ' If adding constraint fails, we want to keep our dimension. We reset to the
old value later
    Dim bPreviousValue As Boolean =
AssocDimDependencyBodyBase.SetEraseDimensionIfDependencyIsErased(False)
    ' Add dimensional constraint to the circle
    Dim consGrpId As ObjectId = GetConstraintGroup(True)
    Dim consGeom As ConstrainedGeometry = Nothing
    Using constGrp As Assoc2dConstraintGroup = DirectCast(myT.GetObject(consGrpId,
OpenMode.ForWrite, False), Assoc2dConstraintGroup)
        ' Constrain circle edge
        consGeom = constGrp.AddConstrainedGeometry(subentPath)
        ' Add constraint to constraint group
        constGrp.AddRadiusDiameterConstraint(consGeom,
RadiusDiameterConstraint.RadDiaConstrType.CircleRadius, valDepId, dimDepId)
        ' Reset SetEraseDimensionIfDependencyIsErased back to old value
    End Using
    AssocDimDependencyBodyBase.SetEraseDimensionIfDependencyIsErased(bPreviousValue)
End Sub

```

As usual, the first part of the code is simply creating our Circle and adding it to model space. We do the same for the RadialDimension that is going to be driven by our constraint.

Then we create our AssocPersSubentityIdPE protocol extension for the circle and use it to find the edge subentities of the circle. This time we've wrapped up all that code into the GetFullSubentityPath helper function. This function returns the FullSubentityPath of the first (edge) subentity in the array returned by the protocol extension for the circle. We use that FullSubentityPath later when we specify the geometry we're constraining.

```

    ' Get FullSubentityPath of circle edge.
    Dim subentPath As FullSubentityPath = GetFullSubentityPath(circle,
SubentityType.Edge)

```

Then we use our helper function to create the user parameter that defines the size of the dimension. Notice that we're setting this to a value of 2.0, but we created the circle with a radius of 3. Just like through the user interface, you can construct your geometry with arbitrary sizes and positions and then constrain them to bring them all into line.

```
'create user parameter
varId = AddOrModifyVariable("Radius", "2.0")
```

Next we have to create an AssocValueDependency class to link the variable to the constraint. We'll use this later when we create the actual constraint in the constraint group. This is slightly unusual in using a CompoundObjectId instead of just an ObjectId:

```
' Create value dependency (the relationship between the AssocAction and a
scalar value - the variable value)
Using valDep As New AssocValueDependency()
    valDepId = db.AddDBObject(valDep)
    Dim compoundId As New CompoundObjectId(varId, db)
    valDep.AttachToObject(compoundId)
End Using
```

Similarly, we create an AssocDimDependencyBody that we'll later use to link the constraint to the RadialDimension entity. Notice that we don't explicitly instantiate the AssocDimDependencyBody. Instead, we call the shared CreateAndPostToDatabase method, which sets dimDepId to the ObjectId of the created AssocDependency object:

```
' Create dim dependency (the relationship between the AssocAction and an
Entity)
Dim dimDepId As ObjectId = ObjectId.Null
Dim dimDepBodyId As ObjectId = ObjectId.Null
AssocDimDependencyBody.CreateAndPostToDatabase(dimRadius.ObjectId, dimDepId,
dimDepBodyId)
```

Next we create or open our Assoc2dConstraintGroup and add the FullSubentityPath of the circle's edge to the constrained geometry list:

```
' If adding constraint fails, we want to keep our dimension. We reset to the
old value later
Dim bPreviousValue As Boolean =
AssocDimDependencyBodyBase.SetEraseDimensionIfDependencyIsErased(False)
Dim consGrpId As ObjectId = GetConstraintGroup(True)
Dim consGeom As ConstrainedGeometry = Nothing
Using constGrp As Assoc2dConstraintGroup = DirectCast(myT.GetObject(consGrpId,
OpenMode.ForWrite, False), Assoc2dConstraintGroup)
    ' Constrain circle edge
    consGeom = constGrp.AddConstrainedGeometry(subentPath)
```

Then we create our constraint. Notice how we use the ObjectIds of the value and dimension dependencies (valDepId and dimDepId) rather than the ObjectIds of the RadialDimension and the AssocVariable themselves.

```
' Add constraint to constraint group
constGrp.AddRadiusDiameterConstraint(constGeom,
RadiusDiameterConstraint.RadDiaConstrType.CircleRadius, valDepId, dimDepId)
' Reset SetEraseDimensionIfDependencyIsErased back to old value
AssocDimDependencyBodyBase.SetEraseDimensionIfDependencyIsErased(bPreviousValue)
```

Miscellany

Dimensional Constraints without Dimensions

Here is an edited version of the TESTRADIUS command, where I'm no longer creating a RadialDimension or its associated dependency object. Instead I pass a null ObjectId (highlighted in yellow in the code below⁹) to the AddRadiusDiameterConstraint function. If you run this code, you'll find the circle is still constrained and you can edit the 'Radius' variable in the Parameter Manager palette. But there's no dimension in the drawing.

