CASE Tools Tutorial

Creating custom features and geodatabase schemas

ESRI

Abstract

This document will guide you through the creation of custom features using the CASE Tools subsystem of ArcInfo 8™ software. A Unified Modeling Language (UML) object model will be created. Based on it, code will be generated to create custom features and then a geodatabase schema.

Table of Contents

TABLE OF CONTENTS ... 2

GETTING STARTED ... 3

WHAT YOU WILL DO ... 3

WHAT YOU WILL NEED ... 3

DESIGNING THE OBJECT MODEL ... 4

EXPORTING A MODEL TO THE REPOSITORY ... 20

GENERATING CODE ... 21

ADDING CUSTOM BEHAVIOR ... 28

CREATING THE SCHEMA ... 39

USING THE CUSTOM FEATURES IN ARCMAP ... 47

Getting started

The CASE tools allow you to create custom features that extend the data model of ArcInfo 8. Object-oriented design tools (OOA&D) can be used to create object models that represent the design of your custom features. These tools make use of the UML to create designs. Based on these models the CASE tools will help you create Component Object Model (COM) classes that implement the behavior of the custom features and database schemas in which custom feature properties are maintained.

The CASE tools consist of two major activities: code generation and schema generation. The former is used to create the behavior, while the latter is used to create schemas in geodatabases. In this tutorial you'll create an object model with custom features, create code adding custom behavior, and create a geodatabase schema.

The tutorial focuses on code generation and object model creation to meet such an objective. For a detailed discussion on how to create other geodatabase elements using UML (such as subtypes, domains, connectivity and relationship rules) refer to *Building a Geodatabase*.

What you will do

In this tutorial you'll create two custom features commonly used in cadastral systems: a parcel and a building. For parcel features you'll store the actual parcel value and the sum of the value of all the buildings in the parcel. For building features you'll store the number of floors, height, and building value.

You'll create a relationship class in the geodatabase to keep track of the buildings contained in a parcel. This will permit the custom features to maintain their relationship automatically, for example, when a new building is created or when an existing building is moved.

You'll also create a simple custom validation rule for buildings: the height must be at least the number of floors times 10 feet.

The UML diagram in the following page depicts the object model that will be built during the tutorial.

What you will need

To follow this tutorial you will need ArcInfo 8, a UML design tool such as Visio Enterprise, and Developer Studio 6.0. You will need two to three hours of focused time to complete the tutorial.

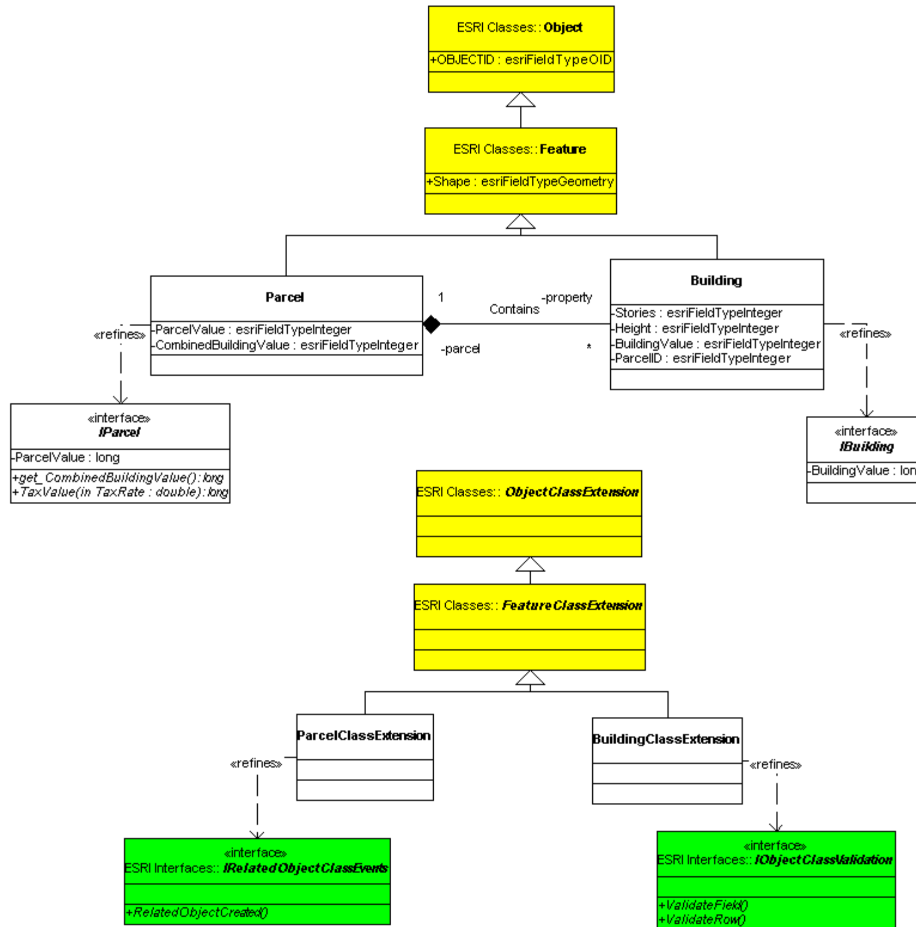
Start by making a copy of the contents of the custom features directory to a working directory, for example, to C:\Temp\Tutorial (the default installation directory is C:\arcgis\arcexe81\ArcObjects Developer Kit\Samples\Geodatabase\Case Tools\Custom Features).

The tutorial directory contains the following:

- A Personal ArcSDE database with spatial data (SampleDB.mdb).
- An empty Microsoft Repository (Repository.mdb).

- A source directory where code will be generated. It already contains a couple files with helper functions.
- A directory with the solution.

The tutorial assumes you are familiar with ArcInfo 8, in particular with the geodata access objects, C++, Active Template Library (ATL) and COM. The figure below shows the object model you will build during the tutorial.



The object model that will be created during the tutorial.

Designing the object model

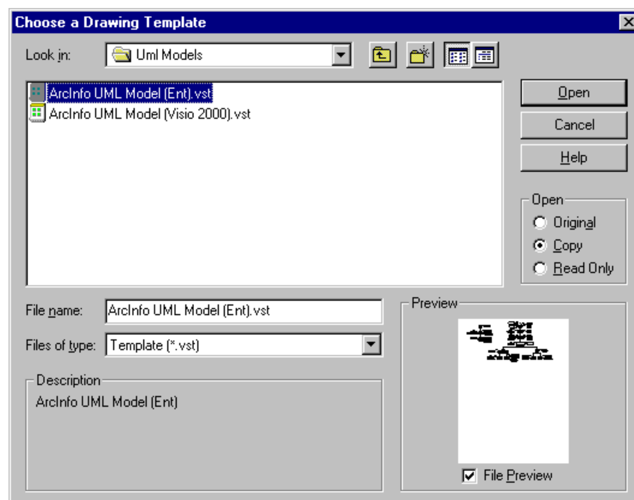
You represent your object models by creating new UML diagrams in Visio Enterprise. This tutorial uses version 5 of Visio Enterprise. However, the instructions can be followed if you are using Visio Enterprise 2000. The following sections will guide you through the creation of the object model.

Creating a new model

In Visio you create a new document by choosing a template. The ArcInfo UML Model template diagram contains the information needed to create your object models.

