Overview

Topics in this section

- Introduction
- Using the API Reference
- Setting Up Visual Studio
- AutoCAD

ntroduction

The AutoCAD® Map 3D 2008 .NET API provides access to AutoCAD Map 3D functionality so you can modify and extend it for your own purposes. The API can be used by any .NET language.

The snippets in this guide are mainly in VB.NET, but most samples are available in VB.NET, C#, and C++.

Some of the short snippets in this guide write to the AutoCAD Map 3D console. Ensure that the console is visible. Press **CTRL+9** to display the or hide console.

Namespaces

The `Autodesk.Gis.Map` namespace contains the .NET classes for AutoCAD Map 3D. Some of the general-purpose classes are defined directly within the top-level `Autodesk.Gis.Map` namespace, but most are grouped into lower-level namespaces below `Autodesk.Gis.Map`.

For the sake of cleaner code, all examples within this guide will assume the following `Imports`:

```
Imports Autodesk.Gis.Map
Imports Autodesk.AutoCAD.DatabaseServices
Imports Autodesk.AutoCAD.ApplicationServices
```

In this guide, objects from namespaces within `Autodesk.Gis.Map` are partially qualified. For example, the `Table` class in the `Autodesk.Gis.Map.ObjectData` namespace is generally given as `ObjectData.Table`. This makes it simple to find the class details in the API Reference.

Certain chapters may define additional imports. For example, the chapter about Object Data defines

```
Imports Autodesk.Gis.Map.ObjectData
```

Related Documentation

AutoCAD Map 3D 2009 includes the new Geospatial Platform API for working with geospatial data. See the Geospatial Platform Developer’s Guide for details.

Using the API Reference

The API reference applies to multiple languages. Because of differences between languages, the terms and syntax used in the reference may not match a given language. Some differences are:

API Ref	VB.NET	C#
__abstract	MustInherit	abstract
__sealed	NotInheritable	sealed
__gc		
::	.	.
NULL	Nothing	Null

This guide uses VB.NET for most examples. The sample applications are available in VB.NET, C#, and C++;

etting Up Visual Studio

The AutoCAD Map 3D SDK requires Microsoft Visual Studio 2005. To set up a project for a custom application, open the project properties.

Note These instructions apply to VB.NET. Setting up a project for C# is slightly different.

On the Application tab, set the application type to Class library.

Add the following references. The DLLs are in the AutoCAD Map 3D installation folder:

- acdbmgd.dll
- acmgd.dll
- ManagedMapApi.dll

`acdbmgd.dll` contains the following AutoCAD namespaces:

- `Autodesk.AutoCAD.Colors`
- `Autodesk.AutoCAD.ComponentModel`
- `Autodesk.AutoCAD.DatabaseServices`
- `Autodesk.AutoCAD.DatabaseServices.Filters`
- `Autodesk.AutoCAD.Geometry`
- `Autodesk.AutoCAD.GraphicsInterface`
- `Autodesk.AutoCAD.GraphicsSystem`
- `Autodesk.AutoCAD.LayerManager`
- `Autodesk.AutoCAD.Runtime`

`acmgd.dll` contains the following AutoCAD namespaces:

- Autodesk.AutoCAD.ApplicationServices
- Autodesk.AutoCAD.EditorInput
- Autodesk.AutoCAD.GraphicsSystem
- Autodesk.AutoCAD.PlottingServices
- Autodesk.AutoCAD.Publishing
- Autodesk.AutoCAD.Runtime
- Autodesk.AutoCAD.Windows
- Autodesk.AutoCAD.Windows.ToolPalette

ManagedMapApi.dll contains the following AutoCAD Map 3D namespaces:

- Autodesk.Gis.Map
- Autodesk.Gis.Map.Annotation
- Autodesk.Gis.Map.Classification
- Autodesk.Gis.Map.Constants
- Autodesk.Gis.Map.DisplayManagement
- Autodesk.Gis.Map.Filters
- Autodesk.Gis.Map.ImportExport
- Autodesk.Gis.Map.MapBook
- Autodesk.Gis.Map.ObjectData
- Autodesk.Gis.Map.Project
- Autodesk.Gis.Map.Query
- Autodesk.Gis.Map.Topology
- Autodesk.Gis.Map.Utilities

For each of the references, set the Copy Local property to False. Double-click the reference to open the properties.

Set the reference path to the AutoCAD Map 3D installation directory.

On the Debug tab, set:

- Start external program: *`InstallDir`*`\acad.exe`, where *InstallDir* is the installation directory for AutoCAD Map 3D.
- Working directory: *`InstallDir`*`\UserDataCache\`

Running AutoCAD Map 3D Custom Applications

Every custom application requires at least one subroutine that can be called from AutoCAD Map 3D. Identify this using the `CommandMethod` attribute. For example, using VB.NET the syntax is:

```
<CommandMethod("CustomCommand")> _
Public Sub CommandSub()
```

Using C# the syntax is:

```
[CommandMethod("CustomCommand")]
public void CommandSub();
```

To run a custom application, type the `NETLOAD` command at the AutoCAD Map 3D command prompt. Browse to the DLL containing the custom application assembly. Open the assembly. This makes any custom commands defined using the `CommandMethod` attribute available to the AutoCAD Map 3D session.

To execute a command, type the command method at the command prompt. In the example above, this would be

```
CustomCommand
```

For more details, refer to the AutoCAD documentation.

AutoCAD

AutoCAD Map 3D relies on AutoCAD for much of its functionality. It is important to understand some basic AutoCAD concepts before writing AutoCAD Map 3D applications. For complete details, refer to the AutoCAD developer documentation.

In particular, managing objects in the AutoCAD database is important.

Transactions

The AutoCAD database uses a transaction model for access to all objects.

To use any object in the database, start a transaction and use the transaction to open the database object in either read-only or read-write mode. `Transaction.Open()` returns a generic reference. Cast that to the type of object being returned. For example, given a database object id for a `MapBook` object, the following will return a reference to the object:

```
Dim bookObj As MapBook.Book
bookObj = CType(trans.GetObject(mapBookId, OpenMode.ForWrite),
   MapBook.Book)
```

Short examples in this guide may not include all the transaction processing, so they can highlight the concepts being discussed. In all cases, though, if any changes are being made to the drawing, it should be assumed that the following general structure is in place:

```
Dim trans As Transaction = Nothing
Dim docs As DocumentCollection = Application.DocumentManager
Dim activeDoc As Document = docs.MdiActiveDocument
Try
   trans = activeDoc.TransactionManager.StartTransaction()
   '
   ' Open object(s)
   '
```

```
    Dim bookObj As MapBook.Book
    bookObj = CType(trans.GetObject(mapBookId, OpenMode.ForWrite),
       MapBook.Book)
    '
    ' Insert code to process transaction
    '
    ' Commit transaction
    '
    trans.Commit()
Catch
    '
    ' Handle exception, and cancel transaction
    '
Finally
    trans.Dispose()
End Try
```

Although transactions can be nested, this is not recommended. One complication is that adding an entity takes place immediately, but removing an entity does not take effect until the transaction has been committed.

Note Many examples in this guide assume that `activeDoc` refers to the active document.

---

ession and Project

Topics in this section

Overview

An AutoCAD Map 3D session represents the active state of the Map 3D application.

Most of the classes for working with the session are defined in the `Autodesk.Gis.Map` namespace.

There is a single instance of the Map application, available through the `Application` property of the abstract class

```
Autodesk.Gis.Map.HostMapApplicationServices
```

This returns a `MapApplication` object that represents the entire application. It has some read-only properties that provide access to objects in the session. One of the main properties is `Projects`.

`Projects` returns a collection of all open projects, a `ProjectCollection` object in the `Autodesk.Gis.Map.Project` namespace. A project is represented by a `ProjectModel` object. A project is the container for a Map 3D drawing and all its associated objects. Nearly all interaction with a drawing begins with a project.

`ActiveProject` returns the `ProjectModel` for the currently active project.

Note For historical reasons, the API uses the term *project* where the user interface will normally use *map* or *drawing*.

For example, the following gets the current project:

```
Dim mapApp As MapApplication
mapApp = HostMapApplicationServices.Application
Dim activeProj As Project.ProjectModel
activeProj = mapApp.ActiveProject
```

The following processes all open projects:

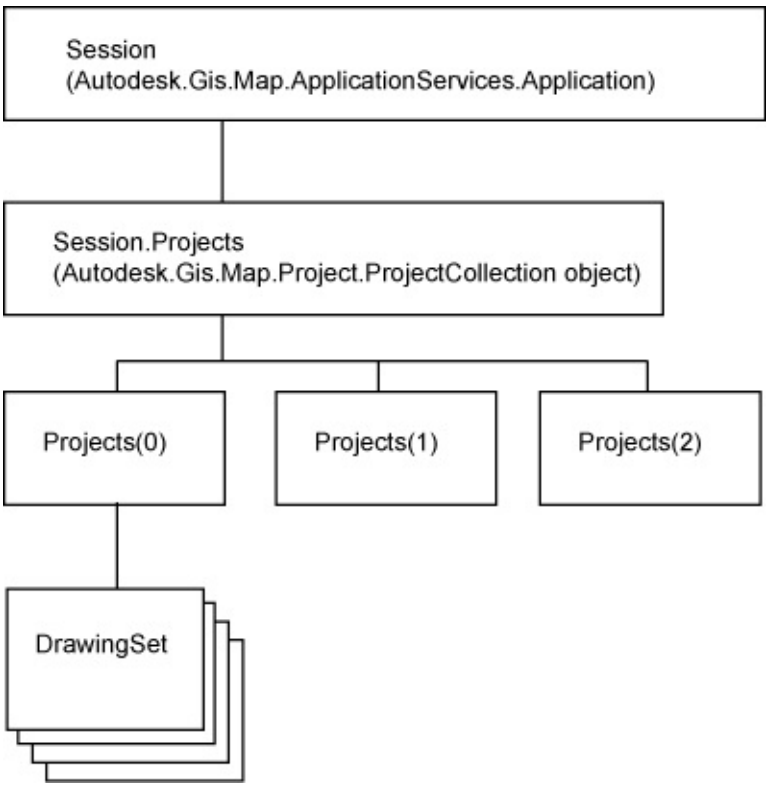
```
Dim mapApp As MapApplication
mapApp = HostMapApplicationServices.Application
Dim projList As Project.ProjectCollection
projList = mapApp.Projects
For Each project As Project.ProjectModel In projList
    ' Process projects
Next
```

Note Many of the examples in this guide assume that `mapApp` and `activeProj` have already been defined as in the example above.