```
' Add a radial dimensional constraint to a circle
<CommandMethod("TESTRADIUS2")>
Public Shared Sub testRadius2Command()
    Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
    Dim tm As Autodesk.AutoCAD.DatabaseServices.TransactionManager =
db.TransactionManager
    Dim circle As Circle
    Using myT As Transaction = tm.StartTransaction()
        ' Create new circle entity
        Dim bt As BlockTable = DirectCast(myT.GetObject(db.BlockTableId,
OpenMode.ForRead, False), BlockTable)
        Dim btr As BlockTableRecord =
DirectCast(myT.GetObject(bt(BlockTableRecord.ModelSpace), OpenMode.ForWrite, False),
BlockTableRecord)
        circle = New Circle(New Point3d(25, 25, 0), New Vector3d(0, 0, 1), 3)
        btr.AppendEntity(circle)
        myT.AddNewlyCreatedDBObject(circle, True)
        ' Get FullSubentityPath of circle edge.
        Dim subentPath As FullSubentityPath = GetFullSubentityPath(circle,
SubentityType.Edge)
        Dim varId As ObjectId, valDepId As ObjectId
        'create user parameter
        varId = AddOrModifyVariable("Radius", "2.0")
        ' Create value dependency (the relationship between the AssocAction and a
scalar value - the variable value)
        Using valDep As New AssocValueDependency()
            valDepId = db.AddDBObject(valDep)
            Dim compoundId As New CompoundObjectId(varId, db)
            valDep.AttachToObject(compoundId)
        End Using
        ' Add dimensional constraint to the circle
```

⁹ Sorry if you printed this in black and white ☹.

```

Dim consGrpId As ObjectId = GetConstraintGroup(True)
Dim consGeom As ConstrainedGeometry = Nothing
Using constGrp As Assoc2dConstraintGroup = DirectCast(myT.GetObject(consGrpId,
OpenMode.ForWrite, False), Assoc2dConstraintGroup)
    ' Constrain circle edge
    consGeom = constGrp.AddConstrainedGeometry(subentPath)
    ' Add constraint to constraint group
    constGrp.AddRadiusDiameterConstraint(consGeom,
RadiusDiameterConstraint.RadDiaConstrType.CircleRadius, valDepId, ObjectId.Null)
End Using
Dim networkId As ObjectId =
AssocNetwork.GetInstanceFromObject(SymbolUtilityServices.GetBlockModelSpaceId(db),
True, True, "")
myT.Commit()
End Using
End Sub

```

Anonymous constraints

Setting dimensional constraints without inserting dimensions in the drawing isn't that useful. After all, you could have just put the dimension on a hidden layer.

But what if the user also couldn't see or edit the 'Radius' variable?

Then you'd have a constraint system that can only be edited through your application. (No chance for those pesky users to mess things up by editing stuff they shouldn't). Well this is possible. Just like you can create an anonymous block or an anonymous group, you can create an anonymous variable.

In the above code, change this line

```
varId = AddOrModifyVariable("Radius", "2.0")
```

to

```
varId = AddOrModifyVariable("*Radius", "2.0")
```

i.e. prefix the variable name with a '*'.

Then run the command and open up the Parameters Manager palette – there is no 'Radius' variable¹⁰.

Now the user can't see the dimension, and they can't edit the dimension value.

¹⁰ There is a limitation to using anonymous variables – you can't use an anonymous variable name in the expression of another AssocVariable. For example, if I have an AssocVariable with name '*MyVar1', I can't set the expression of another AssocVar to '*MyVar+1'.

Exceptions aren't always errors

Most of the time an exception means that an error has occurred. And sometimes, when we're feeling a bit lazy, we don't always check the exception to see if it's something we can manage or ignore.

I've already mentioned that the Parametric Drawing API is a shallow wrapper on top of our C++ ObjectARX API. ObjectARX functions return `Acad::ErrorStatus` values to provide information to the caller about whether the function call was successfully. A return code of `Acad::ErrorStatus::eOk` means everything worked fine. A return code of something else can mean "it's all gone horribly wrong", but it can sometimes mean "everything worked ok, but there's something you might be interested to know".

Being a shallow wrapper, the Parametric Drawing .NET API simply takes any `ErrorStatus` returned from the underlying ObjectARX code that isn't `eOk`, wraps it in an exception, and throws it at you.

There's one place in particular where this behavior is rather bothersome – in the `Assoc2dConstraintGroup.AddConstrainedGeometry` function. If you're adding several constraints to a drawing, it's likely you'll be adding more than one constraint to the same geometry¹¹. When you call `AddConstrainedGeometry` for the second time with the same geometry, the underlying ObjectARX code says "nothing to worry about, but I just wanted you to know that that geometry was already in the list". The result is that you end up with an exception being thrown with `ErrorStatus = AlreadyInGroup`¹².

Now I know what you're wondering – Is there a function to retrieve geometry I already added to the constraint group?

Yes there is – there's `Assoc2dConstraintGroup.GetConstrainedGeometry`.

And now you're thinking – Well that's easy. We can use that function to check if the geometry is already in the group before we try to add it, and so avoid throwing the exception.

Unfortunately it's not that simple - `GetConstrainedGeometry` throws an exception with `ErrorStatus = NotApplicable` if we try to retrieve geometry that isn't in the group.

So, whatever you do, you end up throwing an exception. Doh!

You can handle this in your code using a helper function something like this:

