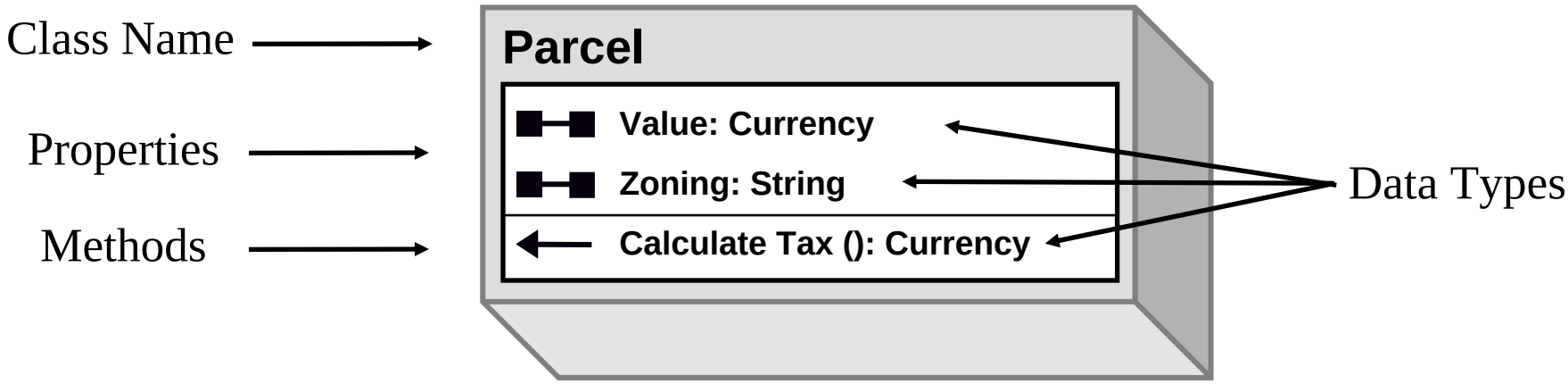
Chapter 11 – Navigating object model diagrams

- Getting layers
- Creating and assigning colors

Chapter 11 – Navigating object model diagrams

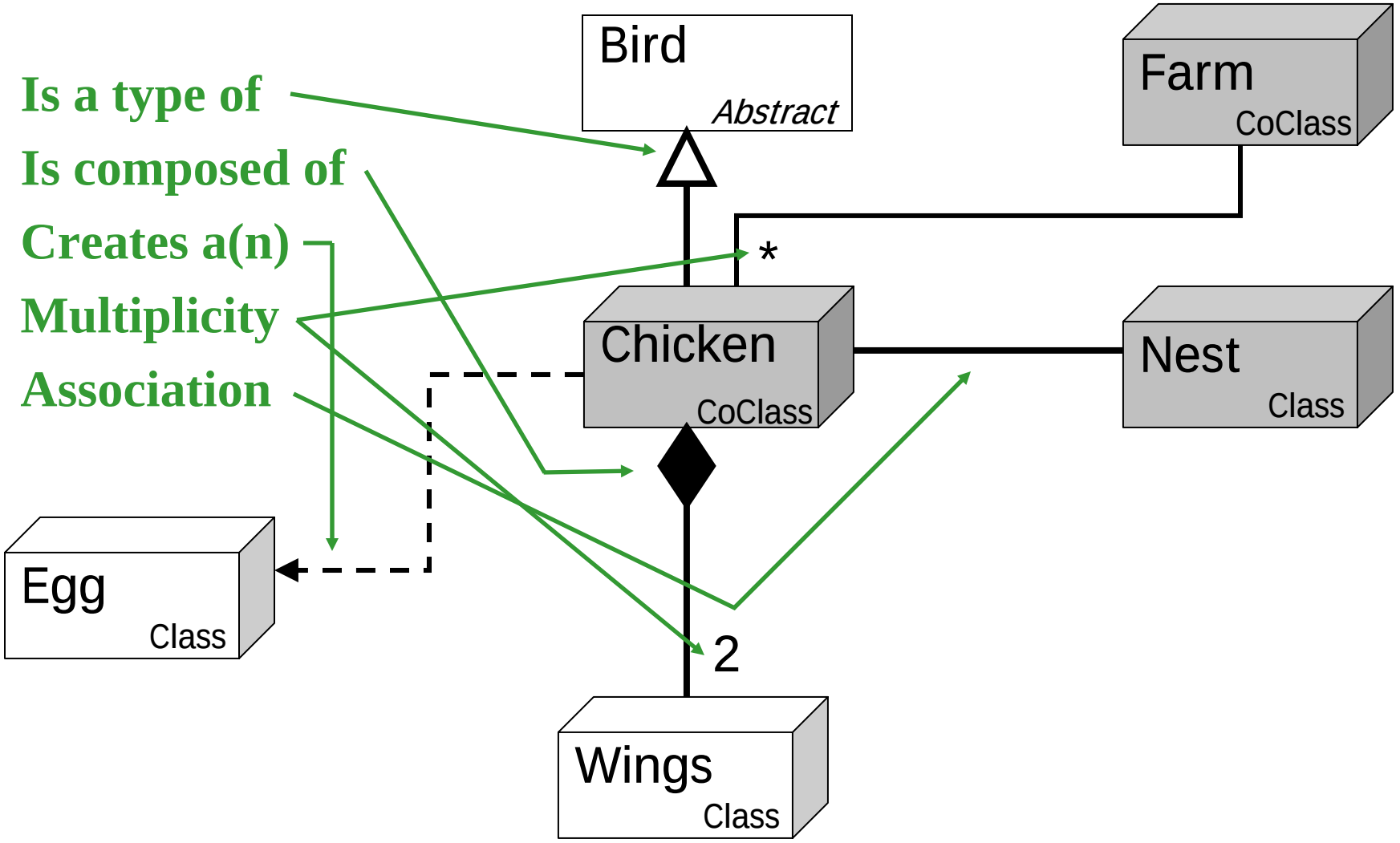
- In our previous lecture, we introduced **Unified Modeling Language (UML) diagrams** for classes:



- While these diagrams are useful to us just to see the **characteristics of a class**, their **real power** comes in showing us the **relationships between classes** in the **ArcGIS object model**

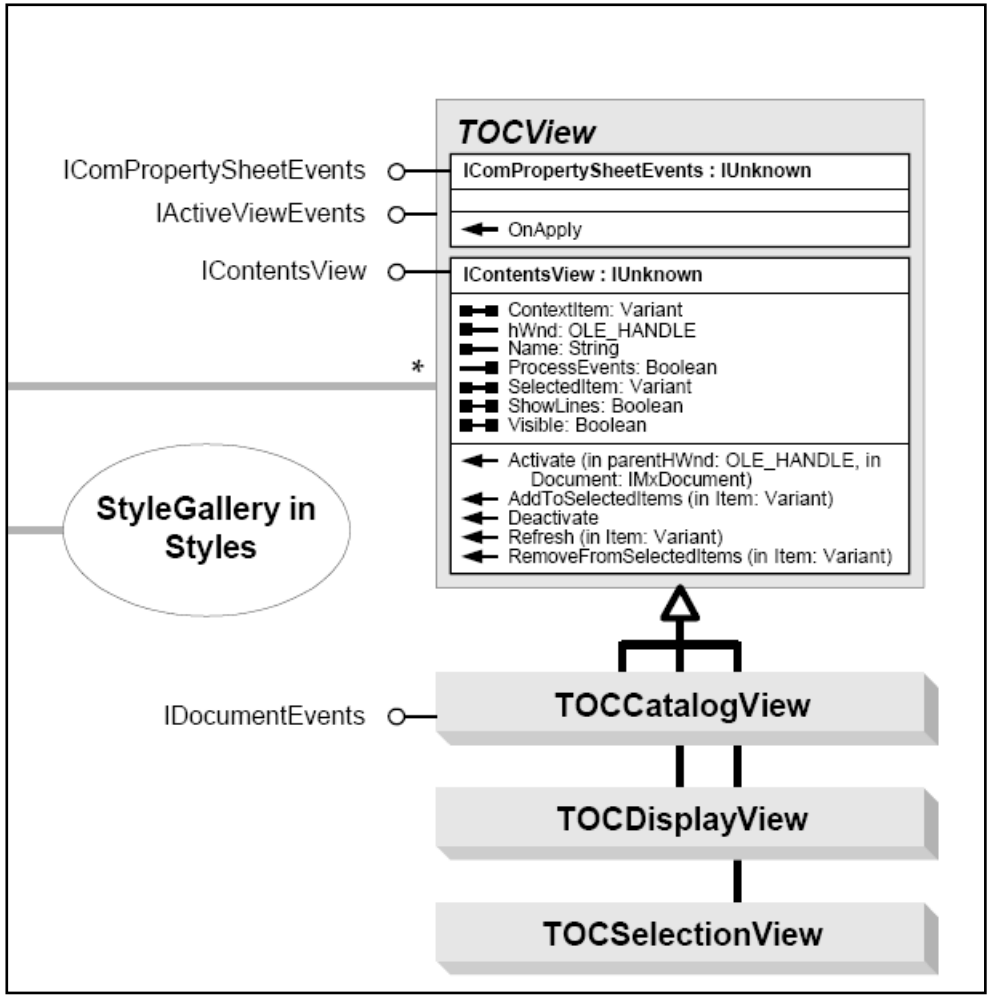
Chapter 11 – Navigating object model diagrams: Relationships between classes

- Is a type of
- Is composed of
- Creates a(n)
- Multiplicity
- Association



Chapter 11 – Navigating object model diagrams: Abstract classes

- **Abstract classes** are symbolized by a **2-D gray box**
- They are **neither instantiable** (using the New keyword) **nor** are they **creatable** (by using requests to other classes)
- They define **general interfaces** for subclasses



Chapter 11 – Navigating object model diagrams: CoClasses

- **CoClasses** are symbolized by a **3-D gray box**
- They are **instantiable**, using the New keyword, e.g.:

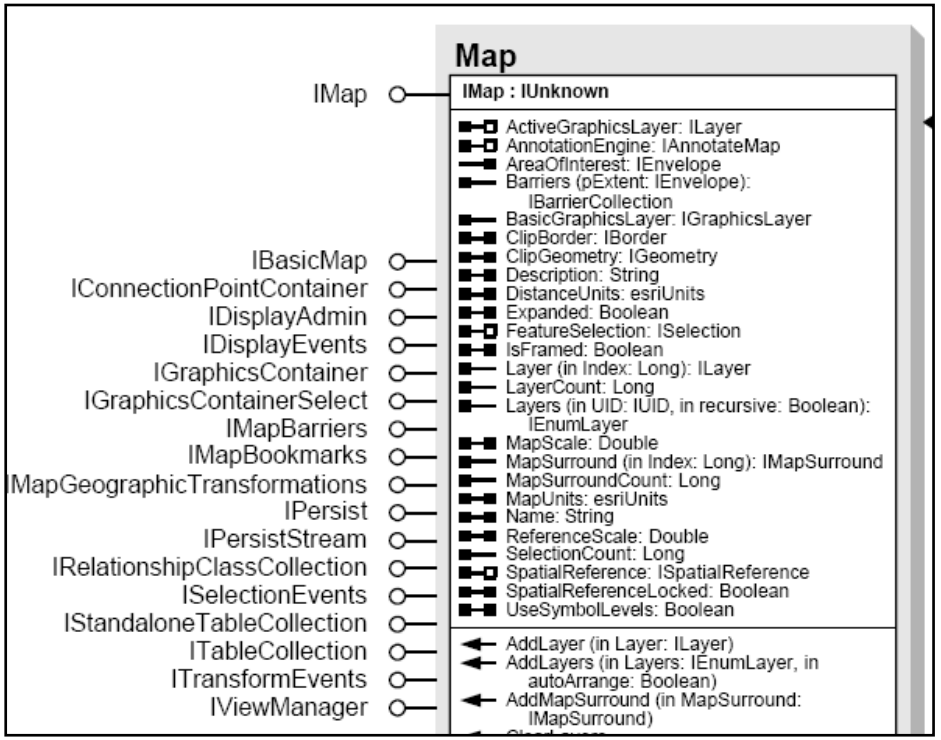
  Dim pMap as IMap

  Set pMap = New Map

- They are **creatable**, e.g.:

  Dim pMap as IMap

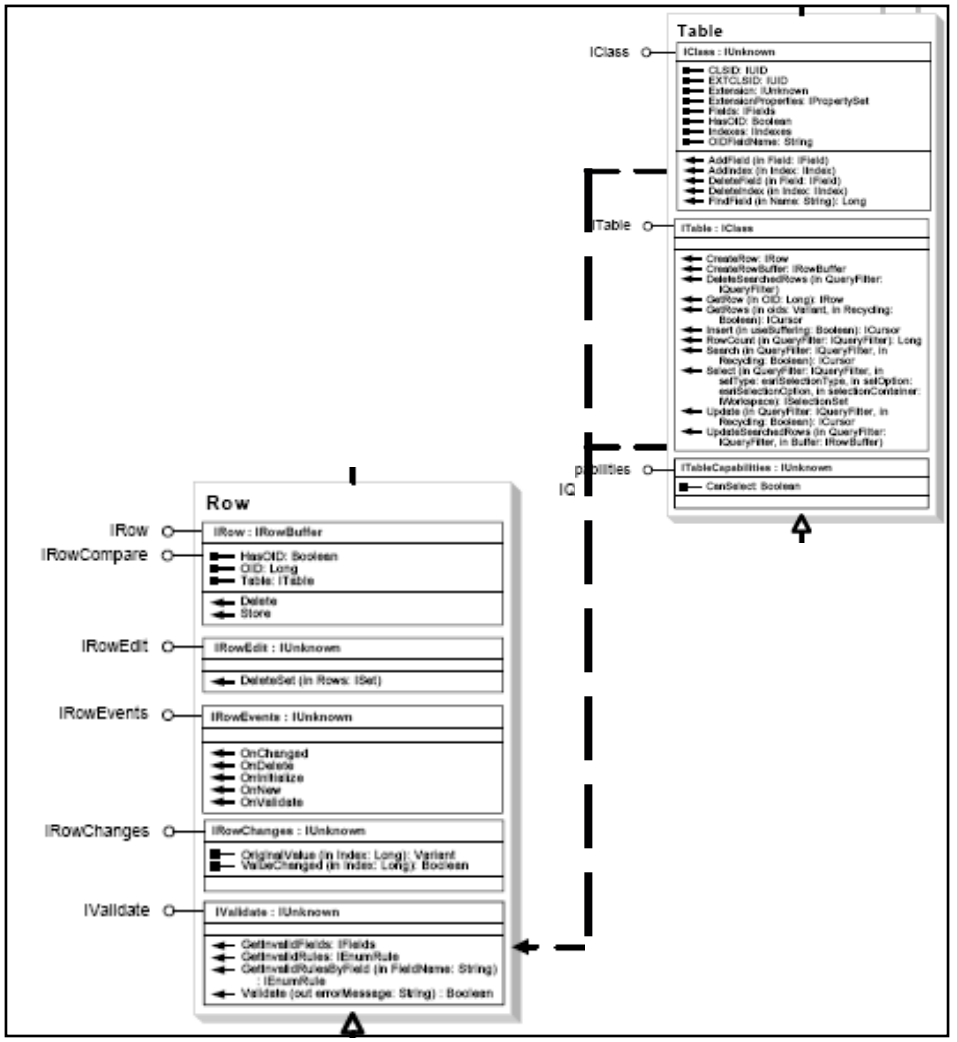
  Set pMap = pMxDocument.FocusMap



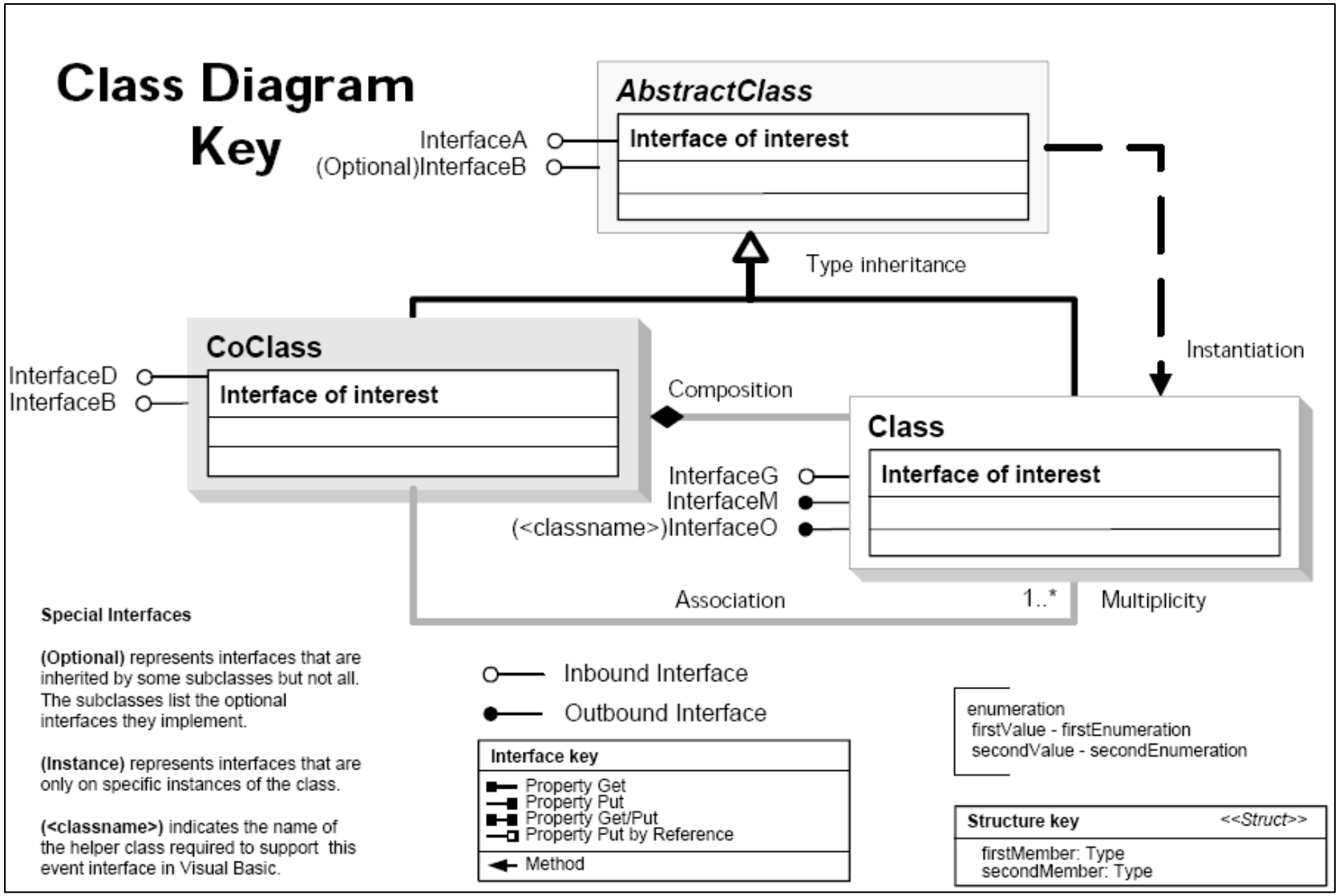
Chapter 11 – Navigating object model diagrams: Classes