A project is closely related to an AutoCAD drawing. In AutoCAD Map 3D, opening a drawing automatically creates a new project. You cannot instantiate a new project directly.

`MapApplication.GetDocument()` returns the AutoCAD document associated with a project. Conversely, `Project.ProjectCollection.GetProject()` returns the `ProjectModel` for an AutoCAD document.

For example, the following diagram shows a session that has three open projects. The first project refers to external drawings in its drawing set. Any one of the projects could be active at one time. Each project has a corresponding AutoCAD drawing.



Additional Project Properties

`ProjectModel` objects have many properties that provide access to different aspects of the drawing. For example, the `DrawingSet` property gets the drawing set for the project.

Some of the important project properties are:

- Annotations
- Database
- DrawingSet
- ODTables
- Topologies

For more details about the various properties, refer to the appropriate chapters in this guide.

AutoCAD Database

The `ProjectModel.Database` property returns a reference to the

AutoCAD database. This is necessary for many AutoCAD operations.

Project Options

The `ProjectModel.Options` property returns a reference to the project's options.

Drawing Objects

Drawing objects are the visible items in an AutoCAD drawing.

Note The AutoCAD developer documentation generally uses the term *entity* or *AcDb entity* when discussing drawing objects.

In the AutoCAD API, an `Autodesk.AutoCAD.DatabaseServices.DBObject` object represents any object in the drawing database, including drawing objects. A `DBObject` can be referred to by either:

- `Autodesk.AutoCAD.DatabaseServices.Handle`
- `Autodesk.AutoCAD.DatabaseServices.ObjectID`

A `Handle` is a persistent identifier that is stored with the AutoCAD database when it is saved. Each handle is unique within a single drawing, but different drawings are likely to have duplicate handles referring to separate objects.

An `ObjectID` is used for quick access to drawing objects within an AutoCAD session. They are not persistent, though. They expire when the drawing is closed.

Map Objects

Because a single AutoCAD Map 3D project can include more than one AutoCAD drawing, an AutoCAD handle is not sufficient to uniquely identify an object.

A `MapObjectId`, defined in the namespace `Autodesk.Gis.Map.Utilities`, identifies an object by its AutoCAD handle and by its drawing identifier. The `MapObjectId.ObjectHandle` property is the AutoCAD handle, and `MapObjectId.DrawingId` is the drawing identifier, an object of type `Utilities.MapId`.

)bject Data

Topics in this section

ntroduction

Object data provides a way of attaching additional information to drawing objects. It is more powerful and flexible than AutoCAD block attributes because object data can be attached to any object in a drawing.

Note Object data can only be attached to drawing objects. FDO feature sources have a different way to handle feature properties.

The classes for handling object data are mostly within the `ObjectData` namespace. Code in this chapter assumes the following:

```
Imports Autodesk.Gis.Map.ObjectData
```

Tables

Internally, object data is stored in tables. Each drawing has its own set of tables, available from the `ProjectModel.ODTables` property. This returns an object of type `ObjectData.Tables`.

For example, if `mapApp` is the Map application, the following will get the object data tables for the active drawing:

```
Dim activeProject As Project.ProjectModel = mapApp.ActiveProject
Dim tableList As ObjectData.Tables = activeProject.ODTables
```

`ObjectData.Tables.GetTableNames()` returns a list of the table names that have been defined for the drawing.

To get a single table from the set of tables, use the `ObjectData.Tables.Item` property. Note that this requires a table name as a parameter, not a table number. For example:

```
Dim table As ObjectData.Table = tableList.Item("table1")
```

or

```
Dim table As ObjectData.Table = tableList("table1")
```

Use `Tables.IsTableDefined()` to see if a table name exists. An attempt to get a table that does not exist throws an exception.

Field Definitions

Columns in a table are defined by `ObjectData.FieldDefinition` objects, which describe the data type and default value. The data types are defined in the `Constants.DataType` enum:

- UnknownType
- Integer
- Real
- Character
- Point

Records

Each row in the table is of type `ObjectData.Record`. Every record in the table is associated with a drawing object.

The `Item` property of an `ObjectData.Record` contains the values for the record, one for each field definition in the table. Each item is of type `Utilities.MapValue`, which is a general-purpose class for storing data.

Creating a Table

Creating a table requires:

- Creating an `ObjectData.FieldDefinitions` object
- Adding field definitions for every column in the table
- Creating the table by adding the field definitions to the `ODTables` object for the drawing

Create an `ObjectData.FieldDefinitions` object using the `ProjectModel.MapUtility.NewODFieldDefinitions()` method. Add fields using the `FieldDefinitions.Add()` method. For example, if `mapApp` is the Map application, the following creates field definitions for 2 columns:

```
Dim fieldDefs As ObjectData.FieldDefinitions
fieldDefs = _
  mapApp.ActiveProject.MapUtility.NewODFieldDefinitions()
Dim def1 As ObjectData.FieldDefinition
def1 = fieldDefs.Add("FIRST_FIELD", "Owner name", _
  Autodesk.Gis.Map.Constants.DataType.Character, 0)
def1.SetDefaultValue("A")

Dim def2 As ObjectData.FieldDefinition
def2 = fieldDefs.Add("SECOND_FIELD", "Assessment year", _
  Autodesk.Gis.Map.Constants.DataType.Integer, 1)
def2.SetDefaultValue(0)
```

Get a reference to the `ODTables` property for the drawing, and add the field defintions to create a new table.

```
Dim tables As ObjectData.Tables
tables = mapApp.ActiveProject.ODTables
tables.Add("NewTable", fieldDefs, "Description", True)
```

Get a reference to the table using `Tables.Item()`. This expects a string parameter.

```
Dim table As ObjectData.Table
table = tables("NewTable")
```

Removing a Table

To remove a table, get a reference to the `ODTables` property for the drawing, and call `Tables.RemoveTable()`.

```
Dim tables As ObjectData.Tables
tables = mapApp.ActiveProject.ODTables
tables.RemoveTable("NewTable")
```

ttaching Object Data

Adding object data to a drawing object requires:

- Creating an empty record
- Initializing the record with correct types for the table
- Setting values for each column
- Attaching the object data by adding the record to the table

Create an empty record using the static method `ObjectData.Record.Create()`. This does not define any fields for the record. Initialize the record, which creates fields of the correct type, using `Table.InitRecord()`.

```
Dim rec As ObjectData.Record
rec = ObjectData.Record.Create()
table.InitRecord(rec)
```

Each `Item` property in the record is of type `Utilities.MapValue`, which is a general-purpose class for storing data of variant types. To set any field, get a reference to the field from the `Record` object using the `Item` property. Assign the value with `MapValue.Assign()`. For example, if `rec` is a record in a table where the second field is of type integer, the following will assign a value of 10 to the field.

```
Dim val As Utilities.MapValue
val = rec(1)
val.Assign(10)
```

Add the record to the table with `Table.AddRecord()` and associate it with an object. This requires a `Record` and either an AutoCAD `DBObject` or `ObjectId` as parameters.

```
newTable.AddRecord(rec, objId)
```

A single drawing object may have more than one object data record in a given table.

Getting Object Data

To get all object data records from a single table for a drawing object:

- Get the `ObjectData.Tables` object for the drawing.
- Get the individual table.
- Get the `ObjectData.Records` collection for the object, using one of the `GetObjectTableRecords()` methods.
- Iterate through the records in the collection.
- Process the fields in each record.

The following example writes the values from `table` for `objId` to the console.

```
Dim fieldDefs As ObjectData.FieldDefinitions = _
  table.FieldDefinitions
Dim recs As ObjectData.Records
recs = table.GetObjectTableRecords(0, objId, _
  Constants.OpenMode.OpenForRead, True)

If (recs.Count() > 0) Then
   For Each rec As ObjectData.Record In recs
      For i As Integer = 0 To rec.Count() - 1
         Dim val As Autodesk.Gis.Map.Utilities.MapValue
         val = rec(i)
         Dim fieldDef As ObjectData.FieldDefinition
         fieldDef = fieldDefs(i)
         acEditor.WriteMessage(
           vbNewLine + fieldDef.Name + ": ")
         Select Case val.Type
            Case Constants.DataType.Character
               acEditor.WriteMessage(val.StrValue)
            Case Constants.DataType.Integer
               acEditor.WriteMessage(val.Int32Value.ToString)
            Case Constants.DataType.Point
               acEditor.WriteMessage("point")
            Case Constants.DataType.Real
               acEditor.WriteMessage(val.DoubleValue.ToString)
```

```
            Case Else
               acEditor.WriteMessage("undefined")
         End Select
      Next
   Next
End If
recs.Dispose()
```

Processing all tables for an object is similar. Instead of calling `Table.GetObjectTableRecords()` for an individual table, call `Tables.GetObjectRecords()` for all tables. When processing the fields, be sure to to get the field definitions from the correct table for the current record.