```
' Helper function to handle exceptions being thrown if you try to add the same
constrained geometry twice.
' You could work out what constrained geometry to add up front. That would be more
efficient as no exceptions would be thrown,
```

¹¹ For example, if you want to fix a line and also make its end point coincident with another point.

¹² This is unfortunate, because routinely throwing exceptions can have a significant impact on the performance of your code.


```
' but isn't always possible (e.g. if running a command multiple times to
constrain existing geometry in a drawing.
' (This helper function isn't used by any functions in this sample project).
Private Shared Function AddConstrainedGeometry(ByVal constGrp As
Assoc2dConstraintGroup, ByVal path As FullSubentityPath) As ConstrainedGeometry
    Dim consGeom As ConstrainedGeometry = Nothing
    Try
        consGeom = constGrp.AddConstrainedGeometry(path)
    Catch ex As Autodesk.AutoCAD.Runtime.Exception
        ' This error isn't really an error
        If ex.ErrorStatus <> ErrorStatus.AlreadyInGroup Then
            Throw ex
        End If
        ' If we get to here, then 'path' was already in the group so we can retrieve
        it.
        consGeom = constGrp.GetConstrainedGeometry(path, False)
    End Try
    Return consGeom
End Function
```

Conclusion

And so we come to the end of this introduction to the Parametric Drawing .NET API: we've covered the basic architecture of the associative framework and the Parametric Drawing feature that uses it; and we've stepped through some simple examples of creating geometric and dimensional constraints. This should be enough for you to start using the API in your own applications. Make sure you review the `Parametric_API_Demo_VB` project downloadable from the AU Online website – it has lots more examples, covering most constraint types.

Good luck and have fun using this API in your own applications!

Further Reading

- A good starting point for all .NET developers is the resources listed on the AutoCAD Developer Center – www.autodesk.com/developautocad. These include:
 - Training material, recorded presentations, and our AutoCAD .NET Wizards.
 - The AutoCAD .NET Developers Guide - <http://www.autodesk.com/autocad-net-developers-guide>.
 - Information on joining the Autodesk Developer Network – www.autodesk.com/joinadn
 - Information on training classes and webcasts – www.autodesk.com/apitraining
 - Links to the Autodesk discussion groups – www.autodesk.com/discussion. (Click on the AutoCAD link from there to access the AutoCAD API discussion groups).
- You'll find a conceptual overview of the Associative Framework in the ObjectARX Developer's Guide section 'Advanced Topics->Associative Framework'. This is an overview of the framework in general, and doesn't describe the Parametric Drawing API.
- The ObjectARX Reference Guide has detailed descriptions of the Parametric Drawing API classes. All classes begin with 'Assoc' ('AcDbAssoc' in ObjectARX). Sometimes, the ObjectARX class documentation is more detailed than the .NET documentation (and vice versa) so make sure you review both.
- Protocol Extensions are described (from a C++ standpoint) in the ObjectARX Developer's Guide section 'Advanced Topics->Protocol Extension'.
- If you're an ADN partner, there is a wealth of Autodesk API information on the members-only ADN website – <http://adn.autodesk.com>.
 - And ADN members can ask unlimited API questions through our DevHelp Online interface
- Watch out for our regular ADN DevLab events. DevLab is a programmers' workshop (free to ADN and non-ADN members) where you can come and discuss your AutoCAD programming problems with the ADN DevTech team. They are advertised on our API training schedule accessible from www.autodesk.com/apitraining.