- **Classes** are symbolized by a **3-D white box**
- They are **not instantiable** (you cannot make one using the New keyword)
- They are **creatable**, you must **obtain instances from other objects**, e.g.:

```
Dim pNewRow as IRow
Set pNewRow =
    pTable.CreateRow
```



Chapter 11 – Navigating object model diagrams: Reading object model diagrams



Chapter 12 – Making tools

- Reporting coordinates
- Drawing graphics
- Using TypeOf statments

Chapter 12 – Making tools

- On multiple occasions earlier in the course, it has been mentioned that **tools** in ArcGIS **are different from buttons**
- This is obvious even from the **user's point of view**: **Clicking on a button** causes ArcGIS to **do something immediately**, whereas **clicking on a tool changes the appearance of the cursor** … and then the user then use the mouse to control the cursor to use the tool to do something
- As a budding ArcGIS programmer, you probably can guess that **developing the code for a tool** is going to be **more complicated** than it is for a button

Chapter 12 – Making tools

- The **key procedure** for a **button** is the code associated with its **click event**
- But for a **tool**, which the user can interact with in a number of ways, there are **many more events to code**
- And beyond the number of events, developing a tool requires you to have a **broader understanding of a variety of objects** (maps, layers, geometry like points that specify the position of the cursor)
  - Hopefully you are **becoming familiar with these many objects** and even if you are not …
  - Hopefully you now **know how to use the UML object model diagrams** to find out the things you need to know

Reporting coordinates

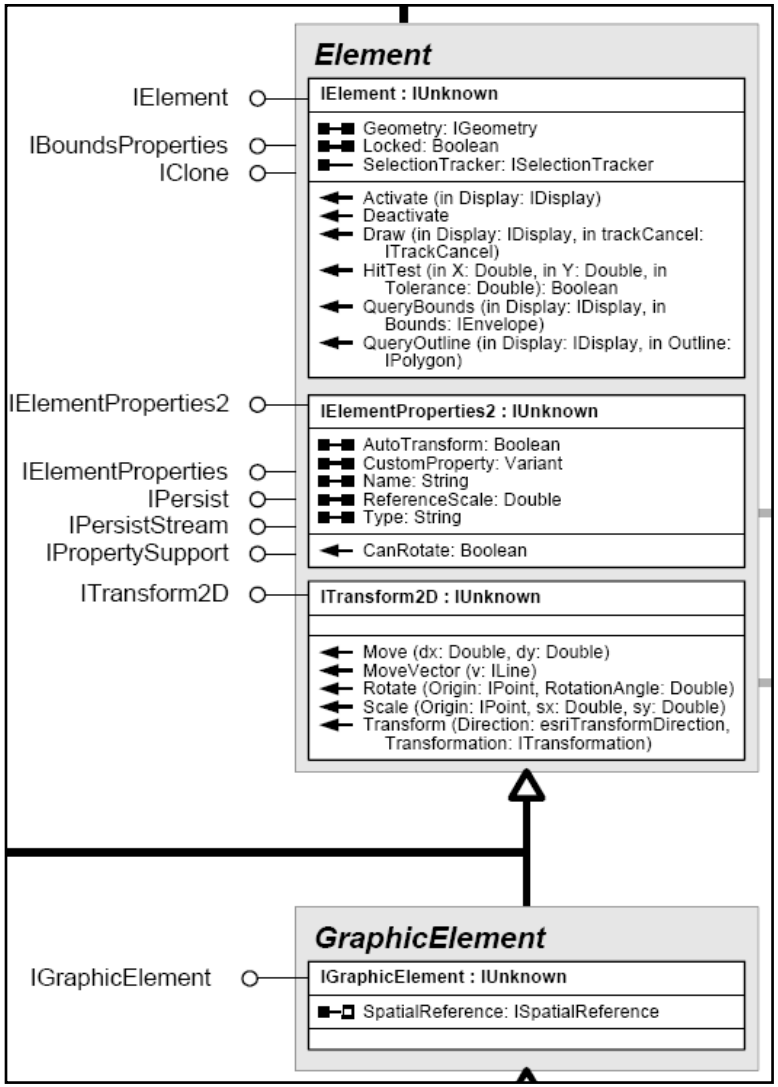
- Tools have a **MouseMove event procedure** that runs whenever the user moves the cursor with the tool selected
- This procedure takes **four arguments**, that the user specifies by using the mouse:
  - button As Long
  - shift as Long
  - x As Long
  - y As Long
- Each of these integers is a **value that represents some part of the mouse state**: Button and shift reflect whether the button or shift key is depressed, x and y report the position in pixels of the mouse pointer

Reporting coordinates

- With the button and shift variables, **If Then statements** can be used to **create appropriate code** for the various permutations
- Further events like **MouseDown and MouseUp** respond to pressing or releasing the mouse button
- Note that the **x, y** reported here are **pixel positions** in the map display which (of course) are **not in geographic coordinates** … but fortunately, we can **navigate through the object model** to find the appropriate objects, interfaces and properties to get the position of the map pointer in geographic coordinates

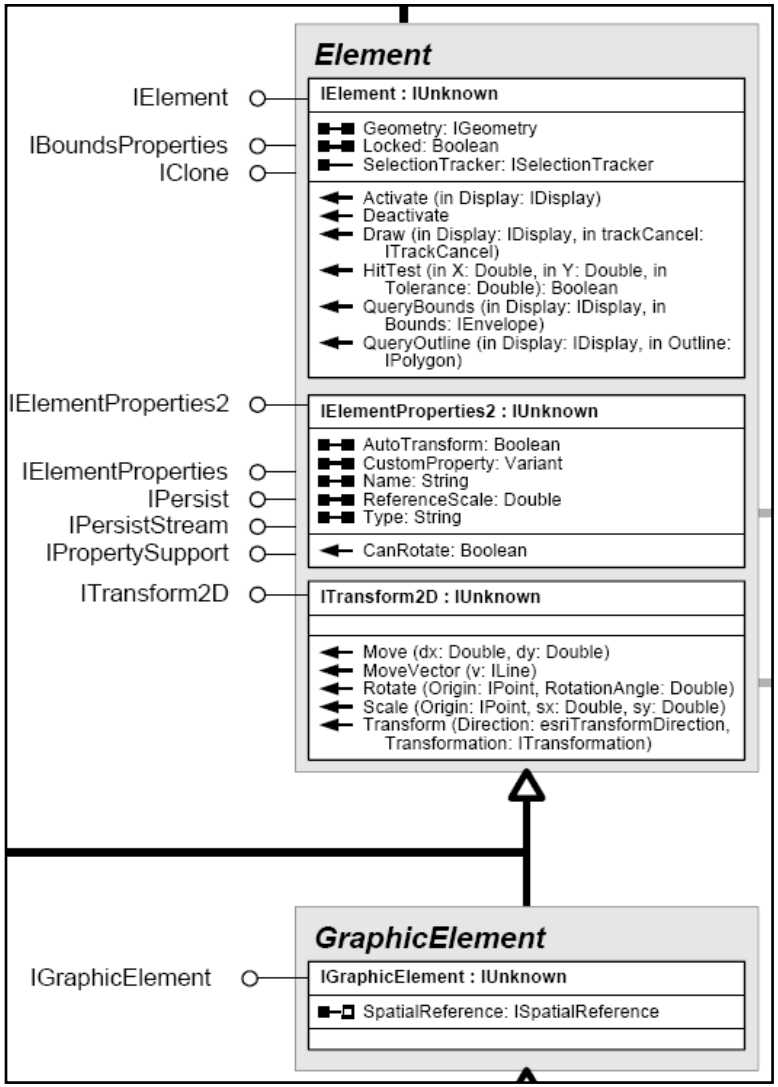
Drawing graphics

- Graphics belong to an abstract class called **Element**
- Element, has **two abstract subclasses** (FrameElement & GraphicElement)
- We are interested in **GraphicElement**, which in turn has coclasses under it named **MarkerElement, LineElement, PolygonElement, and TextElement**



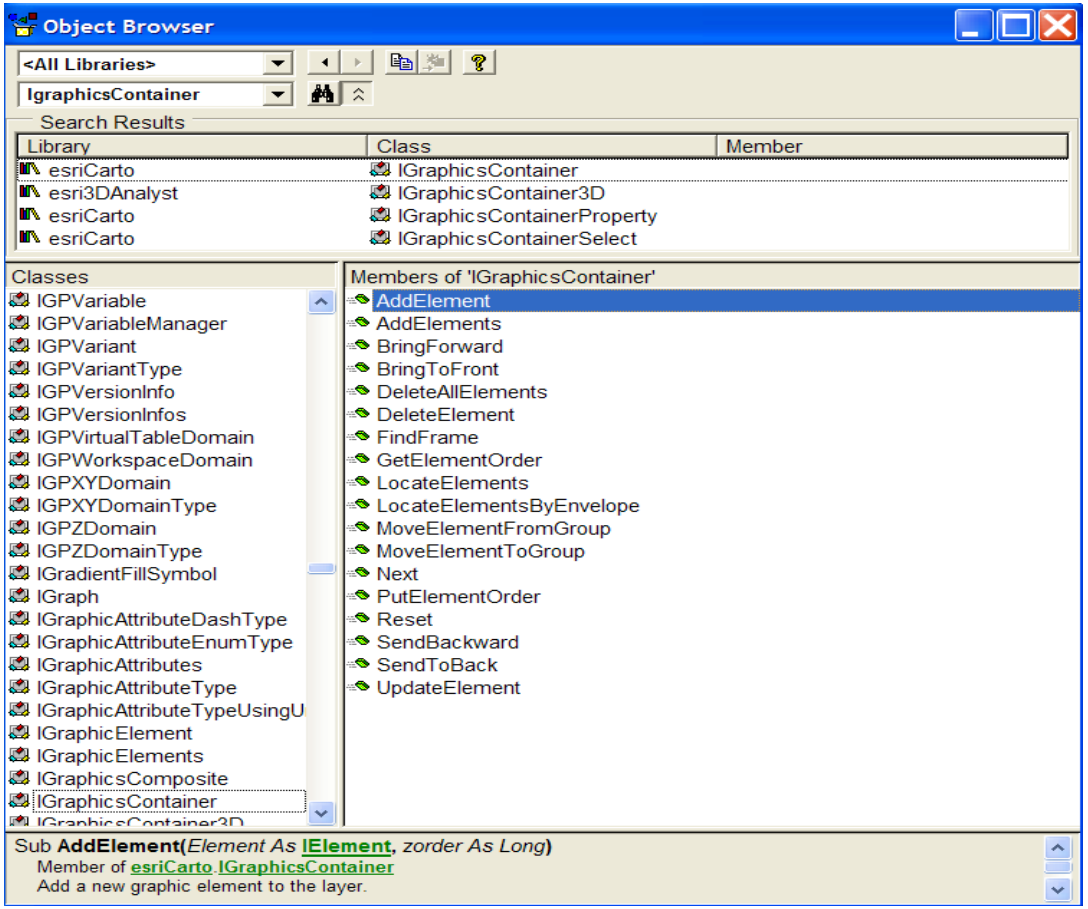
Drawing graphics

- Note that the abstract element class has a **Geometry** property
- This means that **geometry objects** (like points, lines, and polygons) are **associated with elements** (MarkerElements, LineElements, and PolygonElements respectively)
- We thus can create appropriate **geometry objects** (like points) and use them to **position marker elements** on a map



Drawing graphics

- Adding a graphic to a Map is done through the Map's **IGraphicsContainer** interface
- Once added, **refreshing the Map** causes it to redraw itself, including the graphics associated with it (see the text for details)



Using TypeOf statements

- Once you have developed your tool for drawing graphics at rescue sites in the Map View in Exercise 12B, we have **a problem**:
  - This tool **would not work properly in the Layout View**, which does not operate in geographic units
- We need a way to **distinguish between Map and Layout Views** to turn the tool on and off appropriately, and we explore this in Exercise 12C, **using TypeOf statements**
- In this example, and in a diverse set of other situations where having an **object of the wrong type** would **break our code** (and return a type mismatch error), we can use **TypeOf** (which returns TRUE or FALSE) to check if an object is the required type

Chapter 13 – Executing Commands

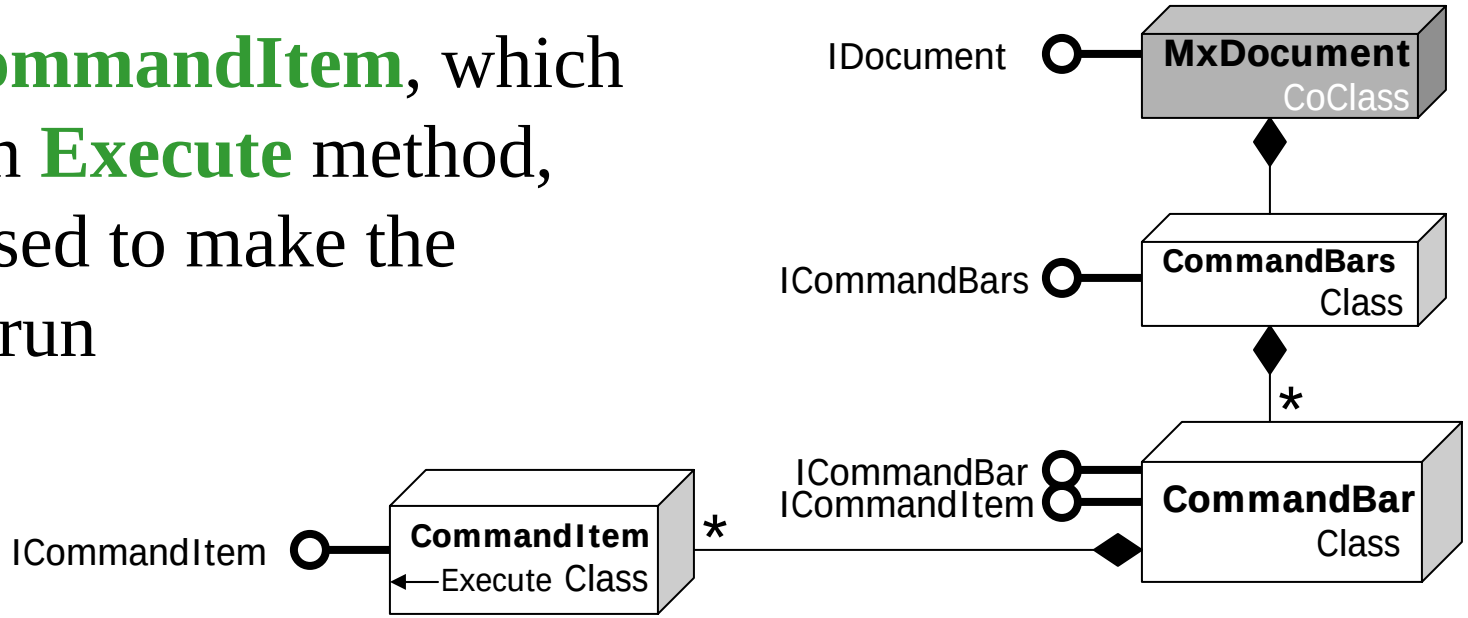
- Using CommandItems and CommandBars

Chapter 13 – Executing Commands

- As you have seen throughout in our exploration of ArcGIS VBA, **modularity and the reusability** of **functionality and code** is a **key concern**
- If at all possible, we want to **avoid** reinventing the wheel:
  - If someone has already **developed the capability to perform a particular function**, the last thing we want to do is replicate their work; **we want to be able to make use of it**
- This is **equally true of ArcGIS' existing commands** and the functions they perform
  - We **do not get to see the code** that runs behind them (they are not written in VBA; using COM they were developed in C++)
  - **We can still call them**, so we can include them in our code