To create the new document

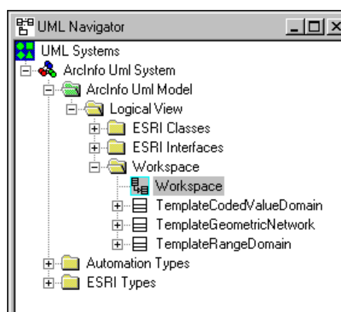
1. Start Visio.
2. Double-click the ArcInfo UML Model (Ent) template file. The default installation path is C:\arcgis\arcexe81\CaseTools\Uml Models (use ArcInfo UML Model (Visio 2000), if you are using Visio 2000).



Creating a new model based on the ArcInfo template diagrams.

The ArcInfo UML Model

The ArcInfo UML Model contains the relevant parts of the geodata access components needed for the creation of custom features. The object model has four packages: Logical View, ESRI Classes, ESRI Interfaces, and Workspace. These UML packages act as directories where different parts of the entire object model are maintained. The Logical View package is the root level and contains the other three packages.



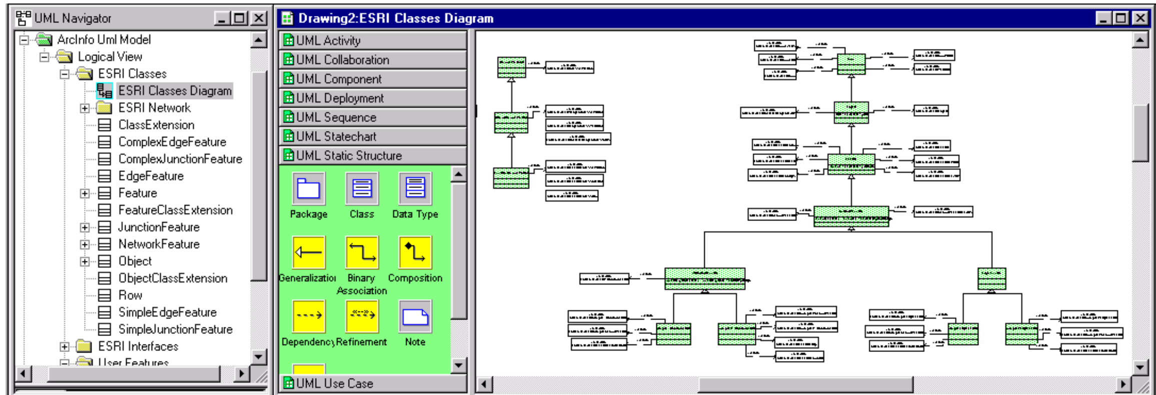
The UML navigator in Visio.

A package contains any number of UML elements, such as other packages, classes, interfaces, and diagrams. Notice you are seeing the Workspace diagram, which belongs to the Workspace package. This package represents the geodatabase. There is no limit on the

number of packages an object model may contain, but all must be created under the Workspace package.

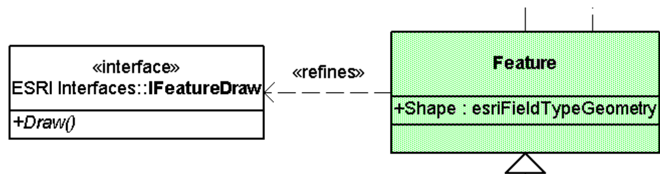
To open the ESRI Classes diagram:

1. Click the plus sign by the ESRI Classes package.
2. Double-click the ESRI Classes diagram.



The ESRI Classes diagram in the template diagram.

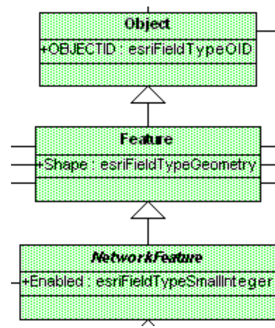
As mentioned before, the UML classes in this diagram represent COM classes that belong to the geodata access components of ArcInfo. These COM classes provide services through interfaces. For example, Feature implements the interface IFeatureDraw. ArcMap asks a feature to draw itself using the method *Draw* in the interface IFeatureDraw. A UML refinement is used to express the relationship between the class and the implemented interface (Tip: to show the operations in an interface, right-click the interface and then click suppress operations).



A COM class implements interfaces.

Notice classes, such as Row, Object, and Feature, are defined inside the ESRI Classes package. Interfaces such as IFeatureDraw are defined in the ESRI Interfaces package. Diagrams also belong to specific packages; for example, the ESRI Classes Diagram belongs to the ESRI Classes package.

Classes inherit from other classes. For example, Feature inherits from Object, meaning Feature “is a kind of” Object. A UML generalization is used to express this relationship.



Type inheritance of COM classes.

How can a child class be a kind of a parent class? By providing the same services the parent class provides. Since COM classes provide services through interfaces, child classes agree to implement the same interfaces their parent implements. In the sample, Feature implements all the interfaces implemented by Object. This is known as type inheritance.

The generalization relationship is “daisy-chained”, so NetworkFeature implements all the interfaces Feature does including those implemented by Object.

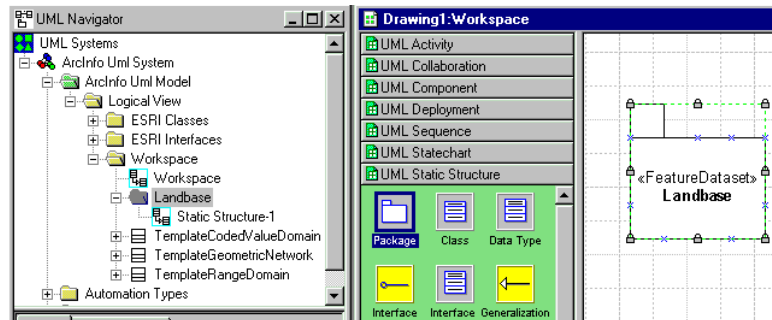
User-defined custom features are objects (non-spatial), simple features, or network features. Parcel and building will inherit from Feature in the tutorial object model, and thus they will agree to be simple features, implementing all the interfaces that Feature does.

Creating a custom feature

You will create a UML class that represents the parcel custom feature. The Schema Wizard will use the information you enter to create a feature class in a geodatabase. It will also be used to generate C++ stub code where custom behavior can be implemented.

As mentioned before, the Workspace package represents the whole geodatabase. Under it you can create feature datasets, tables, and stand-alone feature classes. To hold the parcel and building feature classes you will create a land base feature dataset:

1. In the UML Navigator, double-click the *Workspace* diagram.
2. From the UML Static Structure stencil, drag and drop a package onto the diagram.
3. Double-click the package to open its properties.
4. Type *Landbase* in the name box.
5. Click the Stereotype dropdown and click *FeatureDataset*. Click OK.



Feature datasets are represented in UML as Packages.

In the UML Navigator, notice Visio has created a new diagram for the feature dataset object model. A model may contain many diagrams including more than one per package.