- Get the `ObjectData.Tables` object for the drawing.
- Get the `ObjectData.Records` collection for the object, using one of the `GetObjectRecords()` methods.
- Iterate through the records in the collection.
- Get the table name for the current record.
- Get the fields definitions for the table.
- Process the fields in each record.

Note When you have finished processing the records, release any of the disposable objects with their `Dispose()` methods. This applies to any classes inheriting `Autodesk.AutoCAD.Runtime.DisposableWrapper`, like `ObjectData.Table`, `ObjectData.Records`, or `Utilities.MapValue`.

Updating and Deleting Records

To update or delete records, they must be opened for write access in the call to `Table.GetObjectTableRecords()` or `Tables.GetObjectRecords()`.

Fields in a record are of type `Utilities.MapValue`. To update a field, get a reference to the value from the `Record` object. Assign a new value using `MapValue.Assign()` and update the record using `Records.UpdateRecord()`. The following example sets the value of the first field in a record:

```
Dim val As Utilities.MapValue = rec(0)
val.Assign(19)
recs.UpdateRecord(rec)
```

To delete a record, get an `IEnumerator` using `Records.GetEnumerator()`. Advance the enumerator to the record to be deleted and call `Records.RemoveRecord()`. The following example deletes the first record for an object.

```
Dim recs As ObjectData.Records
recs = table.GetObjectTableRecords(0, objId, _
  Constants.OpenMode.OpenForWrite, True)

Dim ie As IEnumerator
ie = recs.GetEnumerator()
ie.MoveNext()
recs.RemoveRecord()
recs.Dispose()
```

)ata Connect

Topics in this section

- Overview
- Setting Up Visual Studio
- Creating the Plugin

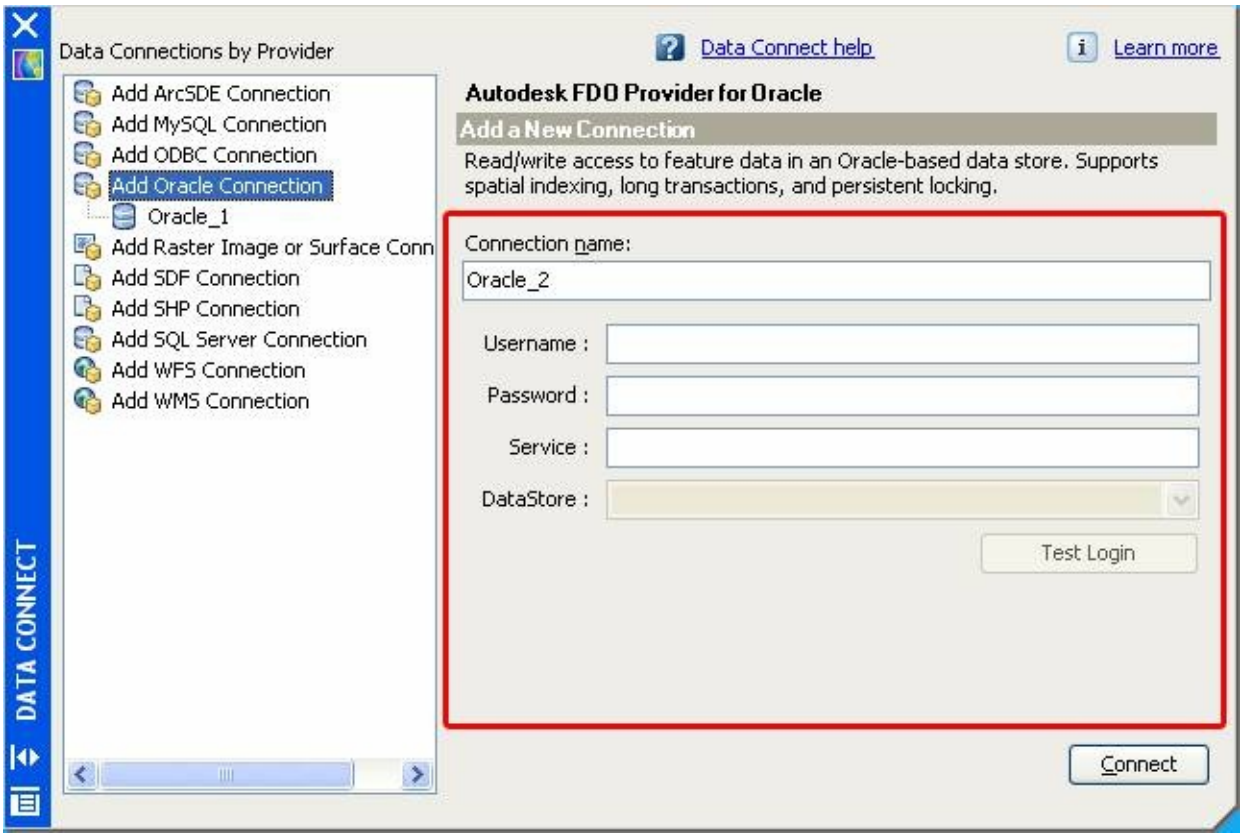
Please send us your comment about this page

Overview

The Data Connect API can be used to create plugins for the Data Connect dialog. Use this to create new connection forms for FDO providers, either providers that are installed with AutoCAD Map 3D or additional providers.

Note The Data Connect API is not part of the Geospatial Platform API. It is, however, included in the technical preview for this release of AutoCAD Map 3D. For that reason it is documented in this guide instead of the .NET Developer's Guide.

If a custom plugin works with a provider installed with AutoCAD Map 3D, it replaces the existing form for the provider. For example, a custom form for the Autodesk Oracle provider could look like the following. The outlined area is the custom form, embedded in the Data Connect dialog.



To add a new option to the Data Connect dialog, complete the following steps:

- Ensure the FDO provider DLL is installed with AutoCAD Map 3D.
- Ensure the FDO provider is listed in *providers.xml*.
- Create the plugin to use the provider.
- Save the plugin DLL in the *Plugins\DataConnect* folder of the AutoCAD Map 3D installation. If the folder does not exist, create it.

providers.xml is located in the *FDO\bin* folder of the AutoCAD Map 3D installation. It contains entries for each available FDO provider.

---

etting Up Visual Studio

To create a project using the Data Connect API, follow the instructions in the AutoCAD Map 3D .NET Developer's Guide. Add the following references to the project:

- `Autodesk.Gis.Plugins.dll`
- `AcMapDataConnectPlugins.dll`

The assemblies are located in the AutoCAD Map 3D installation folder.

Creating the Plugin

Note To ensure the plugin is loaded, place the DLL in the *Plugins\DataConnect* folder under the AutoCAD Map 3D installation folder. Plugins in this folder that follow the proper structure are loaded on demand. There is no need to run the `netload` command.

Using Visual Studio, create a new project. Add a `UserControl` to the project. The control will be embedded in the Data Connect dialog.

The plugin class must implement the `IDataConnectConnectionPlugin` interface. It also requires a `DataConnectPluginAttribute` with the FDO provider name. The provider name must match the name in *providers.xml*. For example:

```
[DataConnectPluginAttribute("Autodesk.Oracle.3.1")]
public partial class SampleProviderConnectUIPlugin
    : UserControl
    , IDataConnectConnectionPlugin
{
```

`IDataConnectConnectionPlugin` provides the necessary methods for AutoCAD Map 3D to interact with the control. It inherits 2 other interfaces: `IDataConnectPlugin` and `IPlugin`.

The implementation for `IDataConnectPlugin` can be simple, as follows. The `IDataConnectPluginHost` interface contains a single property, `HostApplication`, of type `object`. When attached, it contains a reference to the host application object, `Autodesk.AutoCAD.ApplicationServices.Application.AcadA`

```
// IDataConnectPlugin implementation

protected IDataConnectPluginHost _host;
protected string _providerName;
```

```
public void Attach(IDataConnectPluginHost host)
{
    _host = host;
}

public void Detach() { _host = null; }

public IDataConnectPluginHost Host
{
    get { return _host; }
}

public UserControl ClientControl
{
    get { return this; }
}
```

Similarly, the implementation for `IPlugin` needs methods to get and set properties. The properties are used to describe the plugin. They may be displayed to the user and should be localized. The `Dependencies` property is not currently used.

```
// IPlugin implementation

protected string title = "title";
protected string description = "description";
protected string company = "company";
protected string version = "3.0.0";
protected Type[] dependencies;

public string Title
{
  get { return title; }
  set { title = value; }
}

public string Description
{
  get { return description; }
  set { description = value; }
}

public string Company
{
  get { return company; }
  set { company = value; }
}
```

```
public string Version
{
  get { return version; }
  set { version = value; }
}

public Type[] Dependencies
{
  get { return dependencies; }
}
```

`IDataConnectConnectionPlugin` contains methods to get and set the connection parameters for the FDO provider, so the implementation depends on the requirements of the provider. For complete details about the methods, refer to the AutoCAD Map 3D .NET API Reference Supplement. Generally these methods will work with fields on the custom form.

The properties of `IDataConnectConnectionPlugin` return information about the connection parameters for the provider. These are used to open the FDO connection.

Property	Description
ConnectionParameterNames	A list of parameters for the provider.
ConnectionParametersValid	Boolean. True if the current parameters are valid for the provider.
ProviderName	The provider name.

The methods of `IDataConnectConnectionPlugin` get and set parameter values:

Method name	Description
GetConnectionParameter()	Returns the current value of a connection parameter.

SetConnectionParameter()	Sets the value of a connection parameter. This should update the form field for the parameter.
Initialize()	Called when the plugin first loads.
SetConnectionParametersToDefaults()	Sets all connection parameters to default values.