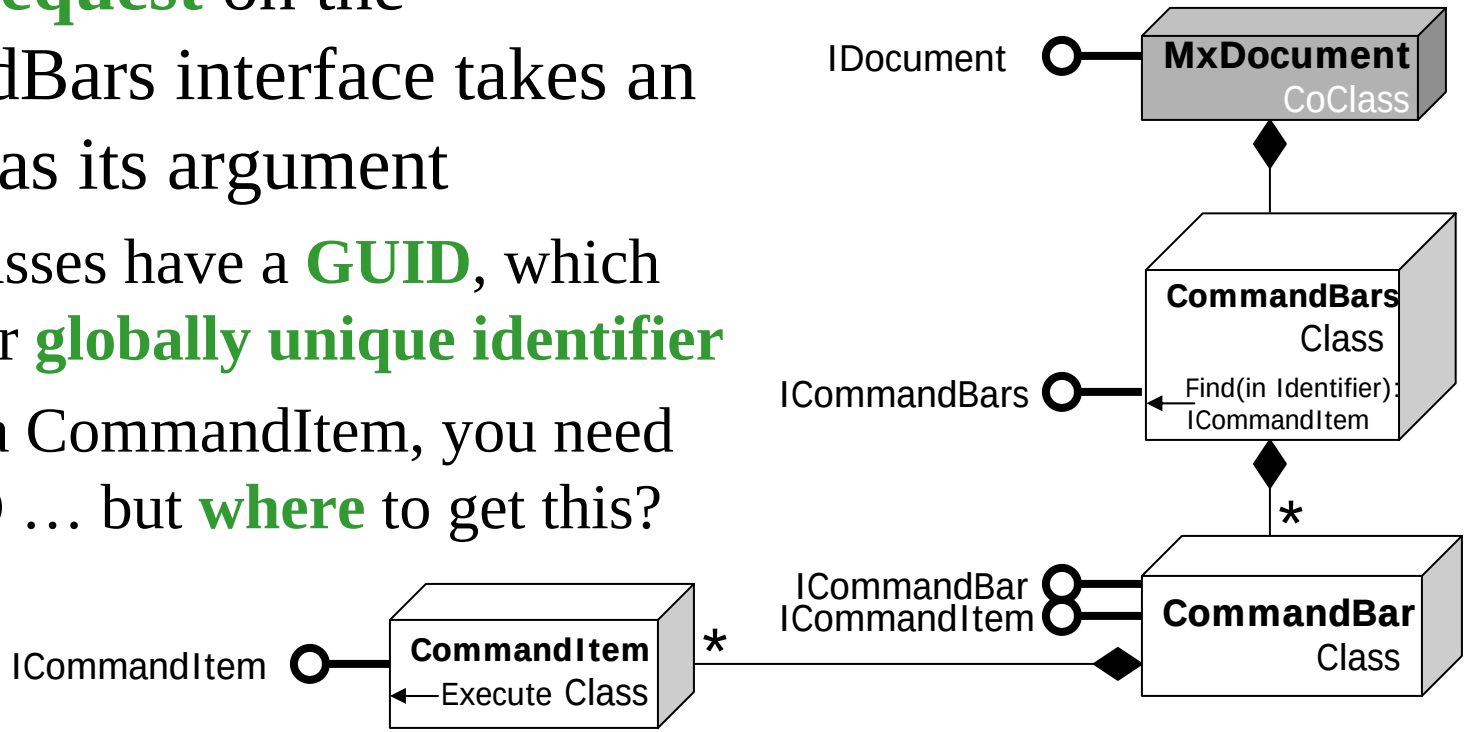
Using CommandItems and CommandBars

- Toolbars are **composed** of commands, whether they contain tools, buttons or menu choices
  - They belong to the **CommandBar class**
  - From the notation below, you can see a **CommandBar is made up of multiple CommandItems** (commands)
- Commands have an **interface called ICommandItem**, which includes an **Execute** method, which is used to make the command run



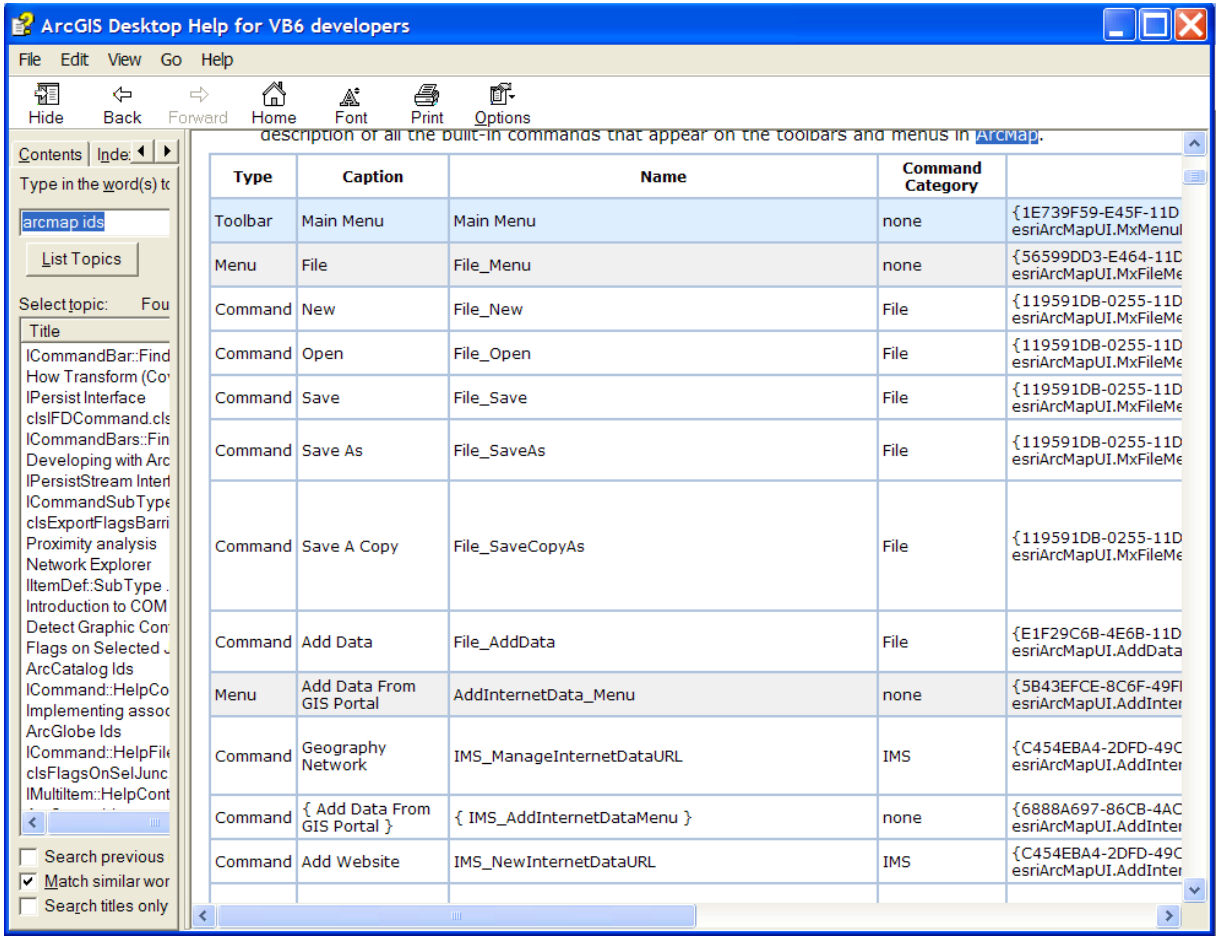
Using CommandItems and CommandBars

- The **CommandBars class** (note the ‘s’ at the end) is a **collection** of all the CommandBar objects available
  - Note the **same symbology here**, showing the ‘composed of multiple objects relationship’
- The **find request** on the ICommandBars interface takes an **identifier** as its argument
  - COM classes have a **GUID**, which stands for **globally unique identifier**
  - To **find** a CommandItem, you need its GUID … but **where** to get this?



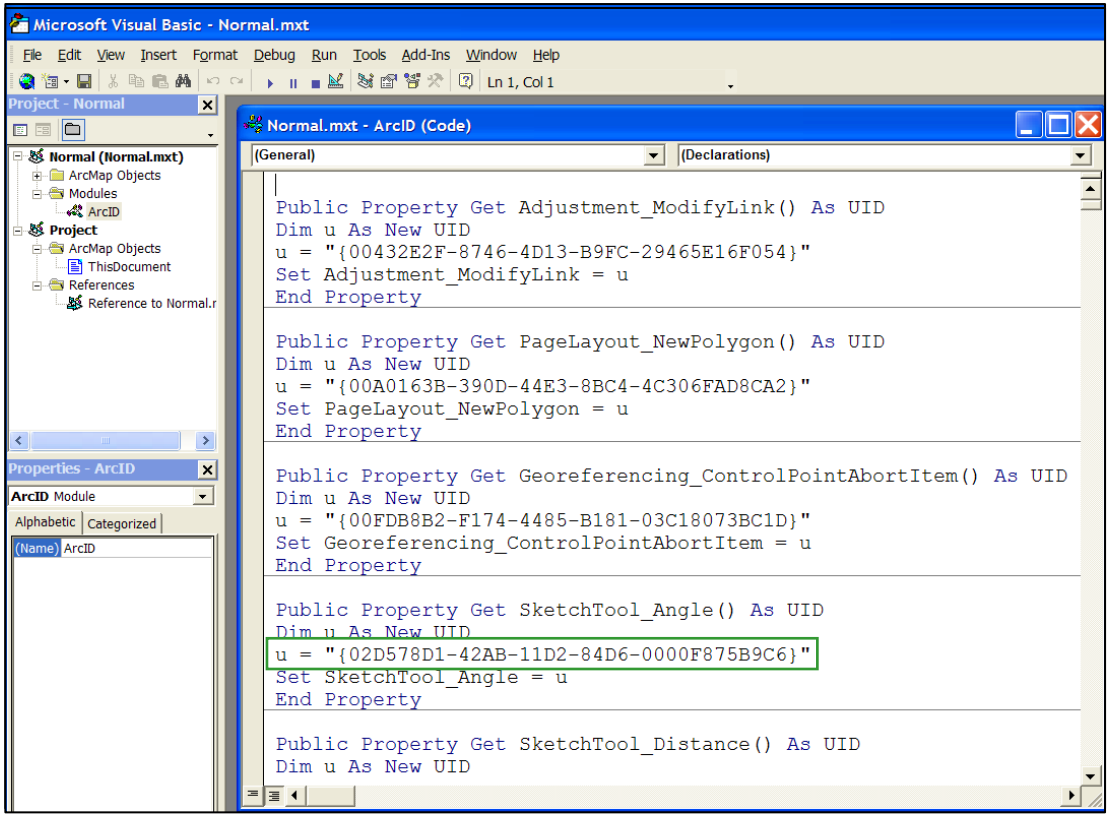
Using CommandItems and CommandBars

- You can **look GUIDs up in the Developer Help** in the topic *ArcMap: Names and IDs of commands and commandbars:*



Using CommandItems and CommandBars

- GUIDs are **32-character hexidecimal strings**, and as such are **inconvenient to copy and paste** into code
- Instead, we can use **procedures built into the ArcID code module** of the normal.mxt project to **fetch** them
- These make it **easy to get a GUID** by getting the **appropriately named property** of ArcID



`ArcID.SketchTool_Angle`

Using CommandItems and CommandBars

- Putting this **all together**:

```
Dim pCommandItem As ICommandItem
Set pCommandItem = CommandBars.Find(ArcID.SketchTool_Angle)
pCommandItem.Execute
```

- Getting a **toolbar** works in a **similar fashion**
  - **Toolbars have GUIDs** too, and can be found in the same way

```
Dim pCommandItem As ICommandItem
Set pCommandItem = CommandBars.Find(ArcID.Editor_EditorToolbar)
```

- However, toolbar properties and methods are on the **ICommandBar interface** (not ICommandItem), so we QueryInterface to get the right interface:

```
Dim pCommandBar As ICommandBar
Set pCommandBar = pCommandItem
```

Chapter 14 – Adding layers to a map

- Adding a geodatabase feature class
- Adding a raster data set

Chapter 14 – Adding layers to a map

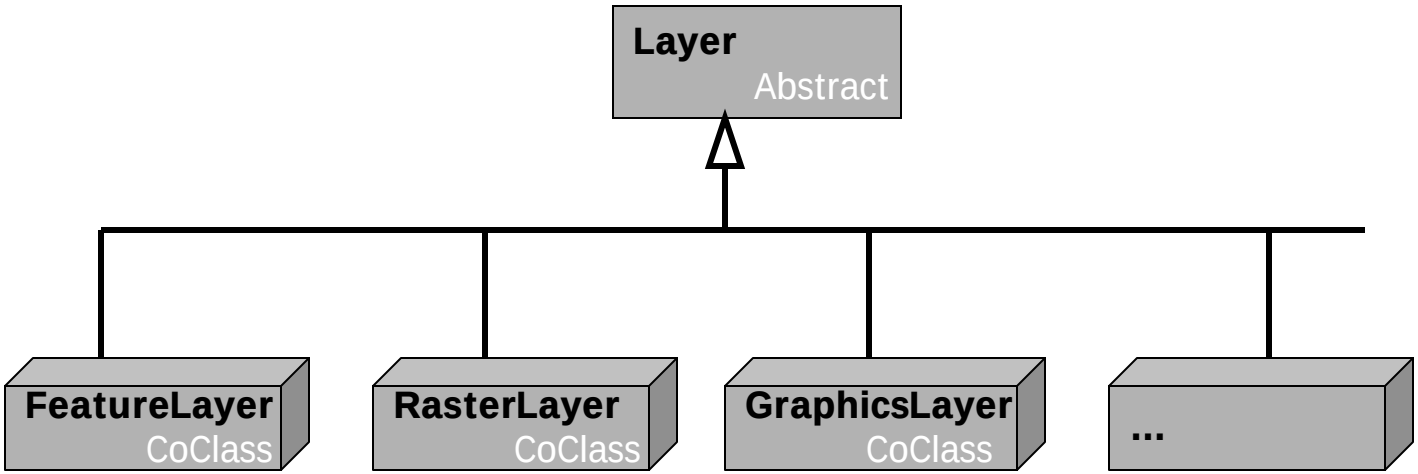
- **Adding layers to maps through the GUI** is something every user does when they use ArcMap
- Equally important to the developer is **to be able to add layers using code**, as this is a necessary precondition to doing something to the layers with the code
- This is really a **four step** process:
  1. **Create the layer** from one of the layer coclasses
  2. **Get the data set from a storage location** that the computer can access (either locally or somewhere networked)
  3. **Associate** the data set with the layer
  4. **Add the layer** to the map

Chapter 14 – Adding layers to a map

- The first step, creating the layer from one of the layer coclasses, uses **straightforward VBA code**:

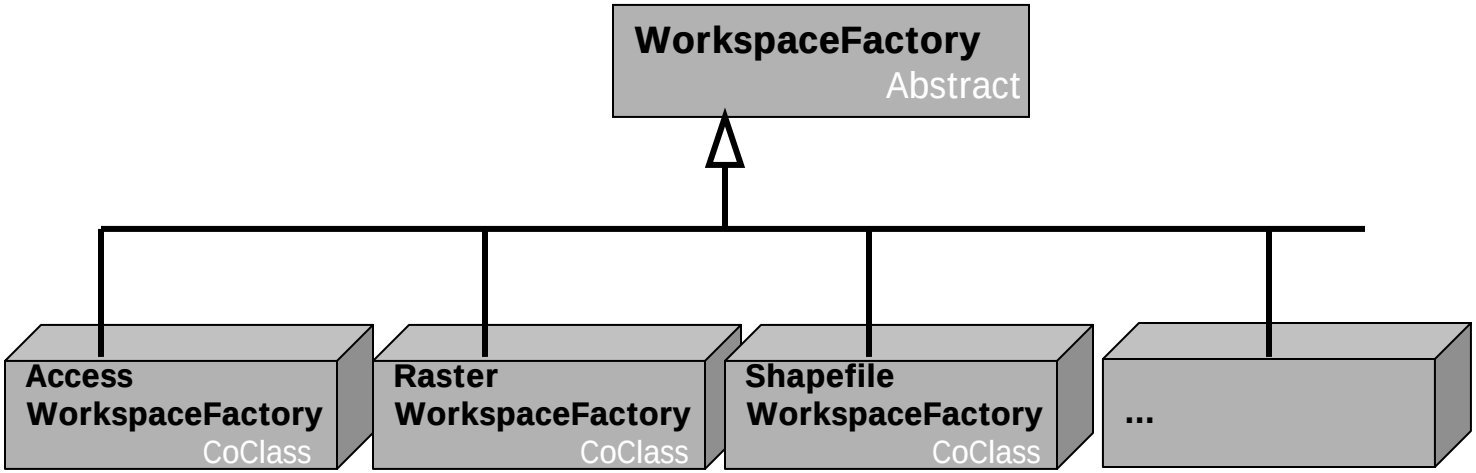
```
Dim pRLayer as IRasterLayer
Set pRLayer = New RasterLayer
```