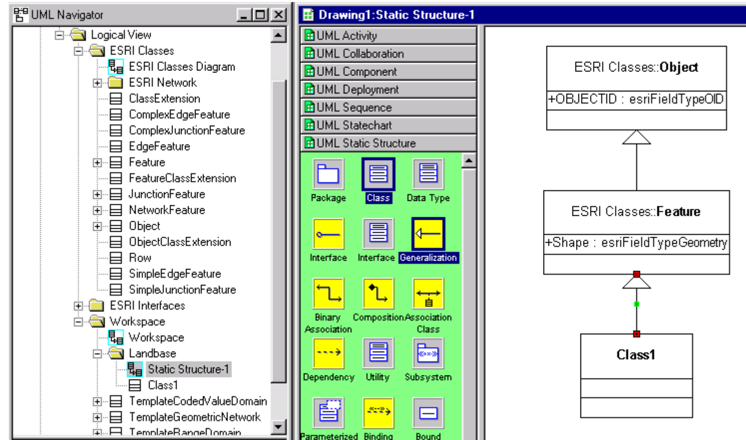
User-defined features should not be defined inside the ESRI Classes or ESRI Interfaces packages. The Workspace package should be used instead (or any other package created under it). To create the parcel custom feature, do the following:

1. In the UML Navigator, double-click on the diagram under the *Landbase* feature dataset.
2. From the ESRI Classes package, drag and drop the *Object* class onto the diagram.
3. From the ESRI Classes package, drag and drop the *Feature* class onto the diagram.

Notice Visio automatically adds a generalization relationship between Object and Feature, as defined in the ESRI Classes diagram. It is easier to think of the model as maintained inside the UML Navigator and displayed through static structure diagrams.

This is why the same object or relationship can be shown in several different diagrams, even if the object and the diagram don't belong to the same package. Notice the classes just added belong to the ESRI Classes package—hence the name ESRI Classes::Object, but the diagram belongs to the User Features package.

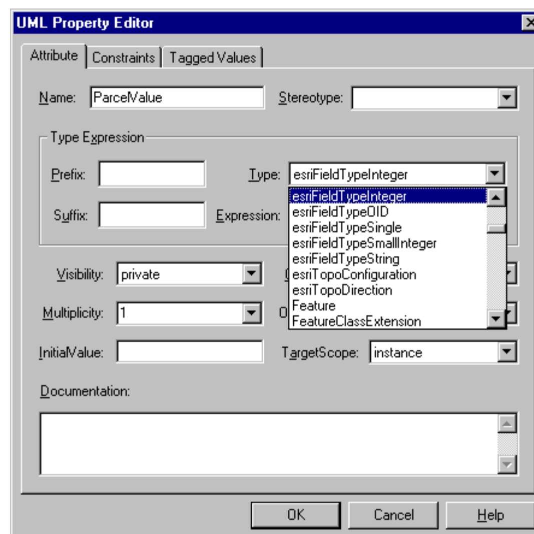
4. From the UML Static Structure stencil, drag and drop a new class onto the diagram.
5. Drag and drop a generalization onto the diagram and connect the new class to Feature.



UML classes are used to represent custom features. They ultimately inherit from a class defined under the ESRI Classes package.

You can set the properties of the newly created class.

1. Double-click the class to open its properties.
2. Type *Parcel* in the name box.
3. Click the Attributes tab.
4. Click New to create a new attribute.



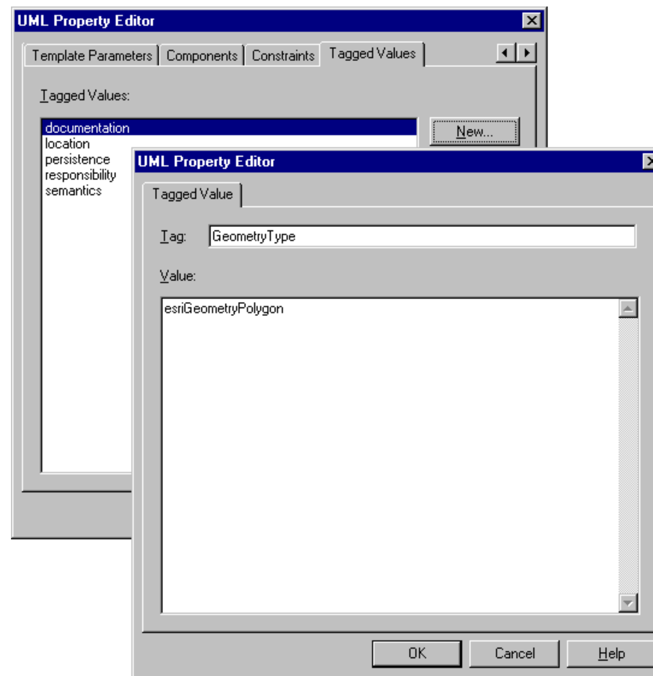
UML attributes are used to represent fields.

5. Type *ParcelValue* in the Name box.
6. Click the Type dropdown and click *esriFieldTypeInteger* to set the field type.
7. Click OK.
8. Repeat steps 9 through 12 to add a second attribute, *CombinedBuildingValue*. Set its type to be esriFieldTypeInteger.

9. Click the Tagged Values tab for the class (you will have to scroll right through the available tabs).

A tagged value is a keyword–value pair that may be attached to any model element. The keyword is called a tag and represents a property applicable to one or many elements. Both the keyword and value are strings, allowing you to attach arbitrary information to models.

1. Click New to create a new tagged value.
2. Type *GeometryType* in the tag box.
3. Type *esriGeometryPolygon* in the value box.
4. Click OK.
5. Click OK.
6. Right-click the class and click Show Properties.



Tagged values are used to specify geodatabase characteristics of the elements in the model, the geometry type of a feature class, for example.

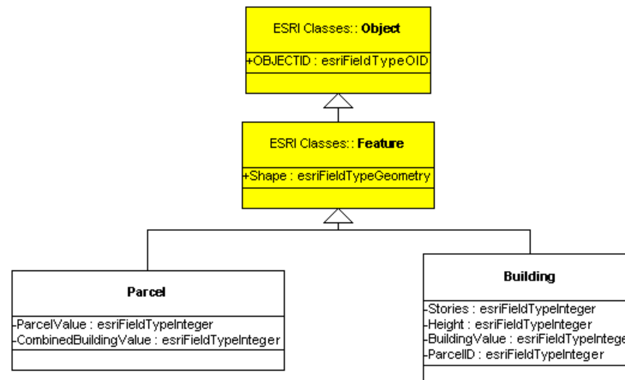
You have created a class that represents the parcel custom feature. The schema wizard will create a feature class based on the information you just entered. The feature class will store polygons as specified by the geometry type tagged value, and the UML attributes (ParcelValue and CombinedBuildingValue) will be used to create two fields in the feature class table.

Notice OBJECTID and Shape are inherited UML attributes (from object and feature, respectively). The feature class will have these two fields as well.

Follow the same procedure (including the steps to make its geometry polygon) to create a second class. Name the class Building and add the following attributes:

Name	Type
Stories	EsriFieldTypeInteger
Height	EsriFieldTypeInteger
BuildingValue	EsriFieldTypeInteger
ParcelID	EsriFieldTypeInteger

The ParcelID attribute will be used to store foreign keys for a relationship class. You'll work with a relationship class in a subsequent section of the tutorial. This is how your model should look.



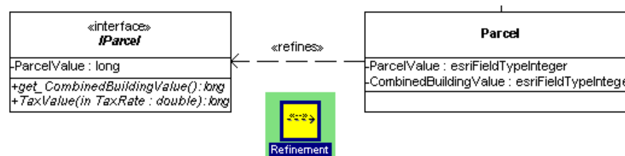
The land base object model.