There is also an event, `ParameterValueChanged`, which should fire when a parameter value changes. This event is required so the containing form is able to update the Connect button state based on the validity of the current parameter values.

Annotation

Topics in this section

Overview

Annotations provide a way to label drawing objects, based on data associated with the objects. The associated data can come from various places, such as object data, linked databases, or object properties. For example, if a drawing contains parcels, and the parcels have object data with the name of the parcel owner or the most recent sale price of the parcel, then these values could be added to the map as part of an annotation.

Each annotation is based on an annotation template, which is a special block in the drawing. The template describes the annotation, and usually contains annotation text defining the variable content.

A new annotation is created by inserting a reference to the annotation template. This creates a reference to the annotation block, substituting the proper values in the expression fields.

The classes for handling annotations are mostly within the `Annotation` namespace. Code in this chapter assumes the following:

```
Imports Autodesk.Gis.Map.Annotation
```

Annotation Templates

An annotation template is a special block in the drawing. It defines the fixed and variable parts of the annotation.

Note For more information about blocks, refer to the AutoCAD developer documentation.

Internally, annotation templates use a special naming convention. The names of all blocks containing annotation templates begin with *ACMAP_ANN_TEMPLATE_*. This prefix is defined in the `Annotations.TemplateNameBlockTableRecordPrefix` property.

Annotation API calls, though, use the annotation name as it appears in the UI. For example, an annotation template named *ParcelName* would be stored in a block named *ACMAP_ANN_TEMPLATE_ParcelName*, but it would be created with a call to `Annotations.CreateAnnotationTemplate("ParcelName")`.

The `ProjectModel.Annotations` property returns an `Annotations` object that can be used for managing the annotations and annotation templates.

Attributes

An AutoCAD block can contain attribute definitions, which are text entities that can define informational text for each block reference.

In an annotation template, attribute definitions are used to define the variable parts of the annotation. For example, if the annotations include object data, then attribute definitions in the block template define what object data should appear and how it will be displayed.

The `AttributeDefinition` class inherits the `DBText` class, which inherits the `Entity` class. These have properties such as `Color` and `Height` that define how the annotation will appear in the drawing. For example, to set the

text height for an annotation, set the `Height` property of the attribute definition used for the annotation.

Note The properties used for the attribute definition can also be modified using expression strings. See Expressions in Annotations for details.

Creating an Annotation Template

Although an annotation template is a form of AutoCAD block, it must be created using the Map API or it will not be recognized properly.

- Start a transaction.
- Create the annotation template using `Annotations.CreateAnnotationTemplate()`.
- Get a reference to the annotation template using `Annotations.Item()`.
- If required, set block properties for the annotation template using `AnnotationTemplate.SetExpressionString()`. For example, this can be used to rotate the block reference to match the rotation of the object being annotated.
- If required, add fixed drawing objects to the annotation template. Get the AutoCAD block id using the `AnnotationTemplate.BlockDefinitionId` property and add drawing objects to the template using standard AutoCAD API calls.
- Add variable annotation text to the template using `AnnotationTemplate.CreateAnnotationText()`. This creates an attribute definition in the block.
- Set the display properties of the annotation text by setting properties for the attribute definition.
- Set the expression string for the annotation text using `Annotations.SetExpressionString()`.
- Commit the transaction.

`Annotations.CreateAnnotationTemplate()` creates an empty template. It returns an AutoCAD `ObjectID` that is the id of the block table

record. Get a reference to the annotation template object using `Annotations.Item()`.

```
Dim annotations As Annotation.Annotations = _
   activeProj.Annotations

Dim trans As Transaction = Nothing
trans = activeDoc.TransactionManager.StartTransaction()
Dim newTemplateId As ObjectId = _
   annotations.CreateAnnotationTemplate("templateName")

Dim newTemplate As Annotation.AnnotationTemplate = _
   annotations(newTemplateId)
```

If required, set block properties for the annotation template. See Expressions in Annotations for details.

```
newTemplate.SetExpressionString(_
   Annotation.AnnotationExpressionFields.BlockRotation, ".ANGLE")
```

Add objects to the template. They can be normal drawing objects or annotation text.

To add normal drawing objects, use standard AutoCAD methods.

```
Dim line As New Line
line.StartPoint = New Geometry.Point3d(0.0, -0.6, 0.0)
line.EndPoint = New Geometry.Point3d(2.0, -0.6, 0.0)
Dim blockTableRec As BlockTableRecord
blockTableRec = newTemplateId.GetObject(OpenMode.ForWrite)
blockTableRec.AppendEntity(line)
trans.AddNewlyCreatedDBObject(line, True)
```

To add annotation text, create an annotation text object. This is a special type of AutoCAD attribute definition. `AnnotationTemplate.CreateAnnotationText()` returns the AutoCAD ObjectId of the attribute definition. Open this object for writing and cast to an `AttributeDefinition` object:

```
Dim expressionTextId As ObjectId
expressionTextId = newTemplate.CreateAnnotationText()
Dim attDef As AttributeDefinition
attDef = _
  CType(trans.GetObject(expressionTextId, OpenMode.ForWrite), _
  AttributeDefinition)
```

Most of the properties for the annotation template can be set using the `AttributeDefinition` properties. For example:

```
attDef.Position = _
  New Autodesk.AutoCAD.Geometry.Point3d(0.0, 0.0, 0.0)
attDef.Tag = "testTag"
attDef.Height = 0.5
attDef.VerticalMode = TextVerticalMode.TextVerticalMid
attDef.HorizontalMode = TextHorizontalMode.TextCenter
attDef.AlignmentPoint =
  New Autodesk.AutoCAD.Geometry.Point3d(0.0, 0.0, 0.0)
```

The annotation text must be set using `Annotations.SetExpressionString()`. See Expressions in Annotations for details.

xpressions in Annotations

Use expressions to set the text or the display properties of the annotation. Some of the items that can use expressions are:

- Annotation text
- Text color
- Text size
- Rotation angle
- Position relative to the drawing object being annotated

Note Properties in an attribute definition can be overridden by annotation expressions. For example, if the attribute definition defines the location of the text, the annotation expression could override it.

`AnnotationTemplate.SetExpressionString()` sets properties for the entire template. `Annotations.SetExpressionString()` sets properties for annotation text within the block.

Expressions are evaluated by the AutoLISP interpreter, and return a single value. Depending on the property being set, the value can be numeric or string. If the expression cannot be evaluated properly it displays the attribute tag name instead.

Note For more details about expressions, including a list of functions and variables, see the Map 3D Help. In the Reference Guide section there is a chapter about the Expression Evaluator.

The enum `Annotation.AnnotationExpressionFields` contains the complete list of fields that can use expressions.

In most cases, expressions are used to define the text of the annotation, but they can also be used to define things like color, size, or position.

Example

To set the annotation text based on object data, use the syntax :*fieldname@tablename*. For example:

```
Imports Autodesk.Gis.Map.Annotation
annotations.SetExpressionString(attDef, _
AnnotationExpressionFields.AttributeDefinitionAnnotationString, _
  ":PARCEL_OWNER@ParcelData")
```

nserting Annotations

To insert an annotation, call one of the `AnnotationTemplate.InsertReference()` methods. They all require an `ObjectId` or `ObjectIdCollection` as parameter, to identify the drawing objects to be annotated.

This creates a block reference in the drawing. It evaluates the annotation expressions and uses the results to set the text or other properties of the reference.

An inserted annotation reference can also have overrides that change the display properties. See Annotation Overrides for details.

Updating and Refreshing Annotations

Once inserted, annotation references do not change unless they are explicitly changed. For example, if the object data for a drawing object changes, any anotations that use the object data will still display the original value.

There are two operations for revising existing annotation references:

- Updating
- Refreshing

Updating removes and recreates all the annotations that use a template. Refreshing re-evaluates the annotation expressions, but does not remove out-of-date annotations.

To update annotations, call `AnnotationTemplate.UpdateReferences()`.

```
newTemplate.UpdateReferences(True)
```

To refresh annotations, call `AnnotationTemplate.RefreshReferences()`.

```
newTemplate.RefreshReferences(True)
```

nnotation Overrides

An annotation override can be applied when an annotation reference is created. It changes selected properties of the annotation template. For example, an annotation override can change the color or text size of the annotation.

Annotation overrides can apply to the static properties of the annotation, which are set using the `AttributeDefinition` properties, or the dynamic properties, which are set using expressions.

Annotation overrides correspond to the Insert Options and Insert Properties of the Insert Annotation dialog in the UI.

For example, to override the static color, set the `ColorOverride` property of the annotation override. To override a color set using an expression, set the `ColorExpressionOverride` property.

To apply an annotation override, insert the annotation using `AnnotationTemplate(ObjectId, AnnotationOverrides)`. For example:

```
Dim annOverrides As New Annotation.AnnotationOverrides
annOverrides.Clear()
Dim greenClr As Autodesk.AutoCAD.Colors.Color = _
  Autodesk.AutoCAD.Colors.Color.FromColorIndex( _
  Autodesk.AutoCAD.Colors.ColorMethod.None, 3)
annOverrides.ColorOverride = greenClr

annTemplate.InsertReference(objId, annOverrides)
```

vents

Topics in this section

- Overview
- Events in the API Reference
- List of Events

Overview

Events and event handlers provide a way for applications to respond to changes in the Map application. For example, opening a new project can fire an event handler to perform additional processing.

The API uses standard .NET mechanisms for handling events. Applications wanting to handle events subscribe to the events. When the event is fired all handlers subscribed to that event are called.