- The key is to **identify the appropriate type** of layer:



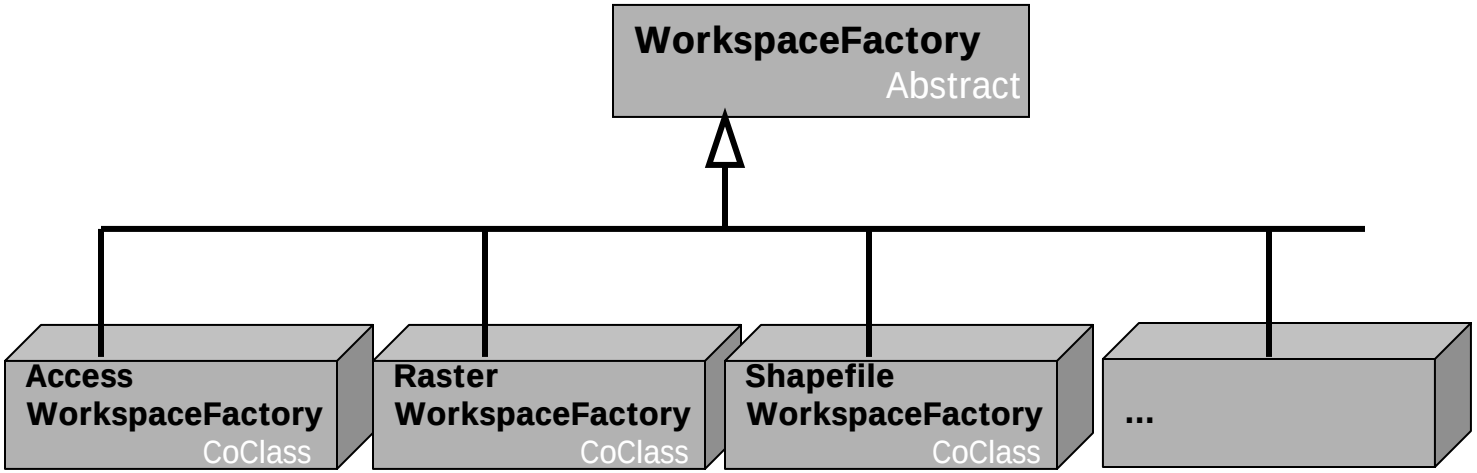
Chapter 14 – Adding layers to a map

- The second step, **getting the data set**, is a little **more tricky** … partly because ArcGIS is **so flexible** with data
  - Because ArcGIS can work with **so many different kinds of data files**, there are **lots of variations** on this
- To simplify the process, in all cases to get a data set, one must first **get its workspace**, which one **creates using a workspace factory**:

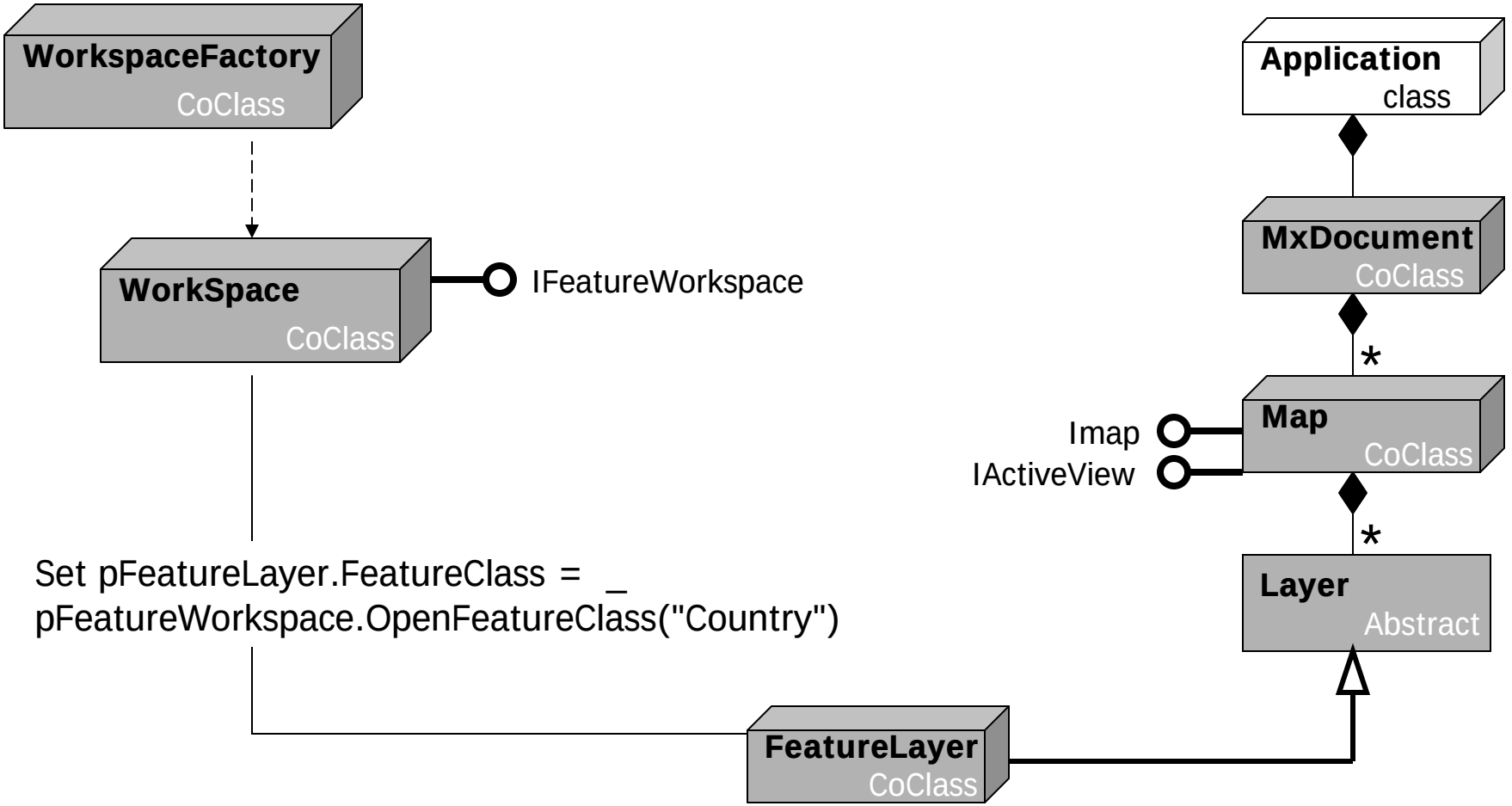


Chapter 14 – Adding layers to a map

- You **select** the right WorkspaceFactory from the many coclasses, and use it to **create the required workspace**
- **Workspaces are composed of data sets** (which is what we are really after)
- There are WorkspaceFactories **specific to each type of data set files** we might want to add to our map:



Chapter 14 – Adding layers to a map
ShapeFile Example



Adding a geodatabase feature class

- Your first exercise will take you through the **four step process** using a **geodatabase feature class**
- The first key thing that **you need to know**, both here and in all cases really, is the **kind of data file** in question → this **determines the right kind of WorkspaceFactory**
- Here we are working with an **MS Access database**, so we need an **AccessWorkspaceFactory**:

```
Dim pAWFactory As IWorkspaceFactory
Set pAWFactory = New AccessWorkspaceFactory
```

- The IWorkspaceFactory interface has an **OpenFromFile method** that is used to open the file:

```
Dim pFWorkspace As IFeatureWorkspace
Set pFWorkspace = pAWFactory.OpenFromFile("thefile.mdb",0)
```

Adding a geodatabase feature class

- We now have the Workspace required and we can now **get the feature class** with the **OpenFeatureClass** method on the IFeatureWorkspace interface of our Workspace:

```
Dim pFClass As IFeatureClass
Set pFClass = pFWorkspace.OpenFeatureClass("Roads")
```

- Setting up a feature layer and associating it with the class is relatively **straightforward**:

```
Dim pFLayer As IFeatureLayer
Set pFLayer = New FeatureLayer
Set pFLayer.FeatureClass = pFClass
```

- Finally, adding it to the Map document is equally **straightforward** (see the text for the five lines of code required)

Adding a raster data set

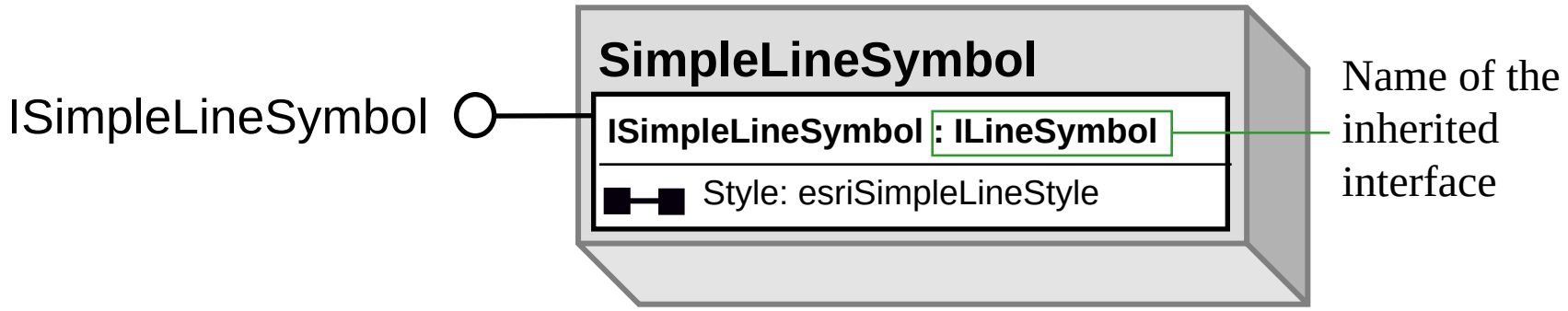
- Your second exercise involves a **similar procedure**, only this time the data set is **raster data rather than features** from within a geodatabase
- The only real wrinkle is switching to **use the right WorkspaceFactory for the particular kind of data** … but the hope is that once you have done this for two different sorts of data, you will be **comfortable** with doing it for **any sort of data set**
- This way, you will have worked with **data sets from both the vector and raster spatial data models**, which covers most of what you are likely to work with in real applications

Chapter 15 – Setting layer symbology

- Setting layer color
- Setting layer symbols
- Creating a class breaks renderer

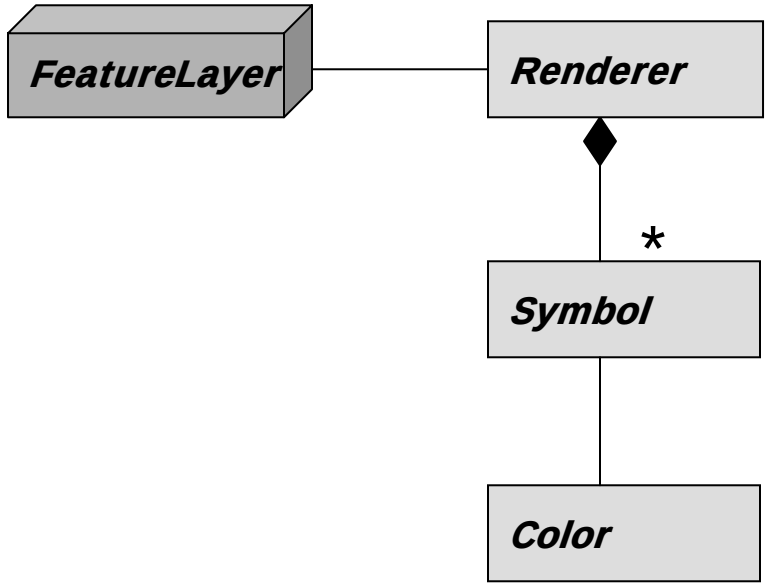
Chapter 15 – Setting layer symbology
Interface inheritance

- Recall back in Chapter 10 when we learned of **class inheritance**: Derived classes can take over (or **inherit**) **properties, methods, and interfaces** of the pre-existing classes, which are referred to as base classes
- In this chapter, we look at a **form of inheritance** that is a **subset** of the above, called **interface inheritance**:
  - The properties and methods **associated with a particular interface** are inherited, but properties and methods from **other interfaces on the same class** **ARE NOT** inherited here



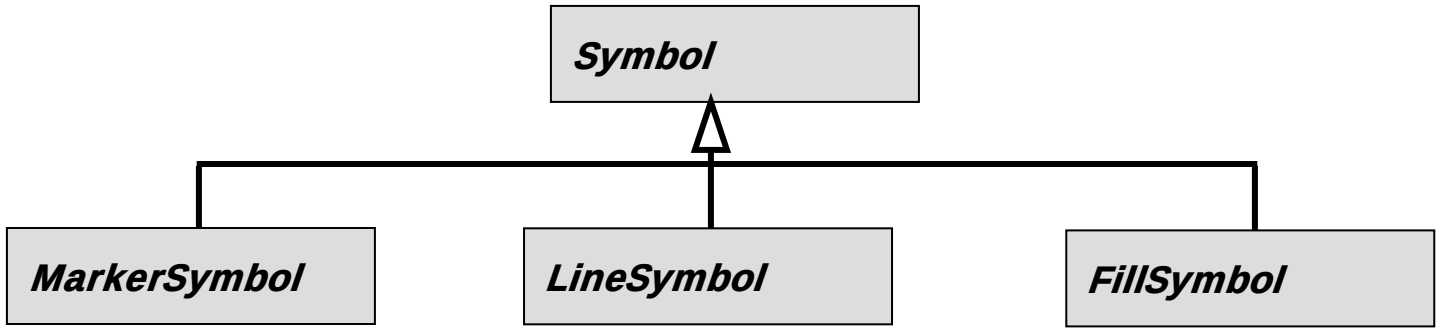
Setting layer color

- By **default**, when a layer is added to a map using the GUI, it is a symbolized with a **single random color**
- This is the **default renderer** assigned to the layer
- As an alternative, we can **write code** to make use of **another renderer**
  - Every **feature layer** has a renderer
  - Renderers are **composed of symbols**
  - Every symbol has a **color** (different kinds of symbols will have other sorts of characteristics as well)



Setting layer color

- The **Symbol abstract class** has many **subclasses**; the basic ones are:
  - The **MarkerSymbol** class for **points**
  - The **LineSymbol** class for **lines**
  - The **FillSymbol** class for **polygons**
- These, in turn, are abstract classes that each have their own **subclasses** (see page 266 of the text)



Setting layer color

- The **usual approach** applies here: **Symbols** and their **Colors** are **declared** with the `Dim` keyword, **created** with the `New` keyword, and **properties** are set with the **object.property** syntax
- Every **FeatureLayer** has one **FeatureRenderer**; FeatureRenderer is an abstract class with **eight subclasses** for the **various legend types**:
  - UniqueValueRenderer
  - DotDensityRenderer
  - SimpleRenderer
  - ClassBreaksRenderer
  - ScaleDependentRenderer
  - ChartRenderer
  - BitUniqueValueRenderer
  - ProportionalSymbolRenderer

Setting layer symbols

- In addition to **specifying the characteristics** of symbols yourself, you can also **draw upon pre-existing sets of symbols**
- ArcGIS symbols are stored in the **Style Manager**, grouped by style gallery classes that contain individual style gallery items
- These are **designed to be used for common thematic maps** of various types
- This is as simple as **finding the styles** you wish to use **in the Manager**, and then **navigating** the associated objects and classes (known as **Enums**, from enumerations) to **obtain those symbols for your use**

Creating a class breaks renderer

- A particularly **useful application** of manipulating legends / renderers by code is to **create them with particular ranges of associated attribute values**
- This kind of renderer is a **ClassBreaksRenderer**, and by working with these through VBA, you can **specify the exact ranges of attribute values** associated with particular symbols
- You might use this approach if you are **making many similar maps, and want to ensure they all have precisely the same legend** (and ranges of values associated with particular symbols)