Creating an interface

As mentioned before, Parcel and Building will behave like features because they will implement all the interfaces implemented by Feature. Clients for such interfaces include ArcMap, ArcCatalog, and the geodata access components themselves.

But what if an application requires calculation of the parcel's tax value based on the buildings it contains, its own value, and a given tax rate? Such calculation could be done by an application querying for the values from the database. But what if the same calculation is needed for several applications? In that case it would be more efficient to perform the calculation inside the Parcel custom feature and provide the service through a user-defined interface.

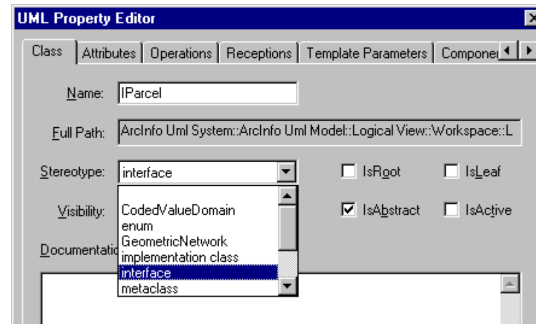
In this step you'll create IParcel, a user-defined interface for the parcel feature. The interface will provide a convenient way to retrieve or write the parcel's value, a read-only property to retrieve the value of all the buildings in the parcel, and a method to calculate the parcel's tax value.



IParcel, an interface through which the Parcel feature will provide custom services.

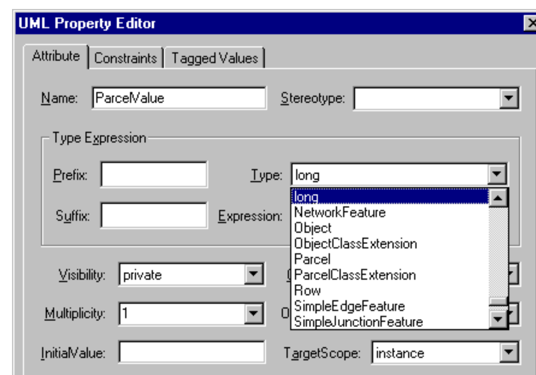
To create the IParcel interface

1. Drag and drop a new class onto the diagram.
2. Double-click the new class to open its properties.
3. Type *IParcel* in the name box.
4. Click the Stereotype dropdown and click *interface*.
5. Click the *IsAbstract* check box.



Interfaces are stereotyped abstract classes.

6. Click the Attributes tab.
7. Click New to add a new attribute.
8. Type *ParcelValue* in the name box.
9. Click the Type dropdown and click *long* to set the attribute type.
10. Click OK.

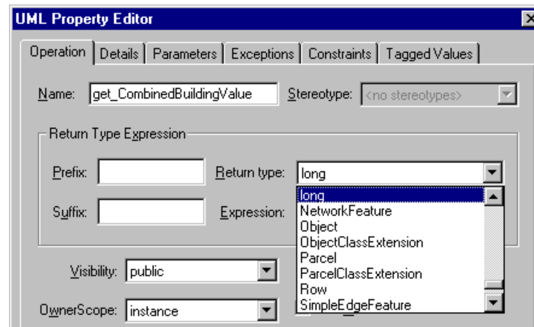


UML Attributes in interfaces are used to create read/write properties.

An operation is a service that an instance of a class may be requested to perform. The behavior of a class is represented by a set of operations. Each operation has a name and a list of arguments. In the next few steps, you'll add a new operation.

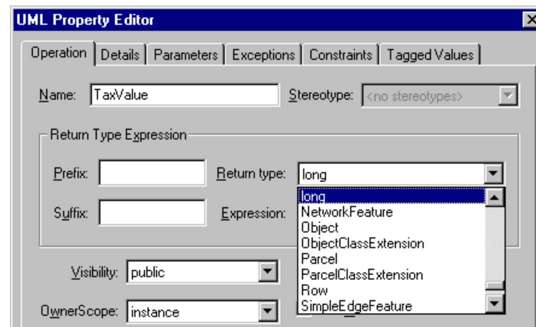
1. Click the Operations tab.
2. Click New to add a new operation.

3. Type *get_CombinedBuildingValue* in the name box (more on the get_ prefix in a subsequent section).
4. Click the Return Type dropdown and click *long* to set the operation's returned type.
5. Click OK.



UML Operations can be used to create read-only or write-only properties.

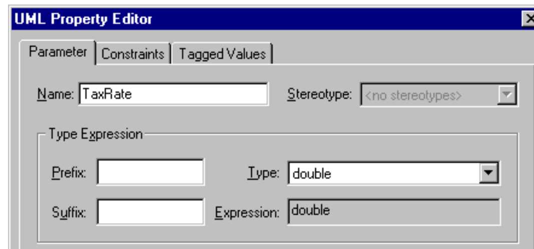
6. Click New to add a second operation.
7. Type *TaxValue* in the name box (Note: the name is case sensitive, so type it as shown).
8. Click the Return Type dropdown and click *long* to set the operation's returned type.



UML Operations can be used to create methods.

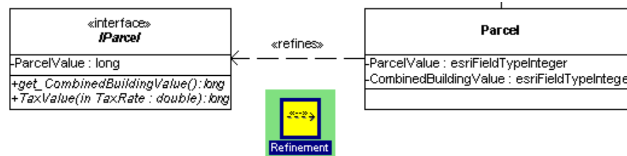
A parameter is an unbound variable that can be changed, passed, or returned. A parameter may include a name, type, and direction of communication. Parameters are used to specify operations, messages, events, templates, and more. In this section of the tutorial, you'll create a new parameter.

1. Click the Parameters tab.
2. Click New to create a new parameter.
3. Type *TaxRate* in the name box (case sensitive).
4. Click the Type dropdown and click to set the parameter type to *double*.
5. Click OK.



UML Operations may have parameters (or arguments).

6. Click OK.
7. Click OK.
8. Drag and drop a Refinement onto the diagram and connect Parcel to its interface.



A UML refinement is used to associate a custom feature with the interfaces it implements.

Notice the types used in custom features (*esriFieldTypeInteger* and the like) are different than the types used in interfaces (*long*, *double*, etc.). The types in custom features are used to create fields in feature classes, whereas the types in the interfaces will be used for code generation and therefore are restricted to C++/Automation types (*esriFieldTypeInteger* is a *long* integer).

A UML attribute in an interface will be code generated into a mutator/accessor pair (or property put/property get pair). For example, this is the Interface Definition Language (IDL) code generated for ParcelValue:

```
[propget]  HRESULT ParcelValue([out, retval] long* ParcelValue);
[propput]  HRESULT ParcelValue([in] long ParcelValue);
```

The get_CombinedBuildingValue UML operation is a read-only property for the CASE tools. The code generation wizard will recognize the 'get_' prefix of the operation and will generate a COM property of the form (IDL):

```
[propget] HRESULT CombinedBuildingValue([out, retval] long * value);
```

It becomes a read-only property because there is no propput being generated. Similarly, the prefix 'put_' can be used to create write-only properties.