Event handlers accept two parameters:

- A reference to the object raising the event
- Event arguments

The class definition for the event arguments is usually specific to the event being handled.

vents in the API Reference

For every event, the AutoCAD Map 3D API Reference contains the following:

- Class definition for the event arguments. The names of these classes usually begin with the event name and end with “EventArgs”. In some cases the event uses `System.EventArgs` instead of defining a new class.
- Methods for adding and removing event handlers. The names of these methods begin with “add_” or “remove_”. Do not call these methods directly. Instead use the correct syntax for the language.
- Type definition for the event handler.
  Note Some events, such as `ProjectModel.BeginClose`, use `System.EventHandler` and `System.EventArgs` instead of objects derived from them. For details refer to the API reference or the Visual Studio Object Browser.

For example, the `ProjectOpened` event in the `Autodesk.Gis.Map` namespace consists of the following:

- `ProjectOpenedEventArgs` class
- `add_ProjectOpened` method in the `MapApplication` class
- `remove_ProjectOpened` method in the `MapApplication` class
- `ProjectOpenedEventHandler` type

Note The actual event name is not used in the API reference. It can always be inferred from the corresponding `add_` or `remove_` methods.

Example: VB.NET

To define an event handler for the `ProjectOpened` event, create a subroutine:

```
Sub handleProjectOpened(ByVal pSender As Object, _
ByVal pArgs As ProjectOpenedEventArgs)
   ' Insert code to handle event
End Sub
```

To subscribe to the event:

```
AddHandler mapApp.ProjectOpened, AddressOf handleProjectOpened
```

To unsubscribe from the event:

```
RemoveHandler mapApp.ProjectOpened, AddressOf handleProjectOpened
```

Example: C#

To define an event handler for the `ProjectOpened` event, create a subroutine:

```
void handleProjectOpened(Object sender,
ProjectOpenedEventArgs args)
{
   // Insert code to handle event
}
```

To subscribe to the event:

```
mapApp.ProjectOpened += new ProjectOpenedEventHandler(
   handleProjectOpened);
```

To unsubscribe from the event:

```
mapApp.ProjectOpened -= new ProjectOpenedEventHandler(
   handleProjectOpened);
```

ist of Events

Namespace Autodesk.Gis.Map

Aliases class

Event name	Description
AlliasAdded	Fired when a new alias is added to the application. The event args pass the alias name.
AliasDeleted	Fired when an alias is deleted from the application. The event args pass the alias name and path of the deleted alias.

MapApplication class

Event name	Description
IntOptionModified	Fired when one of the application options is modified. The event args pass the option name and the new and old values of the option.
ProjectBeginClose	Fired when a project begins the close operation, but before the project has been closed. The event args pass

	the project model.
ProjectCreated	Fired after a new project has been created. The event args pass the project model.
ProjectOpened	Fired when an existing project has been opened. The event args pass the project model.
StringOptionModified	Fired when one of the application options is modified. The event args pass the option name and the new and old values of the option.
UnloadApp	Not used for .NET applications.

Namespace Autodesk.Gis.Map.Classification

ClassificationManager class

Event name	Description
FeatureClassDefinitionCreated	Fired when a new feature class definition has been created. The event args pass the class name and the name of the XML file containing the class definition.
FeatureClassDefinitionDeleted	Fired when a feature class definition has been deleted. The event args pass the class name and the name of

	the XML file containing the class definition.
FeatureClassDefinitionModified	Fired when a feature class definition has been modified. The event args pass the class name and the name of the XML file containing the class definition.
FeatureClassDefinitionRenamed	Fired when a feature class definition has been renamed. The event args pass the new and old class names and the name of the XML file containing the class definition.
FeatureDefinitionFileAttached	Fired when a new definition file is attached. The event args pass the filename.
FeatureDefinitionFileModified	Fired when a definition file is modified. The event args pass the filename.

Namespace Autodesk.Gis.Map.DisplayManagement

DisplayManager class

Event name	Description
CategoryAppended	
CategoryModified	
CategoryUnappended	

MapAppended	
MapGoodBye	
MapSetCurrentBegin	Fired when changing the current display manager map, before the change is made. Returns
MapSetCurrentEnd	
MapSetCurrentFails	
MapUnappended	
StyleAppendedToCategory	
StyleModified	
StyleUnappended	

Map class

Event name	Description
CurrentScaleModified	
DismissStylizationBegin	
DismissStylizationCancel	
DismissStylizationEnd	
ItemAppended	
ItemErased	
ItemModified	
ScaleAdded	
ScaleErased	

ScaleModified	
StyleAppended	
StyleErased	
StyleReferenceAppended	
StyleReferenceErased	
StyleReferenceModified	
UpdateStylizationBegin	
UpdateStylizationCancel	
UpdateStylizationEnd	

Namespace Autodesk.Gis.Map.ImportExport

Exporter class

Event name	Description
ExportRecordError	
RecordExported	
RecordReadyForExport	

Importer class

Event name	Description
ImportRecordError	
RecordImported	
RecordReadyForImport	

Namespace Autodesk.Gis.Map.MapBook

BookManager class

Event name	Description
MapBookAppended	
MapBookErased	
MapBookModified	
MapBookSetCurrent	
MapBookTileModified	
MapBookTileWillBeErased	
MapBookTreeNodeModified	
MapBookWillBeErased	

Namespace Autodesk.Gis.Map.Project

DrawingSet class

Event name	Description
DrawingActivated	
DrawingActivationCancelled	
DrawingAttachCancelled	
DrawingAttached	
DrawingDeactivated	
DrawingDetached	
DrawingSettingsModified	

DrawingToBeActivated	
DrawingToBeAttached	

ProjectModel class

Event name	Description
AbortCSChange	
AbortSwapId	
BeginClose	
BeginCSChange	
BeginDestroy	
BeginOpen	
BeginQuery	
BeginSave	
BeginSaveBack	
BeginSwapId	
BeginTransform	
EndClose	
EndCSChange	
EndOpen	
EndQuery	
EndSave	
EndSaveBack	
EndSwapId	

EndTransform	
IntOptionModified	
StringOptionModified	

Namespace Autodesk.Gis.Map.Query

QueryLibrary class

Event name	Description
QueryAdded	
QueryCategoryAdded	
QueryCategoryDeleted	
QueryCategoryRenamed	
QueryDeleted	
QueryModified	
QueryRenamed	

Drawing Sets

Topics in this section

Overview

Drawing sets provide a way for a single map to combine objects from multiple drawings. One drawing, the project drawing, can attach multiple source drawings. The source drawings, in turn, can attach other source drawings to form a tree of attached drawings.

Running a query on the attached drawings copies selected objects into the project drawing, where they can be displayed and edited. Unless an object from an attached drawing has been “queried in” it does not appear in the project drawing. See Queries and Save Sets for more details.

)rive Aliases

Attached drawings are often shared between different users on different computers. Because of this, the paths to the attached drawings can be different for each user. Aliases help manage these files.

Each alias maps an alias name to a directory path. Each AutoCAD Map 3D user can define different paths for the aliases. The locations of attached drawings are always identified using the aliases, so users can have different paths to the atttached drawings, as long as the aliases are the same.

The `DriveAlias` class in the `Autodesk.Gis.Map` namespace represents an individual alias. It has two properties: `Name` and `Path`.

The `Aliases` property of the map application returns an `Aliases` object for managing the aliases in the session. `Aliases.Item()` returns an individual drive alias, either by alias name or index number.

```
Dim aliasList As Aliases
aliasList = mapApp.Aliases
```

`Aliases` objects have methods for adding and removing aliases, and event handlers for detecting when aliases have been added or removed.

ttaching and Detaching Drawings

Attaching a drawing adds it to the drawing set for a project. Detaching a drawing removes it from the drawing set.

To attach a drawing, use `DrawingSet.AttachDrawing()`. Pass a single string argument that contains the alias and the path to the drawing to attach. The form is:

```
alias:\filename
```

This returns a reference to the attached drawing, an `AttachedDrawing` object.

When an attached drawing is activated, the file is locked against editing by other applications. To remove the lock, but keep the drawing attached, call `AttachedDrawing.Deactivate()`. To reactivate the drawing, call `AttachedDrawing.Activate()`.

Queries and Save Sets

Topics in this section

Overview

Queries and save sets work on attached drawings. A query copies drawing objects from attached drawings into the project drawing. Once in the project drawing, the objects can be edited like any other drawing object.

A save set is a list of objects in the project drawing that are to be updated in attached drawings. The save set can contain:

- objects that have been modified in the project drawing that should also be modified in the attached drawings
- objects that have been deleted from the project drawing and should be deleted from the attached drawings
- new objects that have been added to the project drawing that should also be added to an attached drawing

Objects that have been queried into the project drawing are not added to the save set automatically.

Queries

A query is a tree structure containing branches (`QueryBranch` objects) and conditions (`DataCondition`, `LocationCondition`, `PropertyCondition`, and `SqlCondition` objects). All of these objects are subclassed from `QueryUnit`.

To create a query, call `ProjectModel.CreateQuery()`.

```
query = activeProj.CreateQuery()
```

This returns an empty `QueryModel` object.

A simple query can have a root branch with a single condition. More complex queries combine branches and conditions.