Chapter 16 – Using ArcCatalog objects in ArcMap

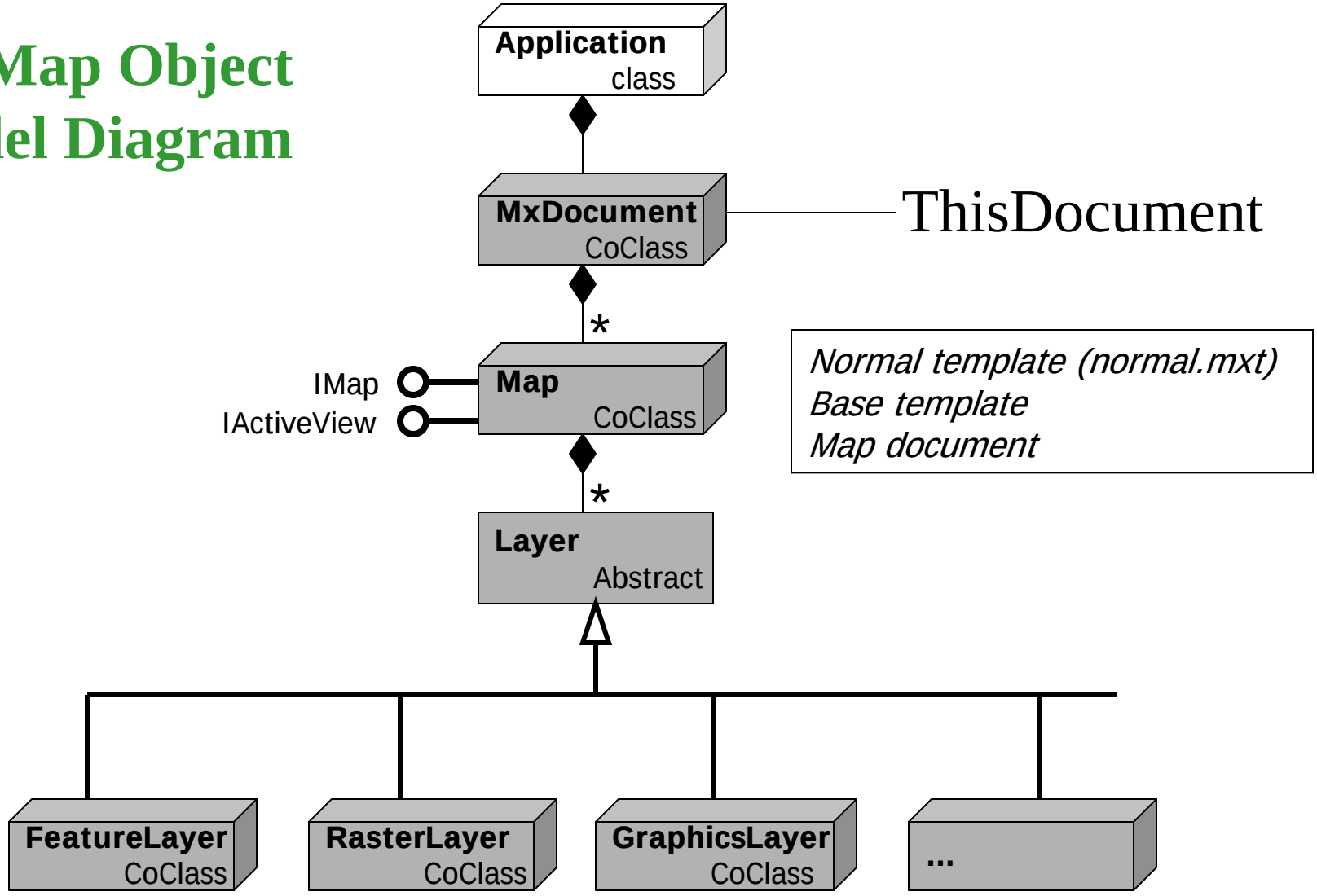
- Adding layer files to ArcMap
- Making your own Add Data dialog box

Chapter 16 – Using ArcCatalog objects in ArcMap

- The ArcCatalog object model has **similar starting points** to that of ArcMap
  - There is an ArcCatalog **Application object** named **Application**
  - There is a **GxDocument object** named **ThisDocument**
- One **key difference** is the location **where customizations can be stored**
  - **Unlike ArcMap** with its options (the project .mxds, base templates and the normal.mxt template), **ArcCatalog has only one place where customizations are stored**, its own **normal.gxt template** (this presents some problems in conveniently distributing ArcCatalog customizations)
- Just as many objects in **ArcMap** have the **Mx prefix** in their name, **Gx** is the **common prefix for ArcCatalog**

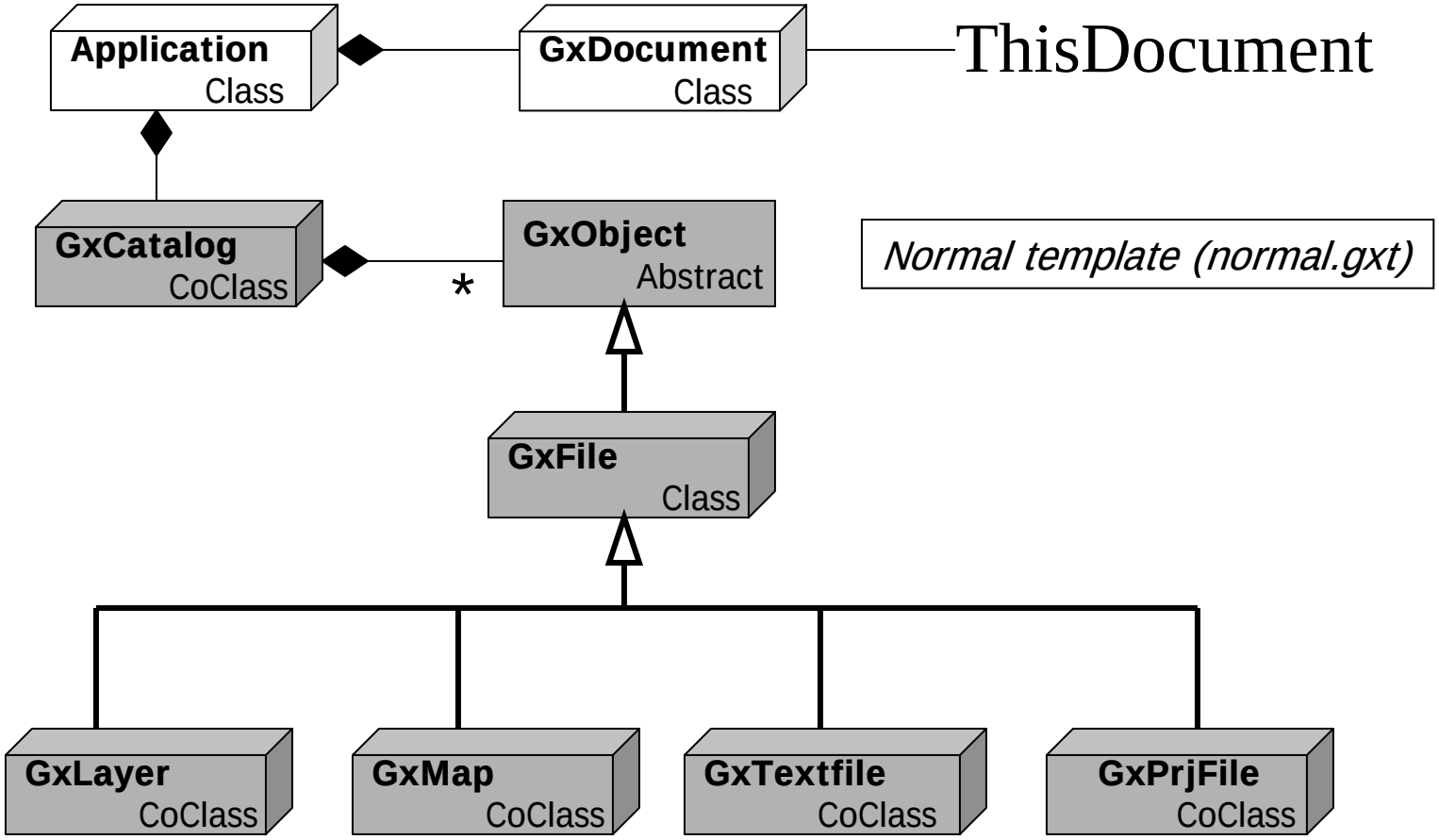
Chapter 16 – Using ArcCatalog objects in ArcMap

ArcMap Object Model Diagram



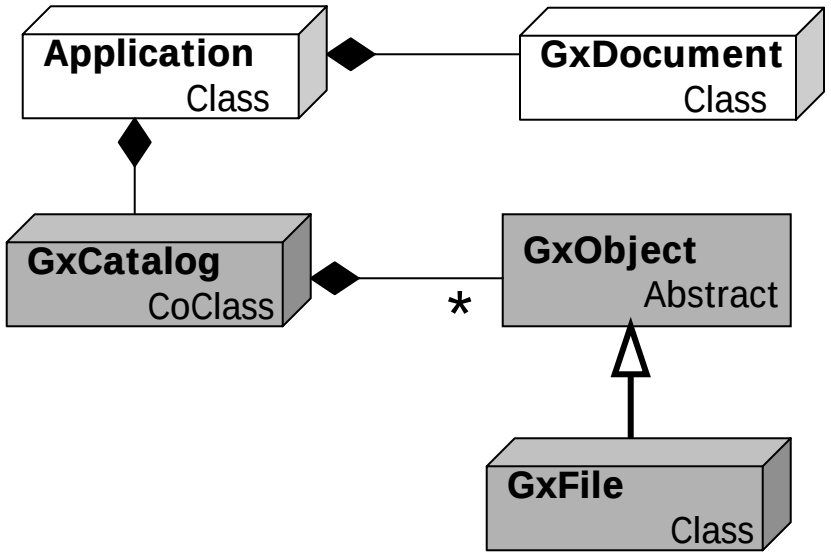
Chapter 16 – Using ArcCatalog objects in ArcMap

ArcCatalog Object Model Diagram

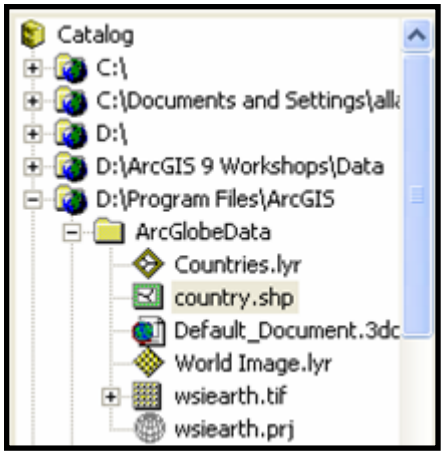


Chapter 16 – Using ArcCatalog objects in ArcMap

- The ArcCatalog Application is composed of **GxCatalog objects**, which in turn are composed of **GxObjects**
- A GxObject is **any file, folder, disk connection, or other object you can click on in the tree view** shown in the left-hand pane of ArcCatalog

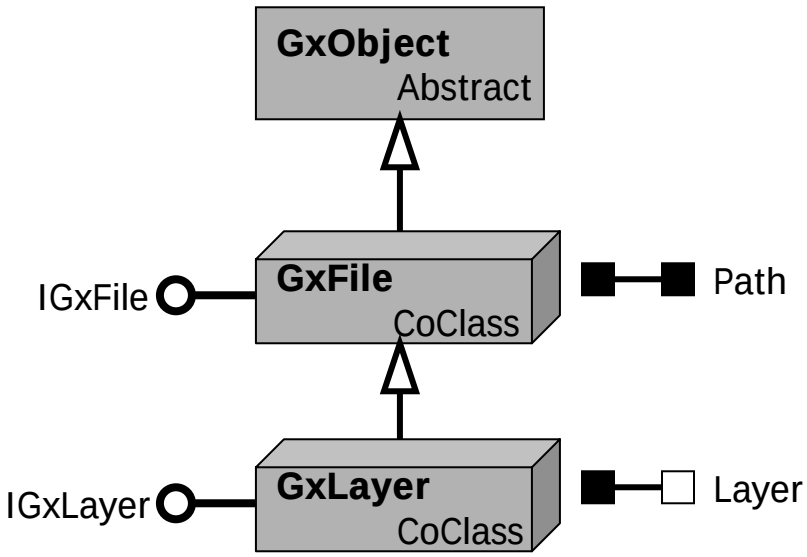


Several kinds of **GxObjects**, shown in the tree view



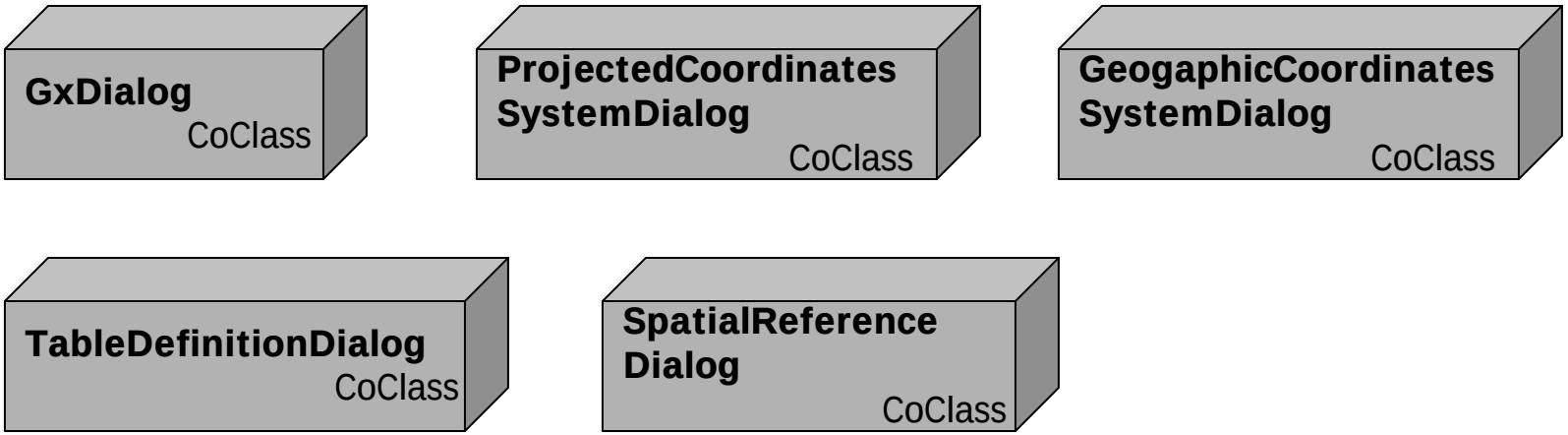
Adding layer files to ArcMap

- A **layer file (extension .lyr)** acts as an intermediate between a spatial data source and the Map document: It **stores information about symbology, the path to the data set** etc.
  - This **simplifies adding a layer to a Map with a particular symbolization**; it is all set up already
- A **GxLayer** is a **GxFile**, and both are **GxObjects**, and as they are **coclasses**, either can be **created directly**
- To **create one from a file**, use GxFile's **path property**:



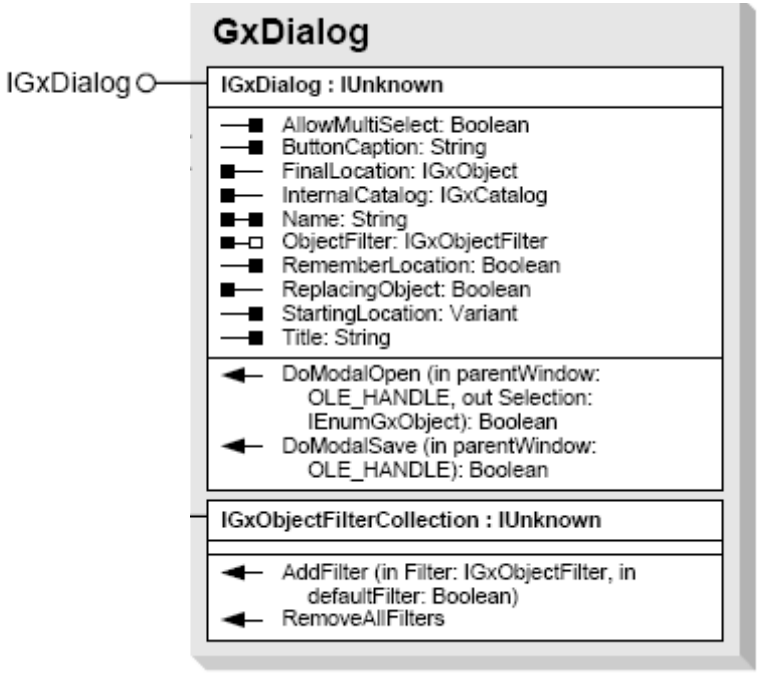
Chapter 16 – Using ArcCatalog objects in ArcMap

- There are **five further coclasses** in the ArcCatalog object model diagram that **represent dialog boxes**
- Each has its uses, but particularly important to us is the **GxDialog**, which gives us the **capability to make customized dialog boxes for specifying files to be opened or saved**



Making your own Add Data dialog box

- In many cases, rather than having a known path to the data we want to add, instead we give the user the chance to **navigate to the correct directory** and **select the data source using a dialog box**
- The **GxDialog** is designed just for this purpose: It allows to create a **file selection dialog box** that we can **customize in various ways** (e.g. to allow specific file types to be selected, single or multiple files selected, what the title and buttons say, what directory it opens in etc.)