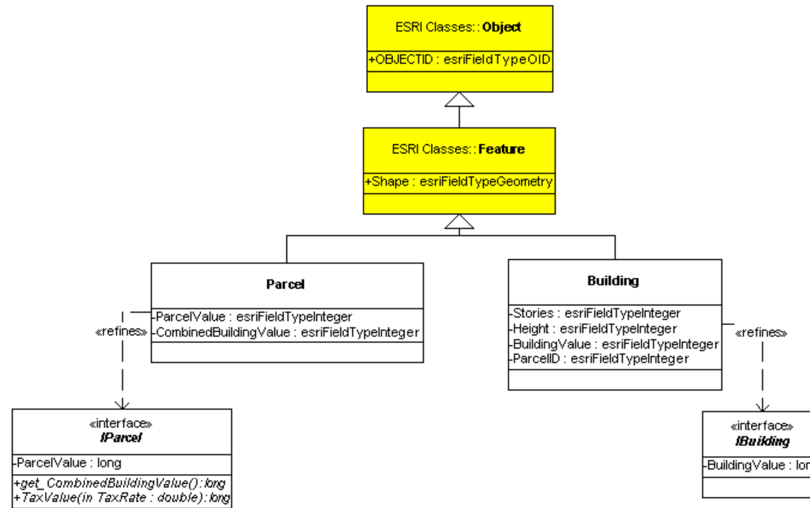
The TaxValue UML operation is a standard COM method. Its IDL signature looks like:

```
HRESULT TaxValue([in] double TaxRate, [out, retval] TaxValue);
```

The building class will also have a user-defined interface. This interface will provide a convenience accessor/mutator pair to the BuildingValue property.

To create the IBuilding interface

1. Repeat steps 1 through 5 to create the interface, use *IBuilding* as the name.
2. Repeat steps 6 through 10 to add a new attribute, use *BuildingValue* as the name and *long* as the type.
3. Click OK.
4. Drag and drop a Refinement onto the diagram and connect Building to its interface IBuilding.

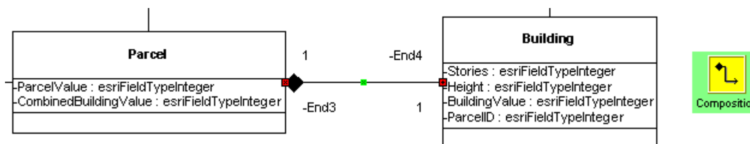


The land base object model with custom interfaces.

Creating a RelationshipClass

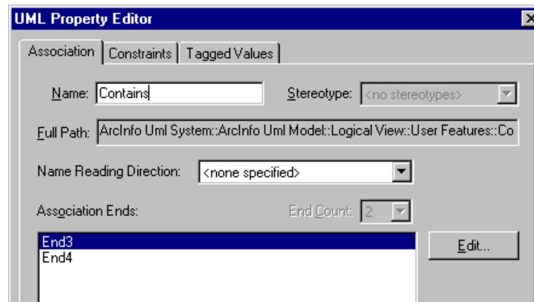
In order to maintain the appropriate relationship between features, you must create the relationship class *Contains*, which associates parcels to buildings.

To create the relationship class



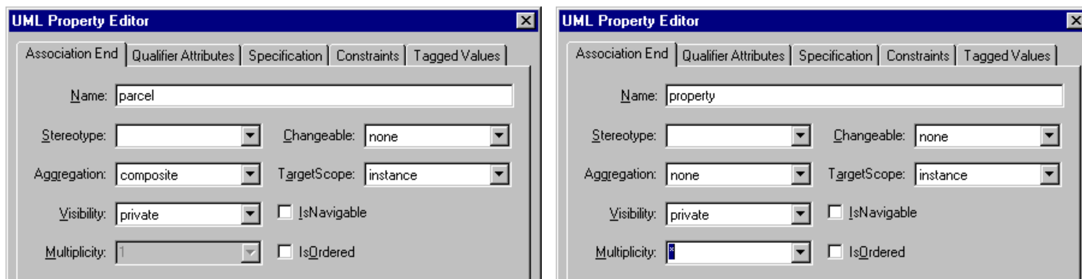
Geodatabase relationship classes are represented with UML associations.

1. Drag and drop a new composite association onto the diagram.
2. Connect the left-hand side of the association to the Parcel class and the right-hand side to the building class.
3. Double-click the association to open its properties.
4. Type *Contains* in the name box.



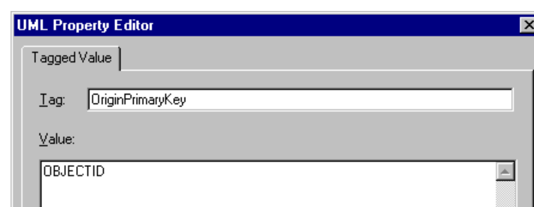
Relationship classes are named.

5. Double-click the End3.
6. Type *parcel* in the name box (notice the multiplicity is 1).
7. Click OK.
8. Double-click the End4.
9. Type *property* in the name box. Click the multiplicity dropdown, then click *.
10. Click OK.



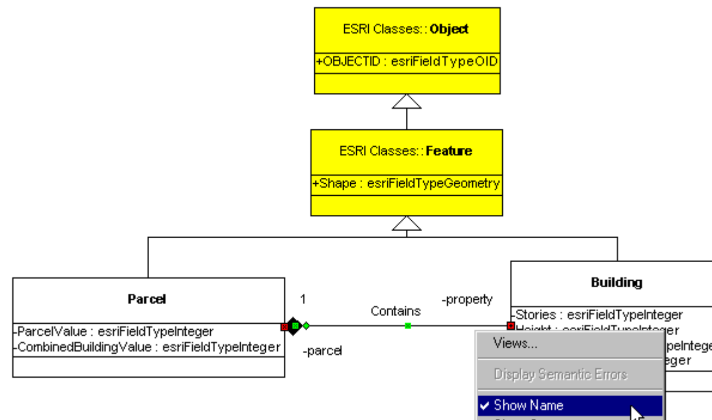
UML Association ends are used to define properties of the relationship class, such as cardinality and forward and backward label names.

11. Click the Tagged Values tab.
12. Click New to add a new tagged value.
13. Type *OriginPrimaryKey* in the tag box.
14. Type *OBJECTID* in the value box.
15. Click OK.



Tagged Values of the UML Association are used to define the primary and foreign keys of relationship classes.

16. Repeat steps 12 through 15 to create a second tagged value. Use *OriginForeignKey* as the tag and *ParcelID* as the value.
17. Repeat steps 12 through 15 once again to create a third tagged value. Use *Notification* as the tag and *esriRelNotificationBoth* as the value.
18. Click OK.
19. Right-click the association, then click Show Name.



Display characteristics of UML elements are available as context menu options in Visio.

The schema creation wizard will use the information in the UML association to create a relationship class in the geodatabase named Contains. The relationship class type will be composite (in contrast to simple), its cardinality will be one to many (1-M), and both objects will be notified when a peer object is changed (notification).

In a composite relationship, one of the objects controls the lifetime of the associated objects. In the tutorial case the parcel controls the lifetime of the buildings, so when a parcel is deleted, the buildings inside will be deleted as well. This behavior is not present in peer-to-peer or simple relationship classes.

Relationship classes are implemented in the underlying geodatabase using primary and foreign key fields in the feature classes' tables. The primary key for the relationship is the parcel's OBJECTID field (inherited from Object in the model), and the foreign key is the building's ParcelID field.