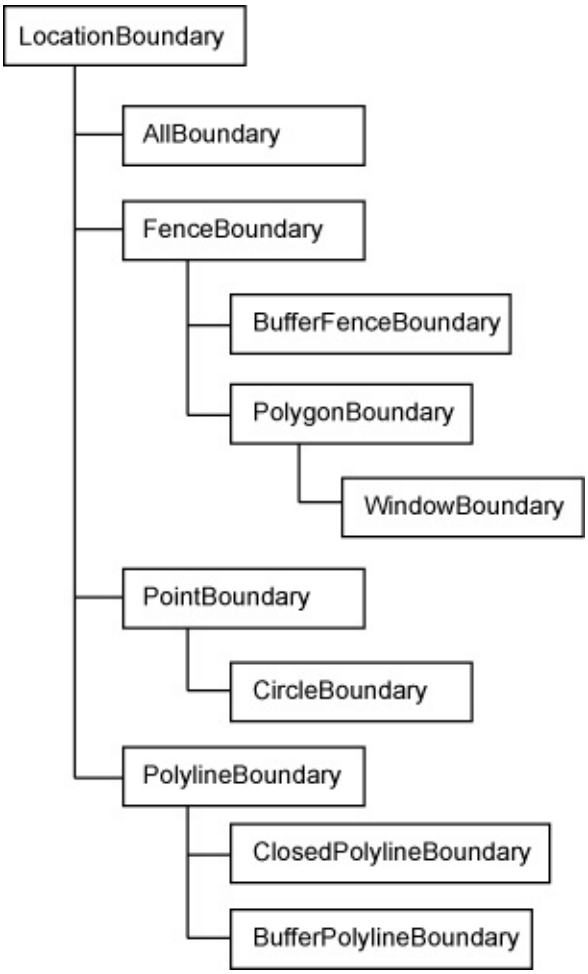
The criteria that the query uses to select objects are expressed in query conditions. There are four types of query conditions.

	Description
Location Conditions	Based on the location of objects relative to a boundary. There are several boundary types. See Location Boundaries below.
Property Conditions	Based on a particular AutoCAD property.
Data Conditions	Based on object data. To query object data set the query type to `DataIrd`. To query object classes set the

	query type to `DataFeature`.
SQL Conditions	Based on data about drawing objects that is stored in external database tables and is specified by the WHERE clause of a SQL query.

Location Boundaries

There are several types of location boundaries. They are all represented by descendents of the `LocationBoundary` class, as illustrated in the following diagram.



Executing Queries

To build a query

1. Create the query using `ProjectModel.CreateQuery()`.
2. Create one or more query conditions using the constructors for `DataCondition`, `LocationCondition`, `PropertyCondition`, and `SqlCondition`.
3. Create one or more query branches using the constructor for `QueryBranch`.
4. Build the query tree by appending query conditions and branches using `QueryBranch.AppendOperand()`.
5. Create the query definition by passing the root query branch to `QueryModel.Define()`.
6. Optionally, save the query definition in an external file or query library.

To execute a query

1. You may want to set the mode, enable or disable property alteration, or create a report template for the query.
2. Call `QueryModel.Run()` to execute the query against all attached drawings. Call `QueryModel.Execute()` to execute the query for a particular drawing set.

The query mode determines if the objects from the attached drawings are displayed as a preview or cloned into the project drawing. `QueryModel.Execute()` returns a list of objects that have been queried in.

Query Libraries

Query libraries provide a way to save and re-use queries. The queries can be saved internally in the project file or externally. Each project has its own library, available with `ProjectModel.QueryCategories`.

Libraries are divided into categories, where each category is a container for saved queries. The saved queries are represented by `QueryAttribute` objects.

To add a query to the library, first add a category then add the query.

```
Dim queryLib As Query.QueryLibrary = activeProject.QueryCategories
Dim queryCat As Query.QueryCategory
queryCat = queryLib.AddCategory("catName")
queryCat.AddQuery("queryName", "query description", queryObj)
```

To save a query to an external file, call `QueryModel.Save()`.

ave Sets

Each project has a save set, which manages a list of object ids to be updated in attached drawings.

Objects are not added to the save set automatically. It is up to the application to add objects using `SaveSet.AddObjects()`.

To save objects back to an attached drawing, call `AttachedDrawing.CloneBack()`, `AttachedDrawing.CloneBackArea()`, or `AttachedDrawing.CloneBackLayer()`.

Classification

Topics in this section

Overview

Object classification is a way of tagging drawing objects with an object class name. This can help organize the objects in the drawing, and enforce drawing standards.

Note For historical reasons, the API uses the terms *feature* and *feature class* for object classification. These are different from FDO features and feature classes. It is important not to confuse the two.

Managing Feature Definition Files

Feature class definitions are stored in XML files, external to the drawing file. Before definitions from a file can be used, the file must be attached to a drawing. There can only be one attached feature definition file active at one time for a given project.

Note In the UI, feature definition files are called object class definition files.

Creating Feature Definition Files

To create a feature definition file:

- Get the classification manager object for the project using `ProjectModel.ClassificationManager`.
- Create the new file using `ClassificationManager.CreateFeatureDefinitionFile` Note that the current user must have administrative privileges in the Map session. To check, test `ClassificationManager.CanCurrentUserAlterSchema`.

```
Dim classMgr As Classification.ClassificationManager
classMgr = activeProj.ClassificationManager
If (classMgr.CanCurrentUserAlterSchema) Then
   classMgr.CreateFeatureDefinitionFile(filename)
Else
   ' Error
End If
```

Attaching and Detaching Feature Definition Files

To attach a feature definition file, call `ClassificationManager.AttachFeatureDefinitionFile()`.

To detach the current file, call

ClassificationManager.DetachCurrentFeatureDefinitionFi

Creating Feature Class Definitions

Feature class definitions are composed of properties that define how classified objects will appear. Each feature class definition can only be used with certain types of drawing objects.

To create a feature class definition, start with a drawing object to use as a template. Get the properties of the drawing object using `ClassificationManager.GetProperties()`. The current values of the drawing object become the default property values.

Determine the list of object types that the feature class definition can be used with. This list can be expressed as a string collection or a collection of AutoCAD `RXClass` objects. Create the empty feature class definition using `ClassificationManager.CreateFeatureClassDefinition()`. Set a drawing object type to use for creating new instances of the class using `FeatureClassDefinition.SetCreateMethod()`.

```
Dim classMgr As Classification.ClassificationManager
Dim newDef As Classification.FeatureClassDefinition
classMgr = activeProj.ClassificationManager

Dim trans As Transaction = Nothing
Dim obj As DBObject = Nothing
Dim cls As RXClass = Nothing
Try
   trans = _
      MdiActiveDocument.TransactionManager.StartTransaction()
   obj = trans.GetObject(objId, OpenMode.ForRead)
   cls = obj.GetRXClass()
   trans.Commit()
Finally
   trans.Dispose()
End Try
Dim entType As System.String = System.String.Copy(cls.Name)
Dim entTypesCol As StringCollection = New StringCollection()
entTypesCol.Add(entType)
newDef = classMgr.CreateFeatureClassDefinition( _
```

```
   defName, Nothing, entTypesCol, Nothing, False)
newDef.SetCreateMethod(entType, "")
```

Use the `FeatureClassPropertyCollection` as an initial set of properties for the feature class definition. Modify it as needed by setting range and default values for the properties in the collection. Create a new `FeatureClassPropertyCollection` with the updated properties. Save the feature definition file.

```
Dim classProp As Classification.FeatureClassProperty
Dim propCollection As _
   Classification.FeatureClassPropertyCollection
propCollection = _
   New Classification.FeatureClassPropertyCollection
classMgr.GetProperties(classPropCollection, Nothing, objId)

For Each classProp In classPropCollection
   ' Modify the property if necessary
   newDef.AddProperty(classProp)
Next

classMgr.SaveCurrentFeatureDefinitionFile()
```

lassifying Objects

To classify a drawing object, call `ClassificationManager.Classify()`. This tags the object with the name of the feature definition file and the feature class name. If the feature definition file is detached, the classification tag remains.

A single object may be classified more than once, by using feature classes from different feature definition files. To get a list of all classifications for an object, call `ClassificationManager.GetAllTags()`.

To unclassify an object, call `ClassificationManager.Unclassify()`.

To get a list of all objects in a drawing that have not been classified, call `ClassificationManager.GetUnclassifiedEntities()`. The result of this can be used to find and classify missing objects.

ilters

Topics in this section

Overview

Filters provide a simple mechanism for selecting drawing objects that meet certain criteria. In the AutoCAD Map 3D UI, a basic filter can be used to select drawing objects for an export operation.

A basic filter has options for filtering based on combinations of layer names, object classification, and block names. Custom filters can filter based on other criteria. Both types are based on the `Filters.ObjectFilter` class.

An `ObjectFilterGroup` can combine multiple filters into a single filter operation.

Objects for working with filters are in the `Autodesk.Gis.Map.Filters` namespace.

asic Filters

The `FilterObjects()` method for any filter takes a list of drawing objects to be filtered and returns a list of drawing objects that meet the filtering criteria.

To create a basic filter, call the constructor with a list of layer names, feature class names, and block names. Separate multiple values with commas. An asterisk (“*”) wild card selects all objects matching the criterion.

```
Dim newFilter As Autodesk.Gis.Map.Filters.BasicFilter
newFilter = New Autodesk.Gis.Map.Filters.BasicFilter
   ("Parcels, Lots", "*", "*"))
```

To run a filter, call its `FilterObjects()` method with the output and input `ObjectIdCollection` objects.

Custom Filters

Custom filters can implement additional filtering capabilities. To create a custom filter, define a new class based on the `ObjectFilter` class. At a minimum, the custom filter must define a `FilterObjects()` method. It can define any other methods needed for creating and modifying the filter.

nport/Export

Topics in this section

Overview

The AutoCAD Map 3D application maintains lists of available import and export formats. It is not possible to modify these lists using the API. The application `Importer` and `Exporter` classes can import and export data using these formats.

Any import or export procedure requires the following:

- Selecting the external file format and location
- Mapping attribute data from the external file to object data
- Setting any necessary driver options
- Importing or exporting

The particular options will vary depending on the type of data being imported or exported, but the overall procedure is the same.

nporting

The import procedure brings in objects from external files and creates new drawing objects. Some external files have a single layer, while others have more than one.