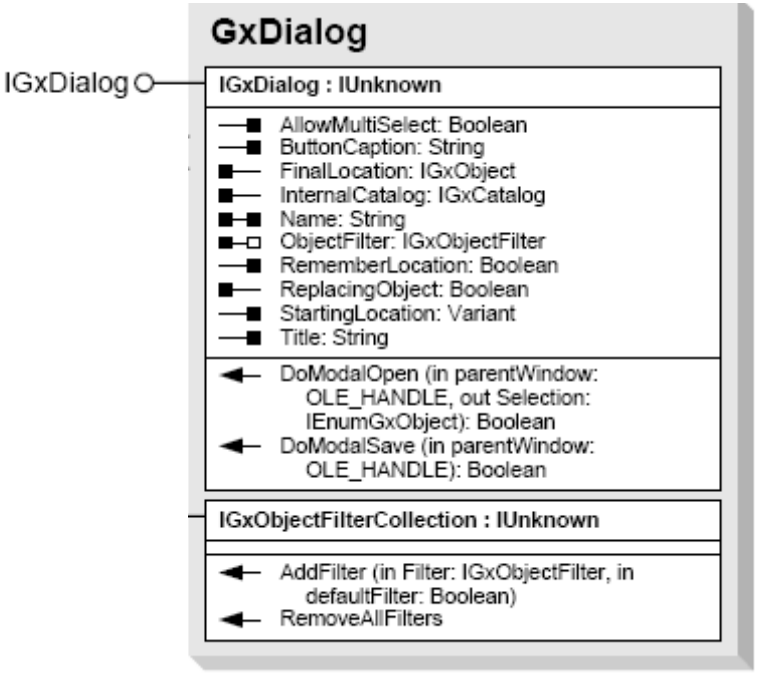


Making your own Add Data dialog box

- **For example**, to create a GxDialog titled “Add Data”, that starts in “Catalog”, with a Button that says “Add”, and only allows the selection of a single file:

```
Dim pGxDialog As IGxDialog
Set pGxDialog = New GxDialog
pGxDialog.ButtonCaption = “Add”
pGxDialog.StartingLocation = _
“Catalog”
pGxDialog.Title = “Add Data”
```

- We can further customize the GxDialog by **restricting the type of files** it can be used to open using an **ObjectFilter**



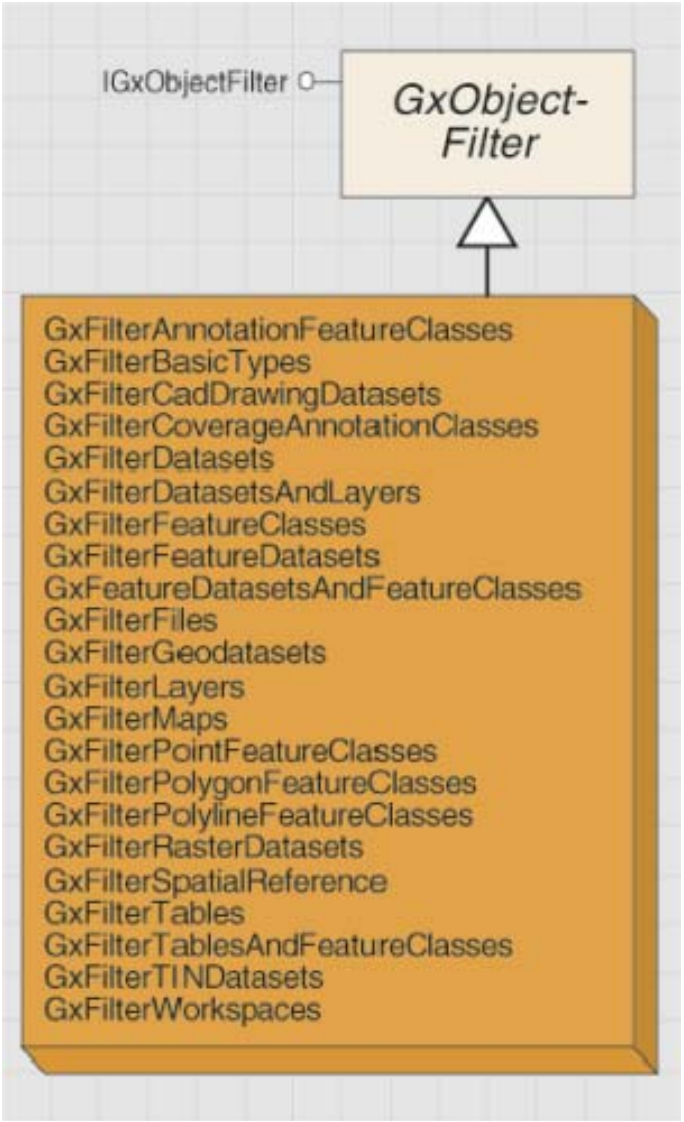
Making your own Add Data dialog box

- There are a **wide variety of types** of **GxObjectFilter** to suit whatever you need your GxDialog to get
- For example to allow our GxDialog to **just open layers**:

```
Dim pLFilter as IGxFilterLayers
Set pLFilter = New GxFilterLayers
```

- We then **set** our GxDialog's **ObjectFilter property** accordingly:

```
Set pGxDialog.ObjectFilter = _
pLFilter
```



Chapter 17 – Controlling feature display

- Making definition queries
- Selecting features and setting the selection color

Chapter 17 – Controlling feature display

- Both definition queries and feature selections are **based on the idea of a query**, which you are undoubtedly familiar with from your previous GIS coursework:
  - Given a **set of features**, can we **identify a subset** of them that **meets a particular set of criteria**
  - E.g.: "Which states in the United States have a population of over twelve million?" which as a query, would read:

    `"State_population > 12000000"`
- A query contains a **field name**, an **operator**, and a **value**
- A **definition query limits the features displayed** to include those that meet the criteria
- A **feature selection** highlights the appropriate features

Making definition queries

- In Exercise 17A, you will use a definition query that **specifies one state in a layer of the United States**, and the user will select which state using a combo box, containing a pull down list of all the states' name attributes
- The **resulting DefinitionExpression** will look like this:

```
pStateLayerDef.DefinitionExpression _
    "State_Name = 'Arizona'"
```

- However, we will need to **use some string operators to form the query**, since we will not know before the fact the name of the state in question (as the user will select it in a combo box)

Making definition queries

- We can obtain the name of the state the user selected in the combo box using the **combo box's EditText property**, and we can store that in a string variable:

```
Dim StrState As String
strState = cboStateNames.EditText
```

- The tricky part is **putting together the full query string**, which can do by **concatenating** several strings together
  - **Concatenation** simply means **attaching multiple strings together**, and it is done in VBA using the **&** symbol
- We know we want the **query string to start with**:

```
"State_Name = '"
```

  - A **single quote inside a string becomes a double quote**

Making definition queries

- We also want the **query string to end with a quote**:

  `“’”`

- We want to **sandwich the state name we stored in strState in between those two parts**, which we can do by **concatenating** the three pieces like so:

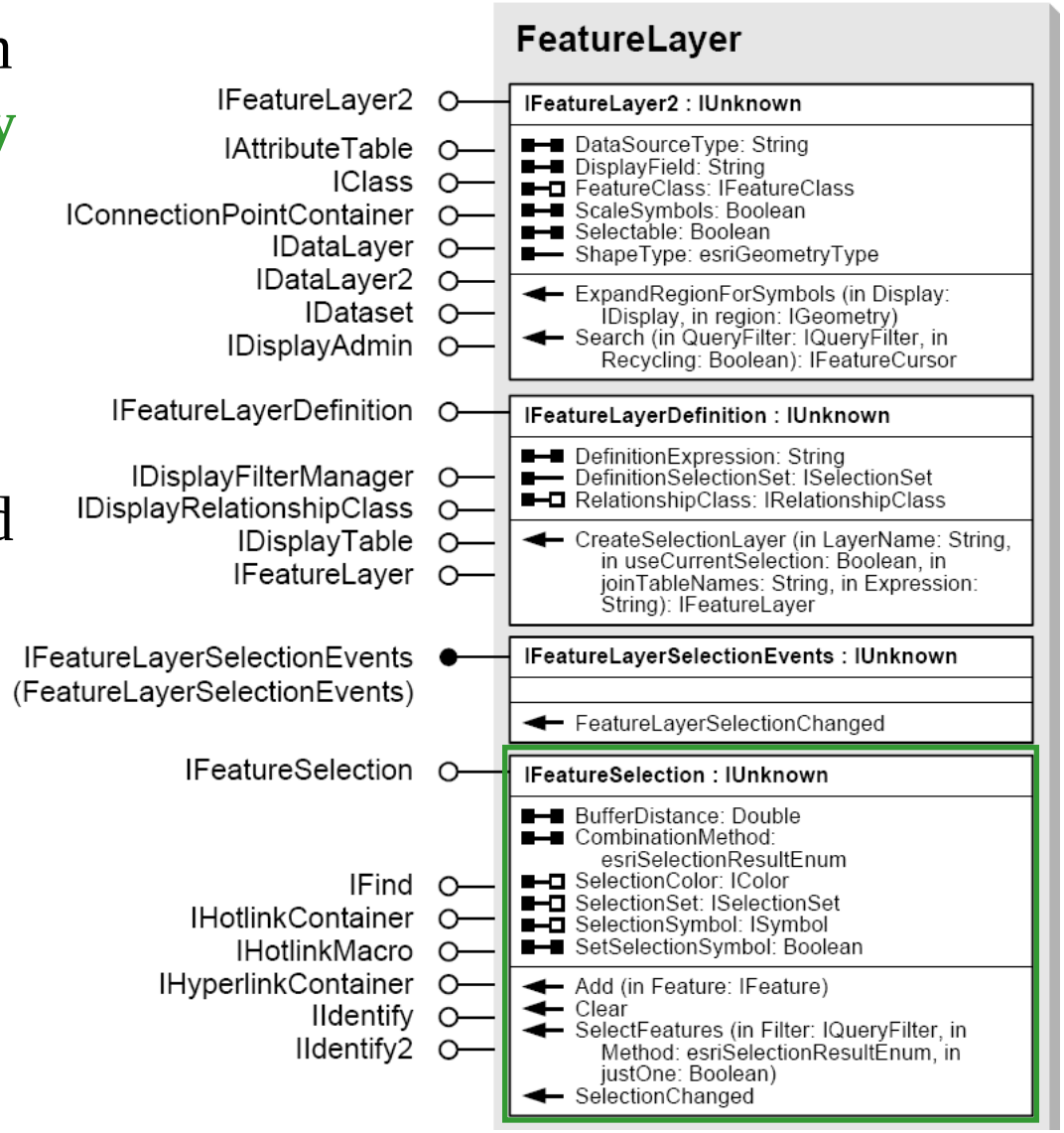
  ```
  “State_Name = ‘” & strState & “’”
  ```

- Altogether, that **makes a single string** that we want to use for the definition expression, which we can declare and store, and then use:

  ```
  Dim strQuery As String
  strQuery = “State_Name = ‘” & strState & “’”
  pStateLayerDef.DefinitionExpression = strQuery
  ```

Selecting features and setting the selection color

- **Selecting features** works in a **similar fashion**: A **query is used to specify what to select**, although it uses different objects, interfaces and properties
- The **SelectFeatures** method on the **IFeatureSelection** interface is one way to make a feature selection
- This method requires a **query filter**, a **selection method**, and the **justOne argument**



Selecting features and setting the selection color

- A **QueryFilter** is an object that can be **used to build and store query statements**
  - The query string is stored in the **WhereClause** property:

```
Dim pFilter As IQueryFilter
Set pFilter = NewQueryFilter
pFilter.WhereClause = “State_Name = ‘Arizona’”
```

- There are five types of **selection methods** that can be used for the **second argument** of the **SelectFeatures method**:
  - esriSelectionResultNew – Create **totally new** selection
  - esriSelectionResultAdd – **Add features** to current selection
  - esriSelectionResultSubtract – **Remove features** from current selection
  - esriSelectionResultAnd – **Select features** from current selection
  - esriSelectionResultXOR – **Reverse status** of features satisfying query

Selecting features and setting the selection color

- The **justOne argument** of the **SelectFeatures method** is a Boolean argument that **specifies whether to find**:
  - **The first feature** that satisfies the query (when true) OR
  - **All features** that satisfy the query (when false)
- **Putting all three arguments together**, the code that would use the SelectFeatures method with a QueryFilter called pFilter, performing a query where the results are used in an entirely new selection, and would only look for the first feature that satisfies the query would be:

```
pFSLayer.SelectFeatures _
    pFilter, esriSelectionResultNew, True
```

Chapter 18 – Working with selected features

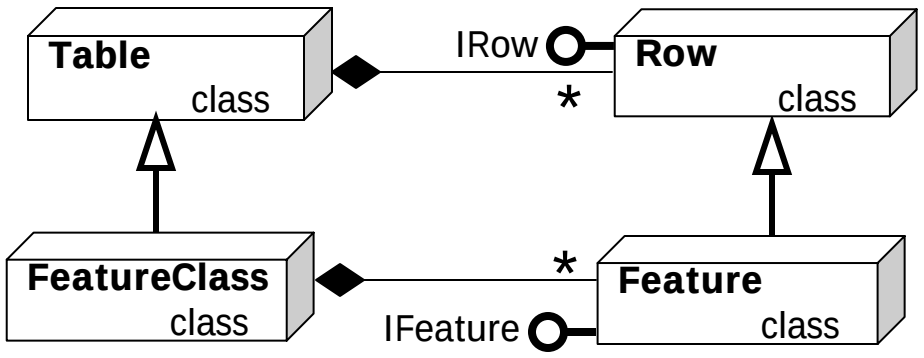
- Using selection sets
- Using cursors

Chapter 18 – Working with selected features

- Now that we know **how to select a set of features**, we will next learn **how to do something** with them
- **Selection sets** collect selected features as a **group**
  - A selection set is a **container** for a set of features
  - Like all **collection objects** we can **add and remove items**
  - **Unlike other collections** we have worked with, you **CANNOT access particular objects** in the selection set
  - One **important property** a selection set does have is a **Count property** to **report the total number of features** it contains
- To work with **selected features one at a time**, you make a **cursor**
  - This usage of the word cursor is **different** from indicating the **position of text** being edited in Word

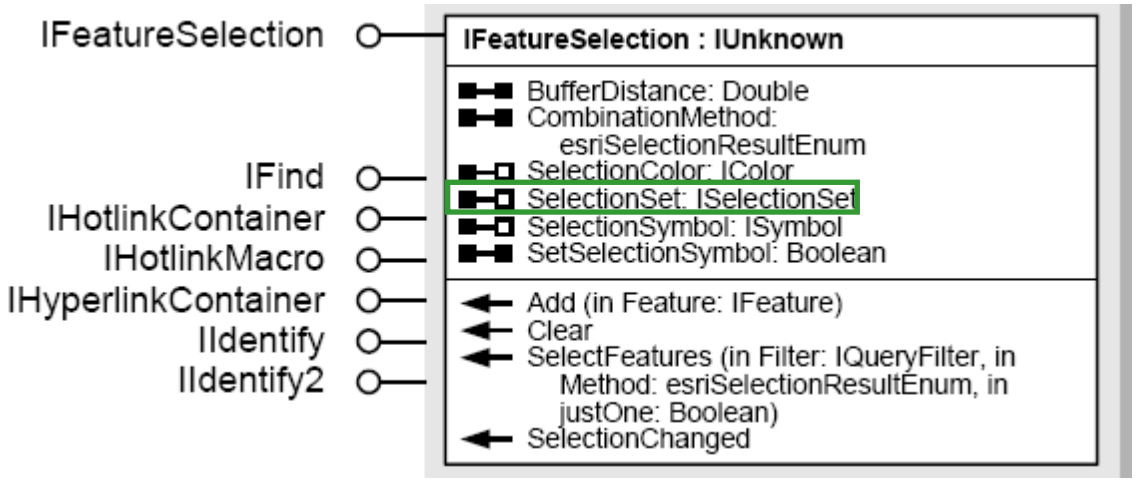
Chapter 18 – Working with selected features

- A **cursor** is like an Enum, with **a pointer** and **method to move from one object to the next** (e.g. in a selection set)
- It can be used to **obtain and modify a feature's spatial and attribute information**
  - When it comes to **editing features** to store (for example) the results of some analysis you just performed using VBA code that you wrote, **a cursor is used** to write results to feature datasets
- Selection sets and cursors are **made up of records**
  - Records refers to both **rows in a table** and **features in a feature class** (each of the latter is **composed** of several of the former)



Using selection sets

- Every feature layer has a **SelectionSet property**
  - Even **if nothing is selected**; it is still there, just **empty**