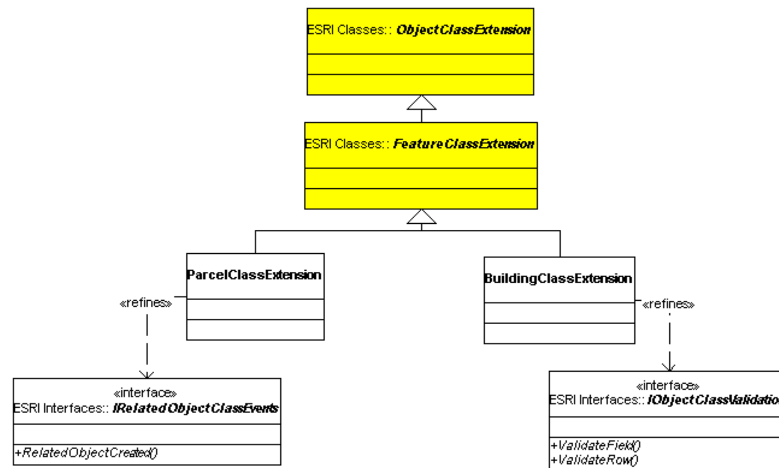
Creating a ClassExtension

A class extension is a COM class that implements behavior that pertains to the whole set of features in a feature class, in contrast to behavior that belongs to a singular feature. For example, the property inspector of a feature class is implemented by the class extension. The same property inspector is used for all features in the feature class.

CASE tools will find class extensions by naming convention. A valid class extension name is made up of the class name followed by the string 'ClassExtension' (for example, BuildingClassExtension).

Class extensions are created in the same way custom features are created, by defining new UML classes that inherit from classes in the ESRI Classes diagram. The building class extension will contain the code for a custom validation rule. To create it, do the following:

1. From the ESRI Classes package, drag and drop *ObjectClassExtension* onto the diagram.
2. From the ESRI Classes package, drag and drop *FeatureClassExtension* onto the diagram.
3. From the UML Static Structure stencil, drag and drop a new class onto the diagram.
4. Drag and drop a generalization onto the diagram and connect the new class to FeatureClassExtension.
5. Double-click the class to open its properties.
6. Type *BuildingClassExtension* in the name box.
7. Click OK.
8. From ESRI Interfaces package, drag and drop *IObjectClassValidation* onto the diagram.
9. Drag and drop a new refinement onto the diagram and connect the new class extension to the interface (notice the direction of the relationship).



Class extensions for Parcel and Building in the UML model.

The geodata access components allow for the specification of rules features must comply with. There are different types of rules, for example, a range domain specifies the minimum and maximum values a field can have. Other types of rules include coded value domains, connectivity rules, and relationship rules. All these rules are defined through parameters.

However, there are rules that can't be defined through parameters easily, for example, a rule that includes some sort of spatial constraint (no industry on residential zones) or a rule that involves the value of several feature properties (height >= stories * 10ft). These custom validation rules can be created in the class extension of a feature class as the implementation of IObjectClassValidation.

When validation is requested, ArcInfo first verifies the rules defined through parameters (domains, connectivity, etc.), then it looks for a class extension. If the class extension is found, and it implements IObjectClassValidation, the methods in the interface are called.

Follow the same procedure to create the parcel class extension. This time grab the IRelatedObjectClassEvents interface from the ESRI Interfaces package. The RelatedObjectCreated method will be used to set the relationship between a new building and its parent parcel. It is the parcel class extension that is going to be 'watching' for new buildings.

You can also create custom interfaces for class extensions whose clients are user applications. Such interfaces are created in the CASE tools in exactly the same way you create an interface for a custom feature.

Take a couple of minutes now to make sure your model is correct (check attribute names and types, for example). Use the model at the beginning of the tutorial as a guide. Save your UML diagram at this time.

UML modeling summary

You are finished with the modeling of the custom features. Notice the following in your diagram:

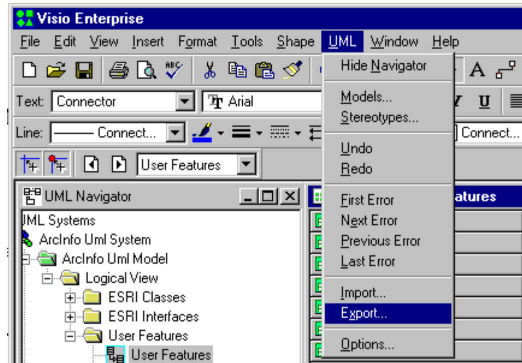
- New ArcInfo UML models are created based on a template.
- Custom features are defined using UML classes.
- User-defined features inherit from classes in the template.
- Feature datasets are represented as UML packages stereotyped as <<FeatureDataset>>.
- User-defined features should be created in the Workspace package or under a feature dataset package.
- Custom features implement interfaces. The relationship is expressed through a UML refinement.
- Interfaces are UML classes stereotyped as <<interface>>.
- Types in UML classes representing custom features are of the form esriFieldTypeXXX.
- Types in UML classes representing interfaces are restricted to COM/Automation types.
- The Schema Wizard uses attributes in a class to create fields in the feature class.
- The Code Generation Wizard uses attributes in interfaces to generate propput/propget pairs.
- Relationship classes require primary and foreign keys.
- Tagged values are used to fully specify model elements (for example, GeometryType, OriginPrimaryKey).

Exporting a model to the Repository

The Microsoft Repository supports the storage of a great deal of information related to the development of software including object models created using the UML. The CASE Tools take advantage of this feature to support any modeling tool that writes to the repository.

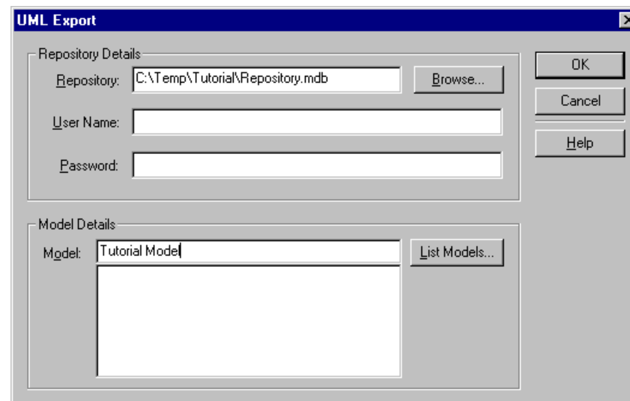
To export the model to a Repository

1. Click the UML menu, then click Export.



Exporting the UML model to the Repository in Visio makes available the model to ArcInfo.

2. Type the name of an existing or new access database or click browse to navigate to it (for example, C:\Temp\Tutorial\Repository.mdb).
3. Type the name of the model as it will be stored in the repository.
4. Click OK to export the UML model.



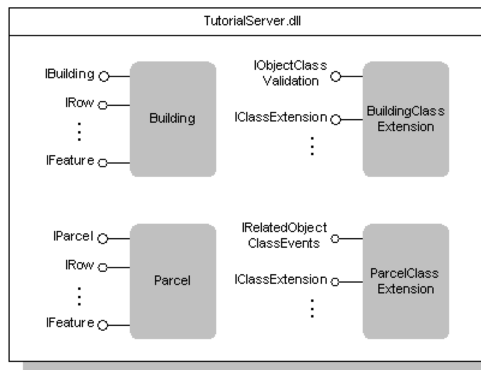
The export dialog in Visio.