To start an import process, get the `Importer` object for the application and initialize it with the import format name and the location of the file or files to import. For example::

```
myImporter.Init("SHP", fileName)
```

This sets up the available layers for the import. Iterate through the layers. For each layer, determine if objects from the layer will be imported. Set `InputLayer.ImportFromInputLayerOn`.

Any attribute data attached to objects in the layers can be brought into the drawing as object data. Each attribute in the source file corresponds to a column in the input layer. These columns can be mapped to fields in object data tables.

To map input columns, set the object data table name using `InputLayer.SetDataMapping()`. Then iterate through each column, setting the mapping for the column using `Column.SetColumnDataMapping()`.

Different import drivers will have different options. The default options are stored in *MapImport.ini*. To modify the options, get the options using `Importer.DriverOptions()`. This returns a name-value collection. Modify the options in the collection and call `Importer.SetDriverOptions()`.

Some drivers also have an options dialog. For those drivers, call `Importer.InvokeDriverOptionsDialog()` to have the user set the options.

When all the mappings and driver options have been set up properly, call `Importer.Import()`. This returns an `ImportResults` object, which contains details of the import.

Import Events

The `RecordReadyForImport` event handler can be used to control which records are imported. The handler is fired for every record, before the import has been completed. To stop the import for a record, set `RecordReadyForImportEventArgs.ContinueImport` to false.

---

xporting

Exporting is similar to importing, with some small differences.

To select which drawing objects should be exported, call `Exporter.SetSelectionSet()`. To export all drawing objects, set `Exporter.ExportAll` to true. To filter the list of objects, set the `Exporter.FeatureClassFilter` or `Exporter.LayerFilter` properties.

The `Exporter` object does not have a property corresponding to `InputLayer`. The mappings for attribute data are set using `Exporter.SetExportDataMappings()`. This requires an `ExpressionTargetCollection` parameter as input.

Items in an `ExpressionTargetCollection` object are name-value pairs, where the name corresponds to an object data expression and the value is the attribute name in the exported file. For details about object data expressions see Expressions in Annotations or the Map 3D Help.

Export Events

The `RecordReadyForExport` event handler is similar to `RecordReadyForImport`. To stop the export of a record, set `RecordReadyForExport.ContinueExport` to false.

opology

Topics in this section

Overview

A network topology contains a set of edges or links. Each link has a node at each end. Multiple links can intersect at a single node.

A polygon topology represents an area coverage.

Topologies describe relationships between drawing objects. There are three types of topology:

- Node, also called point
- Network
- Polygon

A node topology contains a set of points.

A network topology contains a set of edges or links. Each link has a node at each end. Multiple links can intersect at a single node.

A polygon topology represents an area coverage. The borders of polygons are represented by edges. The polygons in a polygon topology cannot overlap, but adjacent polygons share edges.

Each object in the topology (node, link, or polygon centroid) has an ID number that is unique within the topology.

Note The topology is related to drawing objects, but it is stored independently. It is possible to have a topology where the nodes do not correspond to drawing objects.

Internally, the relationship between drawing objects and topologies is implemented using object data tables. For a topology named *`topol_name`*, the following tables are used:

- `TMPCNTR_`*`topol_name`*

- TPMDESC_*topol_name*
- TPMID_*topol_name*
- TPMLINK_*topol_name*
- TPMNODE_*topol_name*

TPMDESC and TPMID are not attached to any drawing objects. They are used to store information about the topology itself. TPMDESC contains the parameters used to create the topology, such as topology type, colors, and layer names. TPMID contains a single value for the last id assigned for the topology.

TPMNODE data is attached to nodes in the topology. Each node has an ID and a resistance value.

TPMLINK data is attached to links between nodes. For network topologies the link has values for the ID, start and end node, direction of the link, and resistance values for traversing the link in each direction. For polygon topologies the link also has values for the polygons on either side of the link.

TPMCNTR data is attached to the centroids of polygons in a polygon topology. Each centroid has values for the ID, area, perimeter, and number of links that form the edges of the polygon.

In most cases, applications do not need to manage the object data directly. The topology API calls perform all the necessary updates. An application needing to know which topologies have been defined in the drawing, however, should check the object data tables for names beginning with “TMPDESC_”.

Drawing Cleanup

Drawing cleanup is essential for polygon and network topologies. It ensures that the objects in the topology can be connected properly. For more details about the various types of cleanup actions, refer to the UI documentation.

A drawing cleanup operation involves combining one or more cleanup actions. Each action is identified by an action number. Many of the actions have additional settings.

Action	Description	Settings
1	Erase Short Objects	CLEAN_TOL
2	Break Crossing Objects	
4	Extend Undershoots	CLEAN_TOL ?? break target
8	Delete Duplicates	CLEAN_TOL INCLUDE_LINEAROBJS INCLUDE_POINTS INCLUDE_BLOCKS INCLUDE_TEXT INCLUDE_MTEXT INCLUDE_ROTATION INCLUDE_ZVALUES
16	Snap Clustered	CLEAN_TOL INCLUDE_POINTS

	Nodes	INCLUDE_BLOCKS SNAP_TO_NODE
32	Dissolve Pseudo Nodes	
64	Erase Dangling Objects	CLEAN_TOL
128	Simplify Objects	CLEAN_TOL ???create arcs
256	Zero Length Objects	
512	Apparent Intersection	CLEAN_TOL
1024	Weed Polylines	WEED_DISTANCE WEED_ANGLE WEED_SUPPLEMENT_DISTANCE WEED_SUPPLEMENT_BULGE

The same class, `Topology.Variable`, is used for both actions and settings. To create a drawing cleanup action, create a settings variable and set its values:

```
Dim toleranceVal As New DatabaseServices.TypedValue _
(Autodesk.AutoCAD.DatabaseServices.DxfCode.Real, 25.5)
Dim toleranceSetting As New DatabaseServices.ResultBuffer
toleranceSetting.Add(toleranceVal)

Dim blocksVal As New DatabaseServices.TypedValue _
(Autodesk.AutoCAD.DatabaseServices.DxfCode.Int16, 1)
Dim blocksSetting As New DatabaseServices.ResultBuffer
blocksSetting.Add(blocksVal)

Dim settings As New Topology.Variable
settings.Set("CLEAN_TOL", toleranceSetting)
settings.Set("INCLUDE_BLOCKS", blocksSetting)
```

Create an action variable and add the action and its settings:

```
Dim action As New Topology.Variable
action.InsertActionToList(-1, 8, settings)
```

If the operation will include more than one action, repeat the process and insert more actions and their corresponding settings into the same action variable.

To perform the cleanup, create a `TopologyClean` object and initialize it with the action variable and a set of drawing objects to clean.

```
Dim cleanObj As New Topology.TopologyClean
cleanObj.Init(action, Nothing)
```

Each individual action in the action variable is a cleanup group. Start the cleanup and go through the groups until all actions have been completed. Commit the changes using `TopologyClean.End()`.

```
cleanObj.Start()
cleanObj.GroupNext()
Do While Not cleanObj.Completed
   cleanObj.GroupFix()
   cleanObj.GroupNext()
Loop
cleanObj.End()
```

For finer control over the objects being cleaned, step through the errors in a group using `TopologyClean.ErrorCur()`. Fix or ignore each one individually. Set `TopologyClean.ErrorPoint` to change the location for the fix.

To save a profile for later use, call `Variable.SaveProfile()` using an action variable object. To reload the profile, call `Variable.LoadProfile()`.

Creating Topologies

To create a new topology, get the `Topologies` object for the project. Select the drawing objects to include in the topology. Call `Topologies.Create()`. Get a reference to the newly created topology using `Topologies.Item()`, which takes a string parameter.

Once a topology has been created, it must be opened using `TopologyModel.Open()`. When the topology is no longer needed, close it with `TopologyModel.Close()`.

lode Topology

A node topology represents a group of related points. Node topologies are often used as part of network or polygon topologies, to represent the endpoints of the links in the topology.

`TopologyModel.GetNodes()` returns the collection of nodes. For each node, `Node.Entity` returns the associated drawing object. If the node does not have a drawing object associated with it, `Node.Entity` throws a `MapException`.

Note Do not update items in a `NodeCollection` object using methods like `Add()`, `Insert()`, and `Remove()`. Instead, call the appropriate methods for the `TopologyModel` object, like `AddPointObject()`.

Call `NodeCollection.Dispose()` when the object is no longer needed.

Network Topology

A network topology represents a group of related nodes and the connections between the nodes. The connections between nodes are links or edges in the topology.

Each full edge is composed of two half edges, representing travel in opposite directions between the nodes. Each half edge can have its own resistance value, which is used in certain types of network analysis.

`TopologyModel.GetFullEdges()` returns the collection of full edges. For each edge, `FullEdge.Entity` returns the associated drawing object. If the edge does not have a drawing object associated with it, `FullEdge.Entity` throws a `MapException`.

`FullEdge.GetHalfEdge` returns a half edge, in either the forward or backward direction.

olygon Topology

A polygon topology represents an area coverage, with polygons inside the area bounded by edges. Any polygon in the topology must be entirely enclosed within its edges.

Each polygon must have a centroid.

`TopologyModel.GetPolygons()` returns the collection of polygons. For each edge, `FullEdge.Entity` returns the associated drawing object. If the edge does not have a drawing object associated with it, `FullEdge.Entity` throws a `MapException`.

Display Manager

Topics in this section

Overview

Display Manager provides a way to organize and style layers in AutoCAD Map 3D.

Note The Display Manager API only manages layers that contain drawing objects. To manage layers containing FDO data use the Geospatial Platform API. See the Geospatial Platform Developer's Guide and the Geospatial Platform API Reference for details.