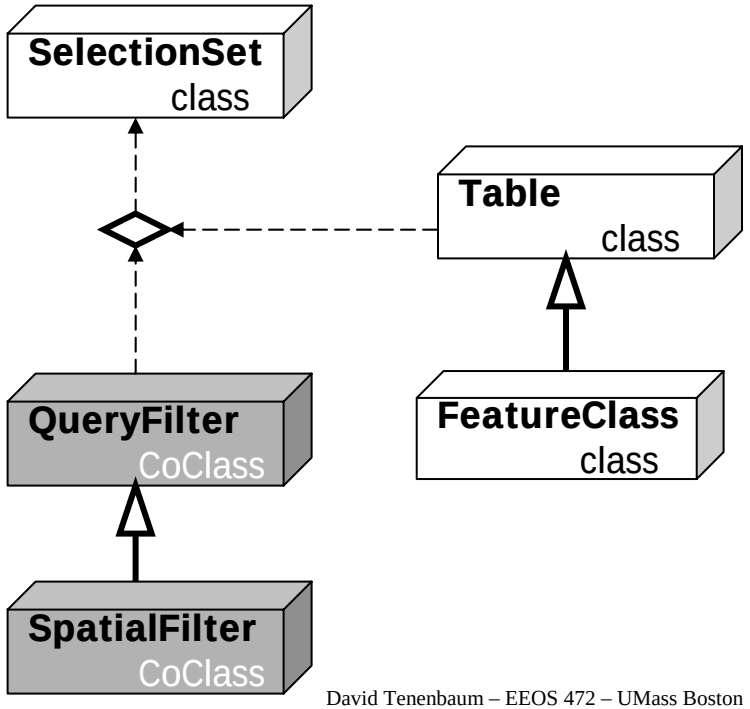
- Whether **user-defined** (using parts of the GUI like the **Select Features tool** or the **Selection menu**) or **set by code** (using a **QueryFilter** as we saw earlier in this class) we can get the selection set by **getting the SelectionSet property** on the **FeatureLayer's IFeatureSelection interface**

Using selection sets

- A feature layer **can have multiple selection sets**, but can **only display one of them at a time**
  - The one displayed is **switched by setting the SelectionSet property**, and then **refreshing the map's active view**

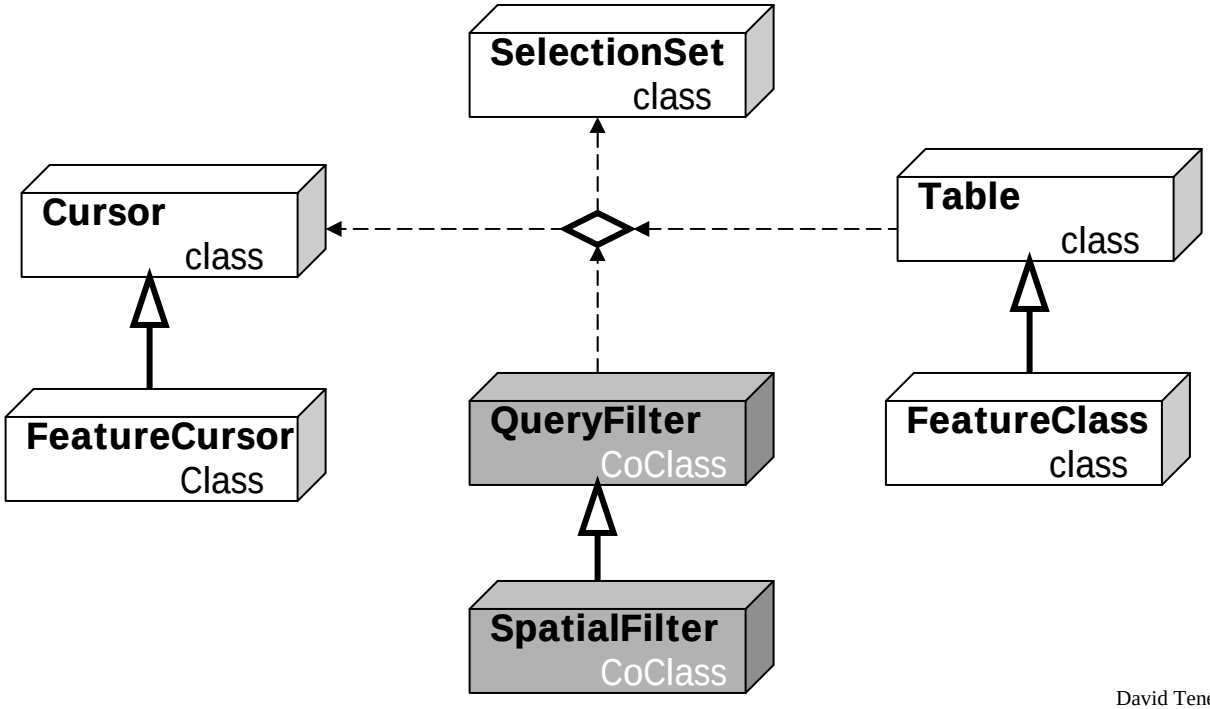
```
Set pFLayer.SelectionSet = pWestSelectionSet
pMxDoc.ActiveView.Refresh
```

- **A Table** and **a QueryFilter** are **both needed** in order to **create a SelectionSet**
  - This is what the **open diamond symbol** in the diagram to the right means (that multiple objects are needed to create another)



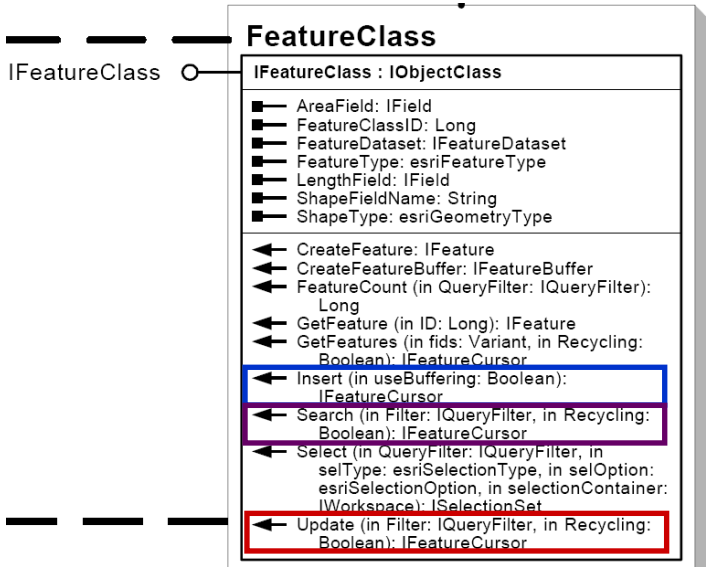
Using cursors

- A **cursor** can be used to **obtain and modify a feature's spatial and attribute information**
  - It is a **group of records organized in rows**, like a table
  - It is **created** using a **query filter** and a **table**
  - A **FeatureCursor** is a type of cursor for use with features



Using cursors

- The **IFeatureClass interface** (which we used to make selection sets) also has **three methods** to **make a feature cursor**:



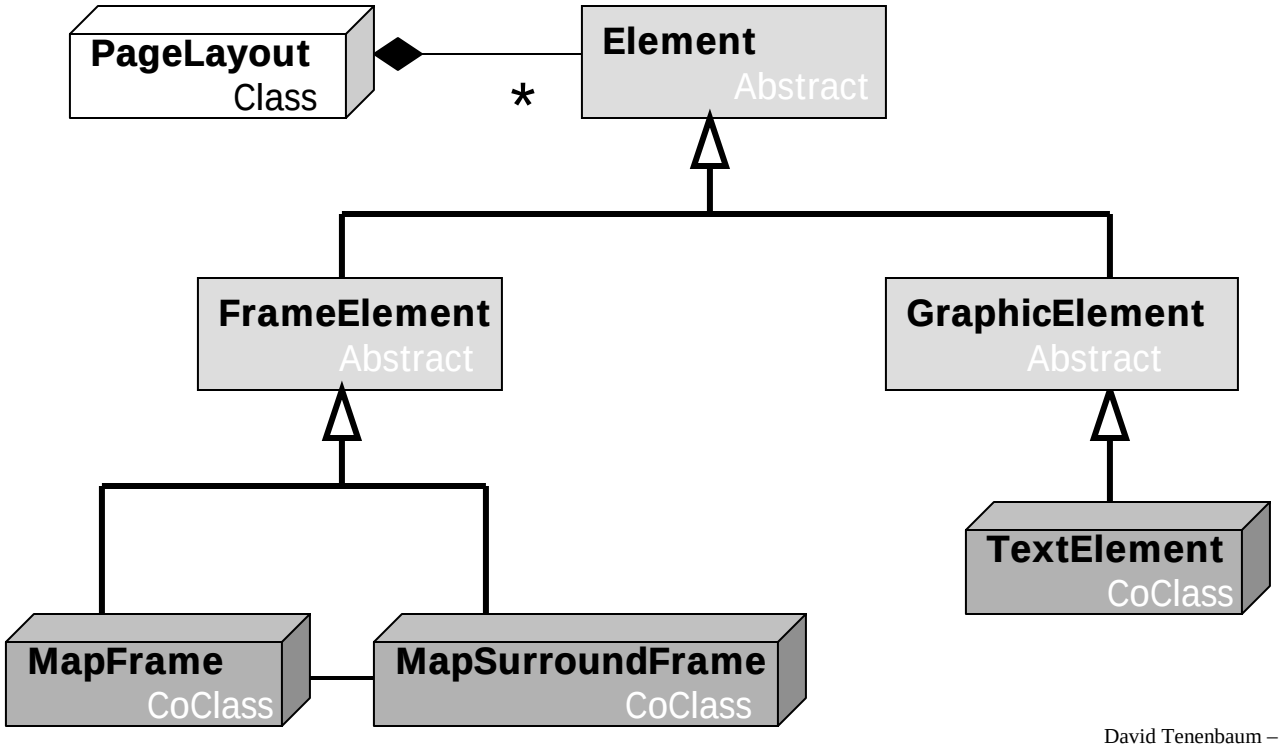
1. The **Insert method** lets you **add new features** to a feature class
2. The **Update method** lets you **edit existing features**
3. The **Search method** makes a cursor that **contains all features satisfying a query statement**
   - This is useful when you want to **get information** about features but **do not want to make any new features**

Chapter 19 – Making dynamic layouts

- Naming elements
- Manipulating text elements

Chapter 19 – Making dynamic layouts

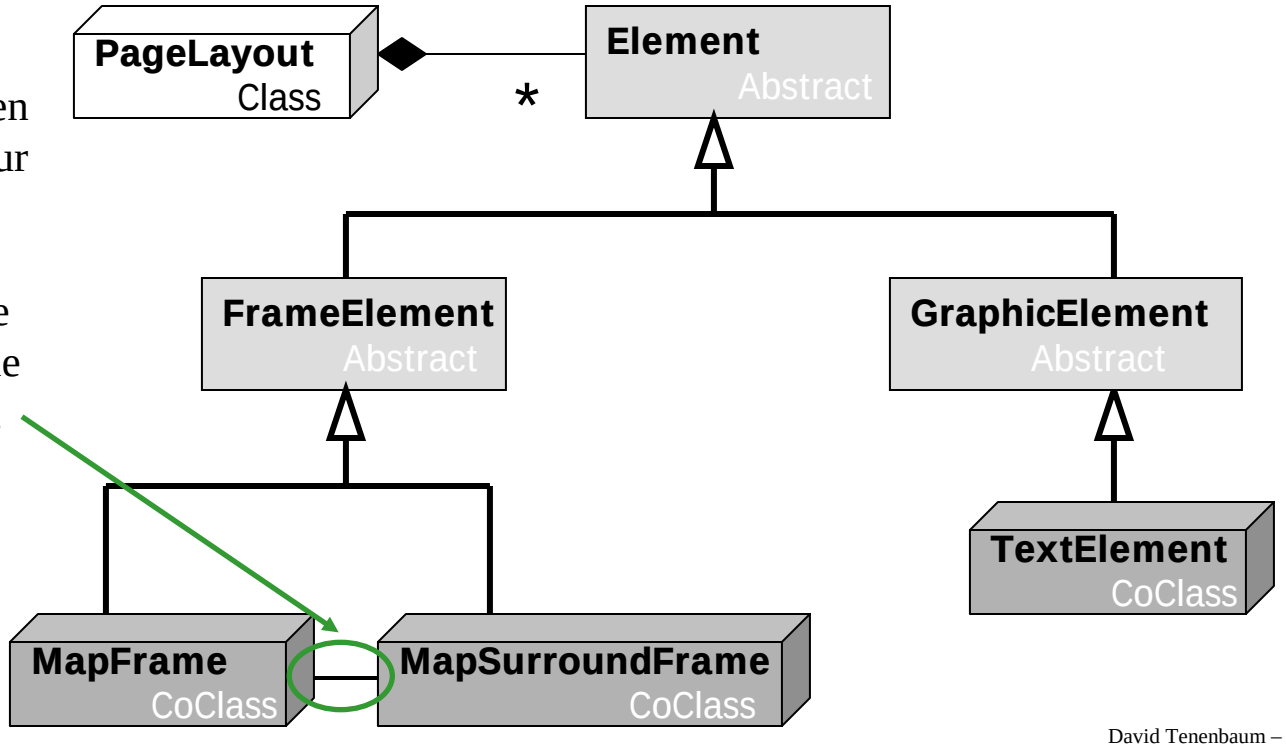
- All the **items found in a map layouts** are, within VBA, objects known as **Elements**
  - The Element class is an **abstract class**, which forms of the basis of **several types of elements** (we used GraphicElements in our Chapter 12 exercises):



Chapter 19 – Making dynamic layouts

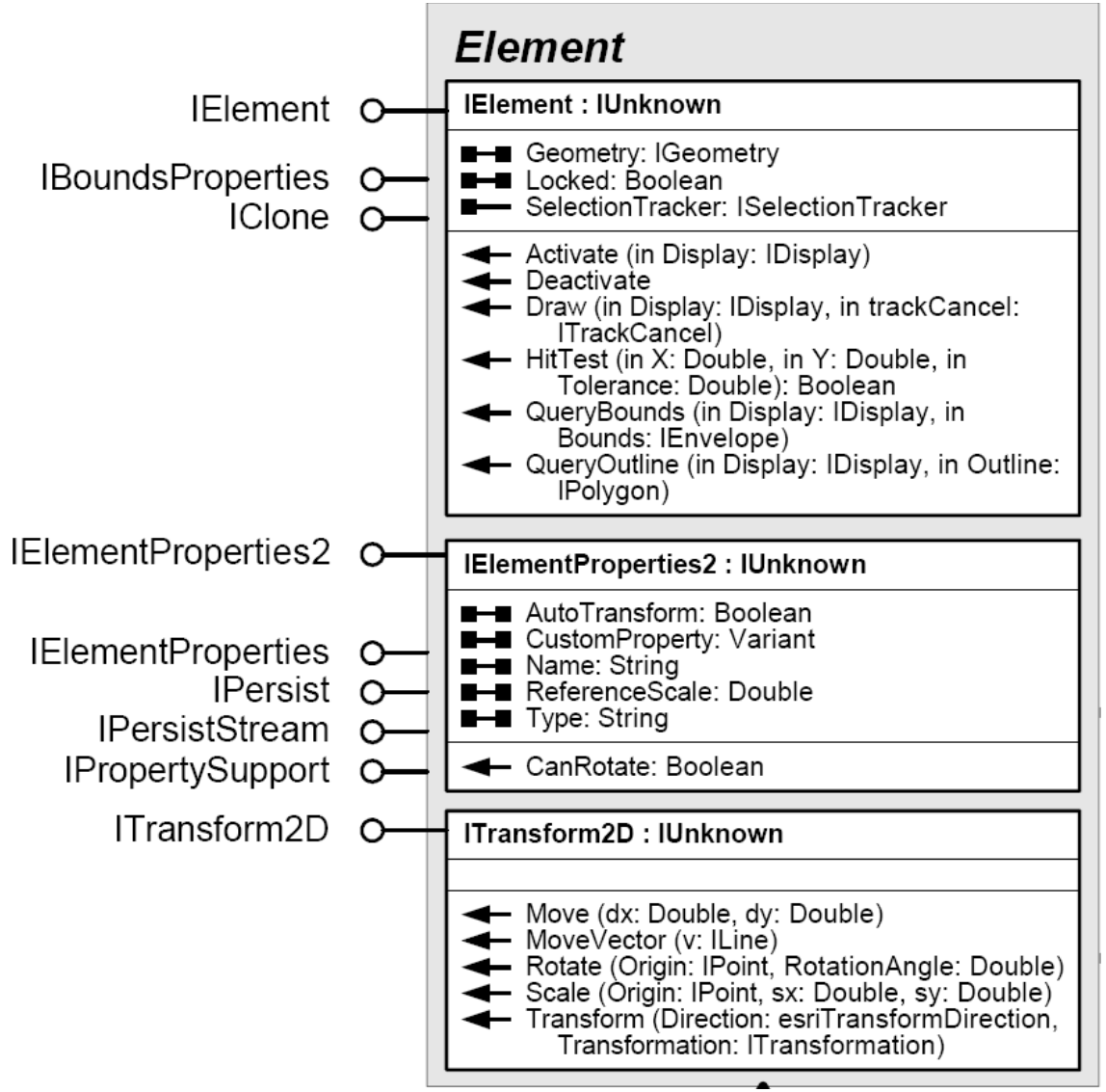
- The split between **FrameElement** and **GraphicElement** is **important**, because they each **behave differently**:
  - **FrameElements** (like data frames and their associated elements) **update to reflect any changes in the map** shown; On the other hand, **GraphicElements do not** … normally they are **static**

This linkage between these coclasses is our indication of their relationship; the MapSurroundFrame will update when the MapFrame updates, for example.