The whole model, including the objects and interfaces in the ESRI packages, is exported to the Repository. Both code and schema will be created using that version of the model.

Generating code

You use the code generation wizard to create stub code for the custom features in your object models. The wizard is an add-in to Developer Studio that will create a C++ project for you. The code generated is based on the ATL, a framework designed by Microsoft to facilitate COM programming in C++.

Upon compilation of the project, a dynamic link library (DLL) with COM classes will be created. Custom features in the model selected for code generation will become COM classes in the DLL, along with a COM class for the associated class extension, if one exists.

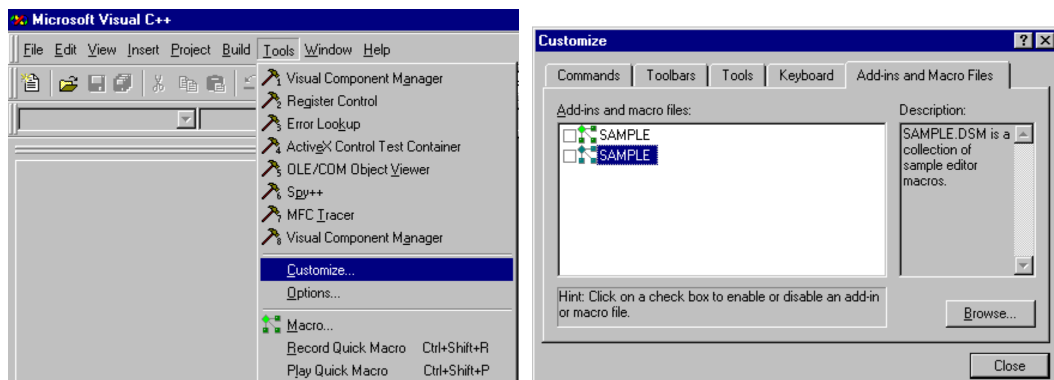


Custom features and class extensions will be COM classes inside a DLL.

Adding the code generation wizard to Developer Studio

To load the code generation wizard add-in

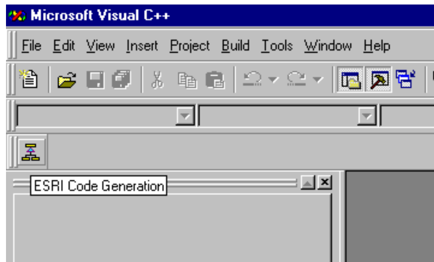
1. Start Developer Studio.
2. Click tools, then click customize.
3. Click the Add-ins and Macro Files tab.
4. Click browse to search for the add-in (the default installation directory is C:\arcgis\arcexe81\bin).



Adding the Code Generation Wizard to Developer Studio.

5. Click CodeGenWiz.dll (make sure the file type is *.dll).
6. Click Close.

The ESRI code generation add-in should be loaded now.



The Code Generation Wizard in Developer Studio.

Overview of the code generation

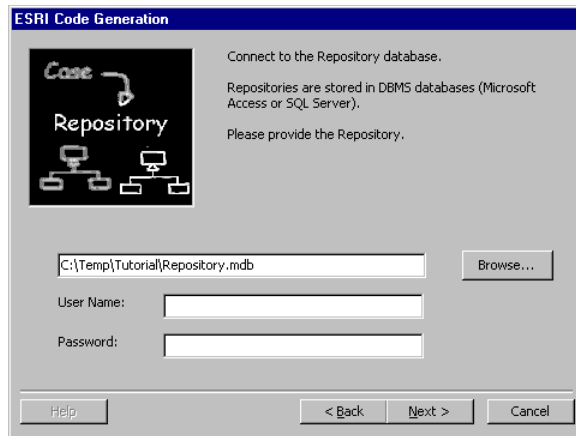
The wizard will guide you through the following steps:

- Connect to the Repository.
- Select the Object Model.
- Select the custom features to generate code for.
- Define properties for each custom feature.
- Specify the output C++ project.

Once finished, you will have a C++ project ready to compile where all custom features behave just like a standard geodata access feature. You can then modify each custom feature to include specific behavior.

Connecting to the repository

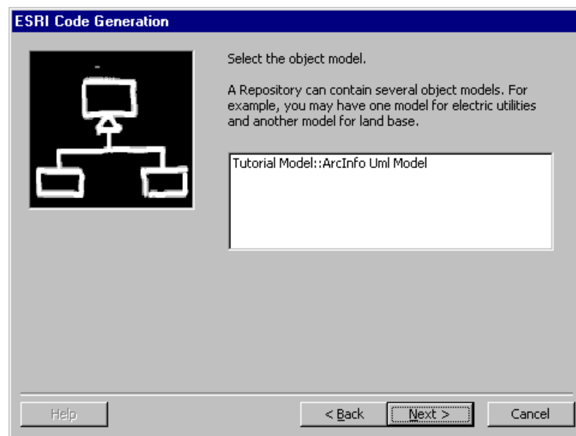
Launch the wizard in Developer Studio. Click Next to skip the introduction step and then click Browse to select the repository database (for example, C:\Temp\Tutorial\Repository.mdb). Click Next to continue.



Specifying the repository.

Selecting the object model

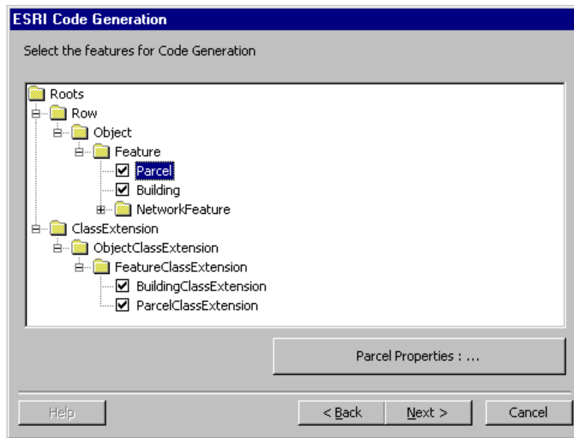
A repository database can contain several object models. Select the tutorial model and click Next. At this point the wizard reads the object model from the repository.



Selecting the object model.

Defining the custom features to create

The wizard will display your object hierarchy in a tree view. Here's where you'll select the custom features you want to create code for. For this example you'll generate code for all the custom features in the model.



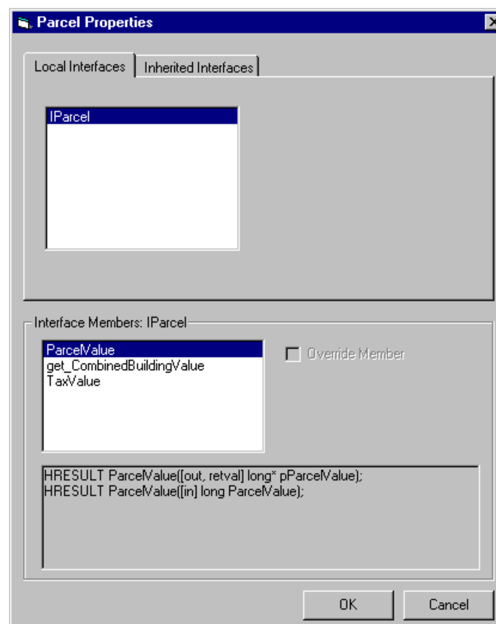
The Code Generation Wizard shows the object model hierarchy.