Each project has its own map manager, which is represented by a `DisplayManagement.MapManager` object. To get the map manager for a project, perform the following steps:

```
trans = activeDoc.TransactionManager.StartTransaction()
Dim managerId As ObjectId
Dim manager As MapManager = Nothing

managerId = DisplayManager.Create(activeProject).MapManagerId( _
  activeProject, True)
manager = trans.GetObject(managerId, OpenMode.ForRead)
```

A single project can contain multiple maps. `MapManager.GetEnumerator()` returns an enumerator that steps through the maps in the project. The properties `MapManager.CurrentMapId` and `MapManager.Current` get or set the current map.

A map (`DisplayManagement.Map` object) is sub-classed from `DisplayManagement.Group`, which represents any group of elements in Display Manager. Maps can contain layers and more groups.

lements

Display Manager elements represent the different layers in the map. Elements can be of the following types, all subclassed from `DisplayManagement.Element`:

- `DisplayManagement.BaseElement`
- `DisplayManagement.LayerElement`
- `DisplayManagement.AttachedDwgsQueryElement`
- `DisplayManagement.TopologyElement`
- `DisplayManagement.TopologyQueryElement`
- `DisplayManagement.FeatureElement`
- `DisplayManagement.RasterElement`

The following iterates through the elements in a map:

```
Dim iterator As IEnumerator = currentMap.NewIterator(True, True)

Dim elementType As Type = GetType(DisplayManagement.Element)
Dim groupType As Type = GetType(DisplayManagement.Group)

Do While (iterator.MoveNext())
  Dim itemId As ObjectId = iterator.Current
  Dim thisItem As Object = _
    trans.GetObject(itemId, OpenMode.ForRead)
  If (thisItem.GetType().Equals(elementType) Or _
      thisItem.GetType().IsSubclassOf(elementType)) Then
    Dim mapElement As Item = thisItem
    ' Process element (layer)
  ElseIf (thisItem.GetType().Equals(groupType)) Then
    Dim thisGroup As DisplayManagement.Group = thisItem
    ' Process group
  Else
    ' Not a Display Manager object, probably contains FDO data
```

```
  End If
Loop
```

The `BaseElement` layer represents the Map Base.

`LayerElement` objects display drawing objects from an AutoCAD layer.

`AttachedDwgsQueryElement` objects represent layers containing data queried in from attached drawings.

`TopologyElement` objects represent Display Manager layers that contain topology from the current drawing. `TopologyQueryElement` objects represent layers that contain topology queried from attached drawings.

`FeatureElement` objects represent layers that contain classified drawing objects.

`RasterElement` objects are not generally used. Instead, use FDO data with the Geospatial Platform API.

)ata Source Descriptors

`DisplayManagement.Element` has two properties to describe the data source for the element:

- `AcquisitionCriteriaString`
- `AcquisitionCriteria`

`AcquisitionCriteriaString` contains a string representation of the data source, as described in the following table:

Element type	AcquisitionCriteriaString value
LayerElement	AutoCAD layer name
FeatureElement	Object classification class name
TopologyElement	Topology name
AttachedDwgsQueryElement	
TopologyQueryElement	

`AcquisitionCriteria` is of type `DisplayManagement.DataSourceDescriptor`. It contains additional data about the source. Each element type has a corresponding data source descriptor type, subclassed from `DataSourceDescriptor`. For example, the `AcquisitionCriteria` property for a `LayerElement` is of type `LayerDataSourceDescriptor`.

For `LayerElement`, `FeatureElement`, and `TopologyElement`, `AcquisitionCriteria.AcquisitionStatement` is the same as `AcquisitionCriteriaString`.

`AttachedDwgsQueryDataSourceDescriptor` and `TopologyQueryDataSourceDescriptor` define additional methods and properties, as described in the following table.

Method or property	Description
GetDrawingList()	Gets the list of attached drawings used in the query
SetDrawingList()	Sets the list of attached drawings used in the query
Query	A result buffer containing the query definition, from the `QueryModel.FileOut` property
TopologyName	For `TopologyQueryDataSourceDescriptor` only, the topology name used in the query

To add a new element, create the element and its associated data source descriptor, then add it to the map. The following example adds a `LayerElement` that references an AutoCAD layer named “Layer1”. For other element types, create the appropriate data source descriptor.

```
Dim activeProject As Project.ProjectModel = _
 HostMapApplicationServices.Application.ActiveProject
Dim docs As DocumentCollection = Application.DocumentManager
Dim activeDoc As Document = docs.MdiActiveDocument
Dim trans As Transaction = Nothing

Try
  trans = activeDoc.TransactionManager.StartTransaction()

  ' Get the Object Id for the current Map
  Dim managerId As ObjectId
  managerId = _
    DisplayManager.Create(activeProject).MapManagerId( _
    activeProject, True)
  Dim manager As MapManager = trans.GetObject(managerId, _
    OpenMode.ForRead)
  Dim currentMapId = manager.CurrentMapId
  Dim currentMap As Map = trans.GetObject(currentMapId, _
    OpenMode.ForWrite)

  ' Create the Layer element and set its
```

```
name
  Dim element As LayerElement = LayerElement.Create()
  element.Name = "NewLayer"

  ' Create the Layer Descriptor
  Dim descriptor As LayerDataSourceDescriptor = Nothing
  descriptor = LayerDataSourceDescriptor.Create()
  descriptor.AcquisitionStatement = "Layer1"

  ' Now Add the new element to the current
Map
  Dim iterator As IEnumerator = _
    currentMap.NewIterator(True, True)
  Dim elementId As ObjectId = _
    currentMap.AddItem(element, iterator)
  trans.AddNewlyCreatedDBObject(element, True)

  element = trans.GetObject(elementId, OpenMode.ForWrite)
  element.AcquisitionCriteria = descriptor

  trans.Commit()
  trans = Nothing
Catch e As Autodesk.AutoCAD.Runtime.Exception
  ' Handle exception
Finally
  If Not trans Is Nothing Then
    trans.Abort()
    trans = Nothing
  End If
End Try
```

tyle

Display Manager elements can have style associated with them.

`DisplayManagement.Style` is the base class for all the style classes. It is based on `AutoCAD.DatabaseServices.DBObject`, so it must be managed using AutoCAD transactions. The available style classes are:

- `DefaultStyle`
- `EntityStyle`
- `StylizationEntityAnnotationStyle`
- `StylizationEntityHatchStyle`
- `StylizationEntityTextStyle`
- `RasterStyle`
- `ThematicStyle`

`Element.AddStyle()` creates a reference from the element to the style object in the database. Multiple elements can refer to the same style object.

To create a new style, call its `Create()` method. Set the appropriate properties for the style type. Save the style in the database and add it to an element. For example, the following creates a new entity style and assigns it to a layer element:

```
Try
  trans = activeDoc.TransactionManager.StartTransaction()

  ' Open the element for write, so the style can be added
  Dim layer As Element = trans.GetObject(layerId, _
    OpenMode.ForWrite)

  ' Pass 0.0 for the current scale
  Dim styleRefIterator As StyleReferenceIterator = _
```

```
    layer.GetStyleReferenceIterator(0.0, True, True)

  ' Add the style
  Dim styleEntity As EntityStyle = EntityStyle.Create()

  ' Set style properties
  Dim color As Autodesk.AutoCAD.Colors.Color = _
  color.FromColorIndex( _
    Autodesk.AutoCAD.Colors.ColorMethod.None, 5)
  styleEntity.Color = color
  styleEntity.Name = styleName

  Dim id As ObjectId
  id = layer.AddStyle(styleEntity, styleRefIterator)
  trans.AddNewlyCreatedDBObject(styleEntity, True)

  trans.Commit()
  trans = Nothing
Catch e As System.Exception
  ' Process exception
Finally
  If Not trans Is Nothing Then
    trans.Abort()
    trans = Nothing
  End If
End Try
```

MapBook

Topics in this section

Overview

A Map Book is a way to divide a large map into smaller tiles, by creating a separate layout for each tile.

Each project has its own MapBook manager. Call `MapApplication.GetBookManager()` to get the book manager for a database.

A Map Book requires a template file that defines the layout for each sheet in the book.

MapBook Templates

A Map Book template is an AutoCAD template (.dwt) file with some special characteristics.

The template file must define at least one layout. The layout can contain viewports for the following purposes:

- Map view, which displays the map for the area covered by a particular sheet
- Key view, which displays a small image of the entire map, outlining the area covered by the map sheet
- Map legend, which displays the map legend

The layout can also contain special blocks that show links to adjacent map sheets. Within AutoCAD Map 3D or a DWF file, the links can be used to jump directly to an adjacent sheet.

The layout can also contain a title block.

To identify objects in the template as any of the special views or blocks, call the static function `MapBook.SheetTemplate.MarkElement()` with the object id and element type. For example, to mark an object as the map view, call

```
MapBook.SheetTemplate.MarkElement(objId, _
   MapBook.TemplateElementType.MapView)
```

Adjacent Map Sheets

The arrows pointing to adjacent map sheets are AutoCAD block references. Each block can have an attribute that defines the direction of the arrow and creates a link to the adjacent sheet.

To create an arrow to an adjacent map sheet, place a block reference of the

desired shape and orientation into the map. The block reference should have an attribute named TAG. Set the text string of the attribute to an expression that defines the adjacent sheet. This expression is of the form:

```
%<\AcSm Sheet.direction >%
```

where *direction* is one of the following:

- Top
- Bottom
- Right
- Left
- TopRight
- TopLeft
- BottomRight
- BottomLeft

Creating a Map Book

To create a Map Book, define the book settings using `TileGenerator` and `TileNameGenerator` objects. Call `BookManager.GenerateMapBook()` to create the new book.