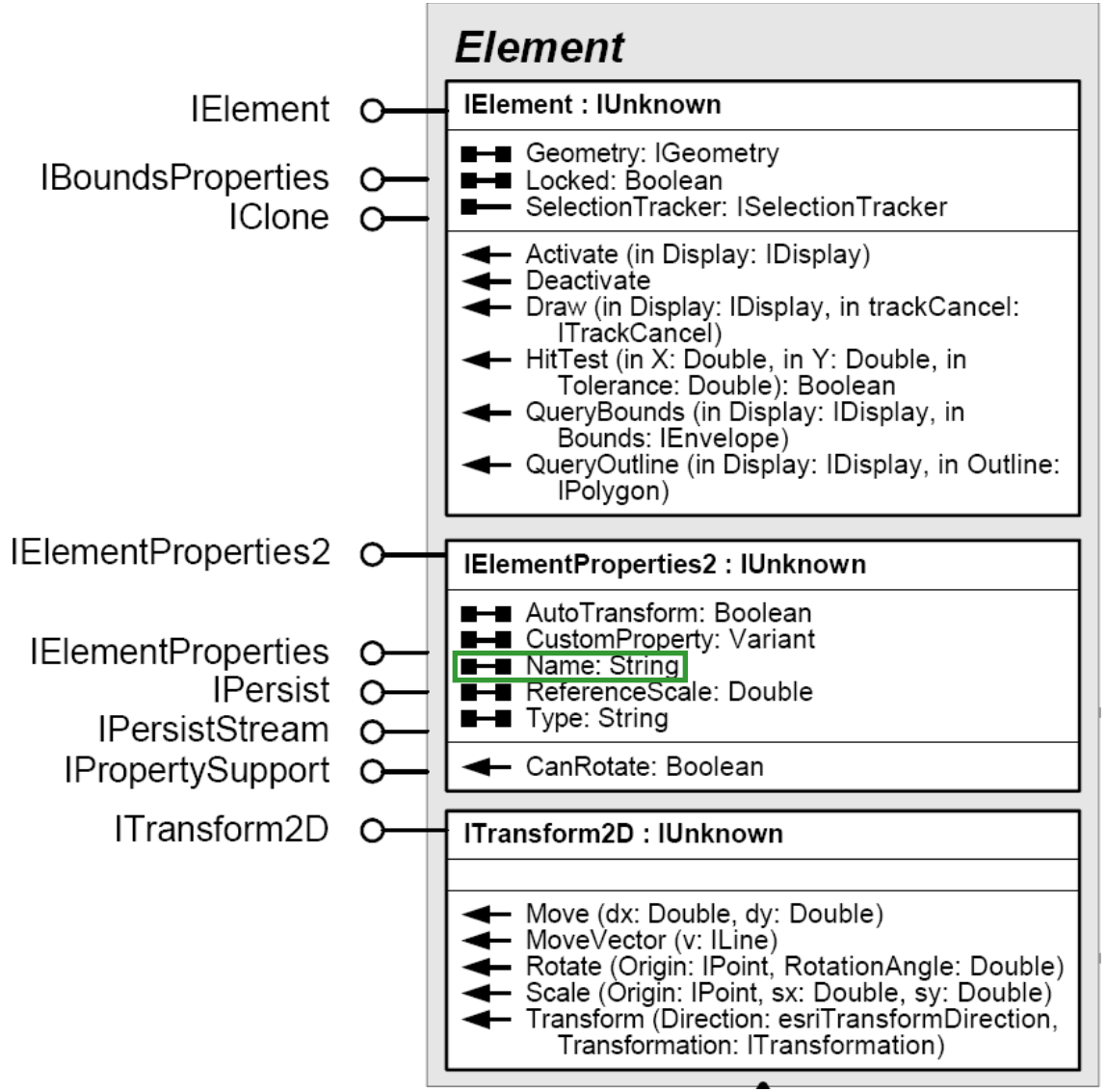
Naming elements

- In this chapter's exercises, you will **change the text elements** in your layout based on some of the code you have developed in previous chapters
- This involves **finding the right elements**, and **updating their properties** according to choices the user makes
- The **tricky part** of this is **identifying the elements you need to change**; this is **easy visually**, but **hard** to do **by code**



Naming elements

- **IElementProperties2** provides a **Name property**
  - Once this has been set, we have an **easy way to find an particular element** within the graphics container
- We will **create buttons to let us get and set element names** to make this convenient for the user



Manipulating text elements

- Once we have got the **functionality set up to get and set our elements' names**, we will make use of it
- We will **use the Name property to find particular elements** by checking through each of the elements that is present in the graphics container to find the right one (based on the name matching)
- We begin by **getting the graphics container** we need, letting VBA do an automatic QueryInterface for us:

```
Dim pMxDoc As IMxDocument
Set pMxDoc = ThisDocument
Dim pGraphics As IGraphicsContainer
Set pGraphics = pMxDoc.PageLayout
```

Manipulating text elements

- We can now **get elements from the graphics container** sequentially using its **Next method**:
  - The Next method returns the **IElement interface** of the element it gets, but we can **use an automatic QueryInterface** to get the interface we really want (**IElementProperties2**, that has the **Name property** on it):

```
Dim pElementProp As IElementProperties2
Set pElementProp = pGraphics.Next
```

- Each time we **get the next element**, we can then **check its name against what we are looking for** using an **If Then (or Case) statement**:

```
If pElementProp.Name = "ToxicMapTitle" Then
```

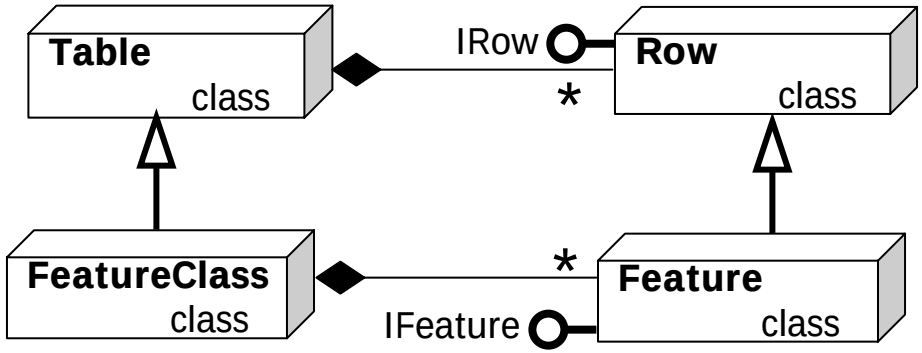
- Once we **find the right one**, we can **set its Text property**

Chapter 20 – Editing tables

- Adding fields
- Getting and setting values

Chapter 20 – Editing tables

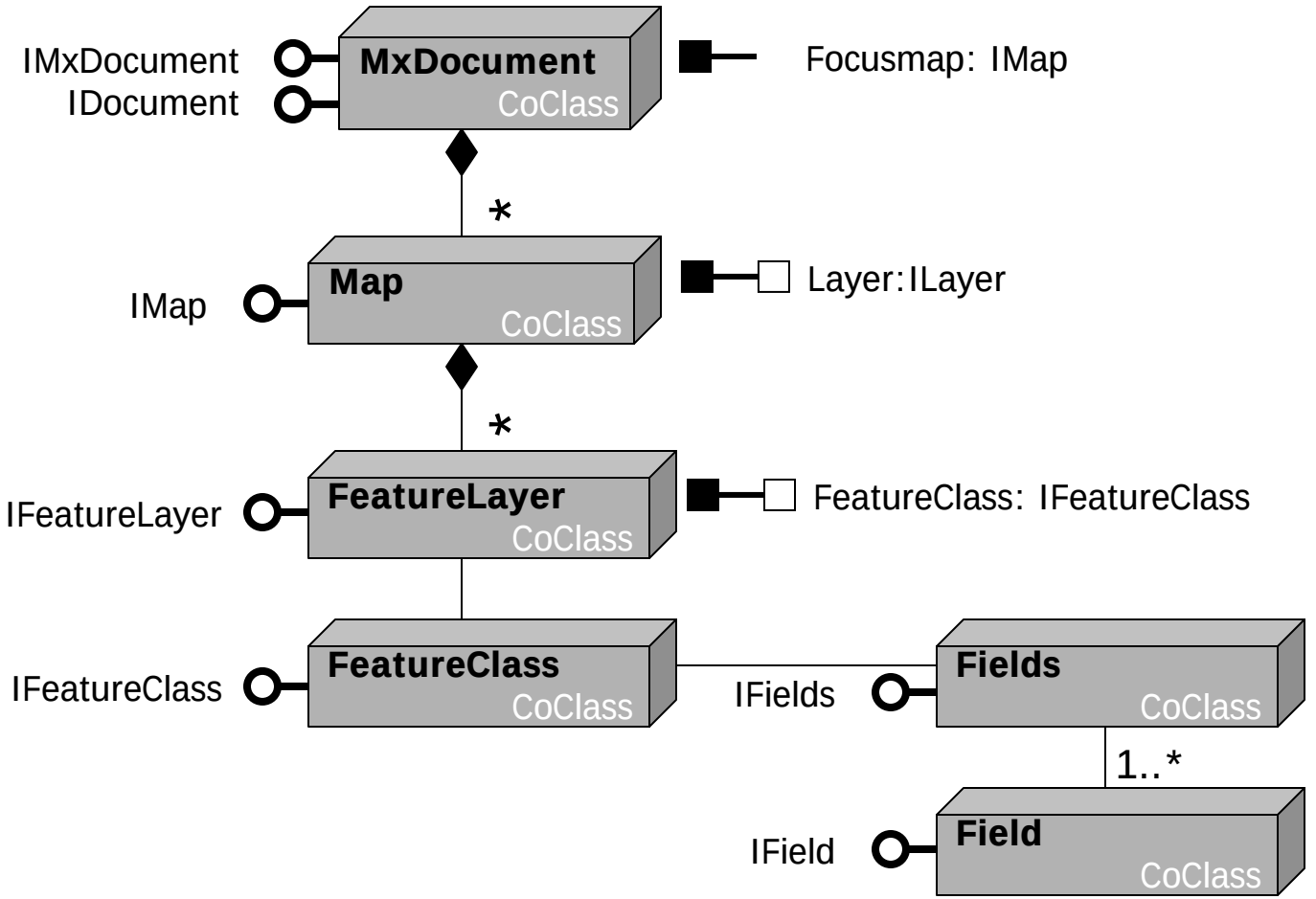
- Recall that the **features** we work with in ArcGIS are actually **stored as records in a** table:



- Tables have a **second dimension** as well: **Columns** in the table represent categories of information. These are **actually stored as fields** in a table
- The **intersection** of a record and a field is a cell; this holds a particular **piece of information known as a value**

Adding fields

- A **feature class** has a **Fields** object, which is a **collection** comprised of **all of its Field** objects:



Adding fields

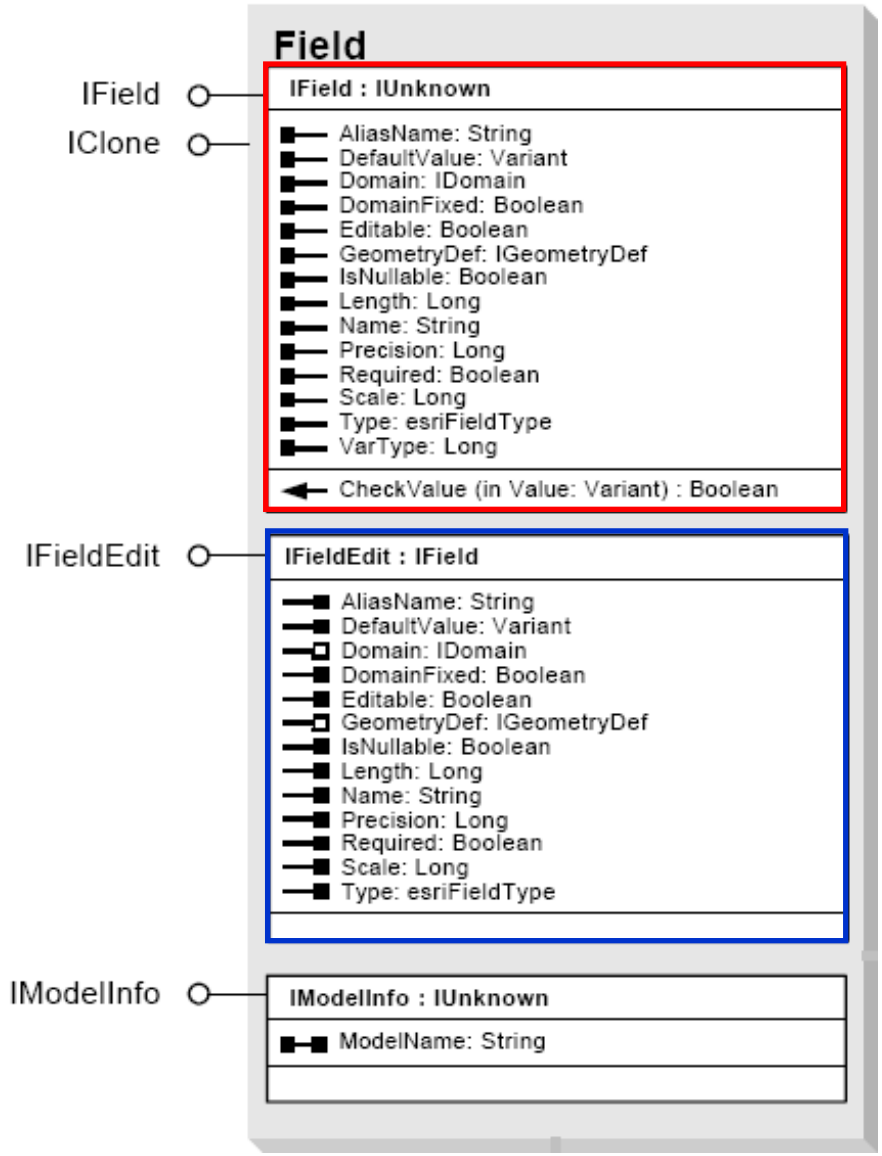
- To **add a field** to a feature class, you first have to **make a new field** from the **Field** coclass in the usual fashion:

```
Dim pField As IField
Set pField = New Field
```

- Once you have **created the field**, you must **set its properties** to make it suitable for the kind of information you want to store within it
  - This needs to be done **before it gets added to table**; once added to a table you can **no longer change the Field's properties**
- The Field coclass has **two nearly identical interfaces**, with **one key difference** between them: One is solely designed for **getting properties**, and the other is for **setting them**

Adding fields

- The **IField interface** has **only left-hand barbells**, so it can only be used to **get a Field's properties**, but not set them
- The **IFieldEdit interface** has **only right-hand barbells**, so it can only be used to **set a Field's properties**, but not get them
- **BUT** … **IFieldEdit inherits from IField**, so … (What does this mean?)



Adding fields

- Once you have made a field, **use the IFieldEdit interface to set its properties**, either by having declared a variable to that interface initially, or by switching to it now:

```
Dim pFieldField As IFieldEdit
Set pFieldEdit = pField
```

- The **two properties** of a Field that you will **always need to set** are its **name** (which is a string) and the **data type**:

```
pFieldField.Name = “Population”
pFieldEdit.Type = esriFieldTypeInteger
```

- There are a number of **field types**, and you can look these up in the help to see which of them you would want to use for a **particular kind of information**

Getting and setting values

- This time, we will make a feature cursor using the **Update method**, and we **do not need to make a query filter** because we want to **get all the records** (rather than a subset of them):

```
Dim pFCursor As IFeatureCursor
Set pFCursor = pFClass.Update(Nothing, False)
```

- We can now move through the records one at a time with the feature cursor's **NextFeature method**:

```
Dim pFeature As IFeature
Set pFeature = pfCursor.NextFeature
```

- The NextFeature method can be **repeated** until the pointer is **pointing at the desired feature**

Getting and setting values

- Once you have the right feature, you can use the **Value property** on the **IRowBuffer interface** to **get or set the value** for a field denoted by an index value, e.g.

  ```
  pFeature.Value(3) = 60000
  ```

  will set the value in the 4th field (remember, the first has index = 0) for the record of interest to 60000
- Once this is done, you have **changed that value in memory**; to make this a **permanent change recorded in the file** corresponding to the table, use the feature cursor's **UpdateFeature method**:

  ```
  pFCursor.UpdateFeature pFeature
  ```
- Use a **Do Until** loop to **change all features** in the cursor