Defining custom feature properties

The behavior of a custom feature is the code behind the properties and methods of the interfaces it supports. The code generation properties of a custom feature allow you to define where the custom behavior will be placed.

1. Double-click Parcel to open its properties.

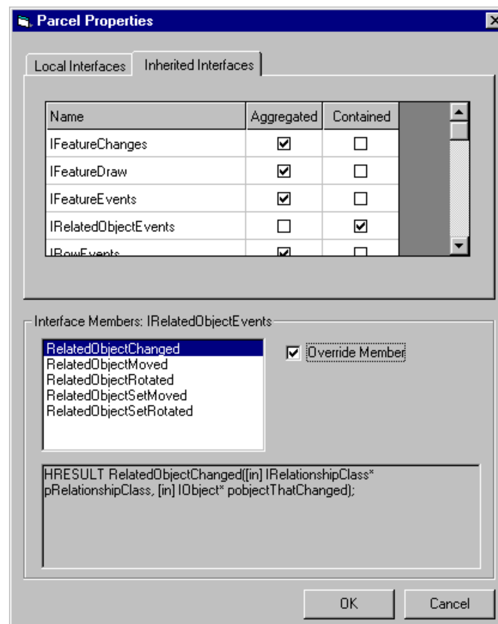
The Local Interfaces tab shows the definition of IParcel, the only local interface of parcel in the model. The lower portion provides the IDL definition of the interface members. All the information needed for code generation regarding local interfaces is defined in the model. This tab is designed to let you verify that information.



The local interfaces tab for the Parcel custom feature.

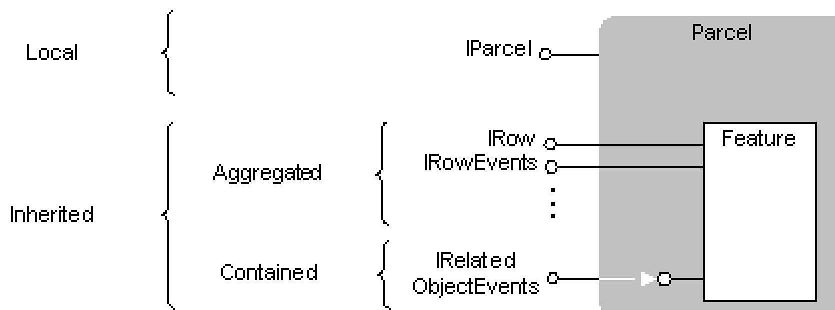
2. Click the Inherited Interfaces tab.

This tab displays the interfaces the parcel custom feature must implement because it inherits from feature. By clicking the name of the interface in the upper grid, the appropriate member information is displayed in the bottom part of the form.



The inherited interfaces tab for Parcel.

The code generation wizard will include the parent of Parcel (Feature), inside the Parcel COM class, in such a way that each time a new instance of Parcel is created a new instance of the inner feature will be created as well.



COM containment and aggregation can be used to reuse the implementation present in an existing COM class.

This technique allows the developer to reuse the implementation of all the interfaces Feature implements by either exposing the inner object's implementation directly (*aggregation*) or by forwarding calls to the inner instance (*containment*). The client of the COM class has no knowledge about this architecture, and to it all interfaces look as though the Parcel COM class has implemented them.

When an interface is contained, each property or method of the interface can either be forwarded to the inner object or entirely overwritten by the outer object. In the case of the parcel, the only interface we want to contain is IRelatedObjectEvents. In methods such as

RelatedObjectChanged we will write custom behavior. All other interfaces will be exposed directly from the inner feature COM class.

To contain the interface and override methods

1. Click the contained check box for IRelatedObjectEvents.
2. Click the RelatedObjectChanged method.
3. Click Override member check box.
4. Repeat steps 2 and 3 for the RelatedObjectSetMoved method.
5. Click OK.

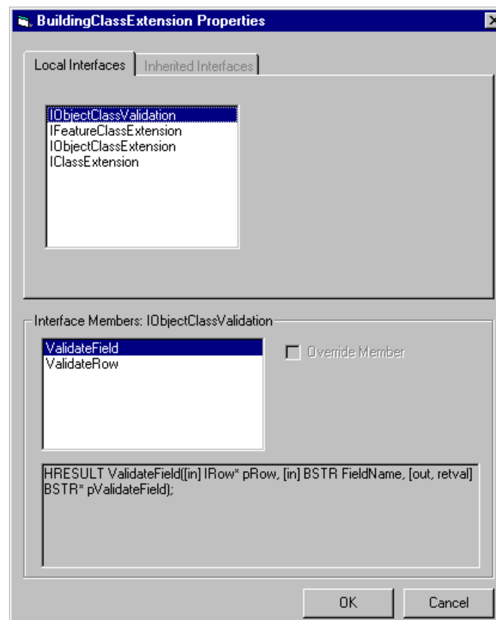
The same considerations apply for Building. However, this custom feature can reuse Feature's implementation of all the interfaces, so all the inherited interfaces will be aggregated.

Class extensions are a special case in one aspect: they inherit from abstract classes. Notice in your UML model that ClassExtension, ObjectClassExtension, and FeatureClassExtension are abstract classes (the UML convention is to present the class name in italics). In this case, there is no concrete COM class we can embed inside the user-defined class, that is, inside the class extensions for building and parcel.

1. Double-click the building class extension.

Notice all the interfaces will be implemented locally, so all of them are shown in the Local Interfaces tab. No interfaces are going to be aggregated or contained, so the Inherited Interfaces tab is disabled.

Notice also the optional interface IObjectClassValidation you added to the UML model is shown as a local interface. The method ValidateRow will be used to create the custom validation rule for buildings.



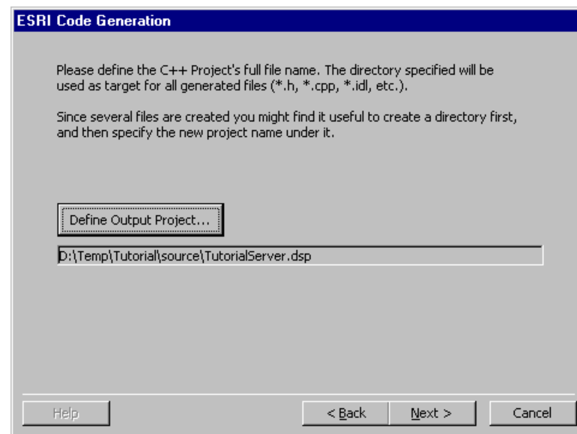
The local interfaces tab for the Building class extension.

2. Click OK to close the dialog.
3. Double-click ParcelClassExtension to open its properties.
4. Notice it will implement the optional interface IRelatedObjectClassEvents, just as you specified it in the UML model. The RelatedObjectCreated method will be used to handle the creation of a new building.
5. Click OK to close the dialog.
6. Click Next to continue with the code generation process.

Defining the output Developer Studio project

As the last step of the code generation process, the wizard prompts you to define the output Developer Studio/C++ project. Several files will be created along the project file, so you are better off using a separate folder.

1. Click Define Output Project.



Defining the output C++ project.

2. Move to the output directory (for example C:\Temp\Tutorial\source).
3. Type TutorialServer in the File Name box.
4. Click Save.
5. Click Next.
6. Click Finish.

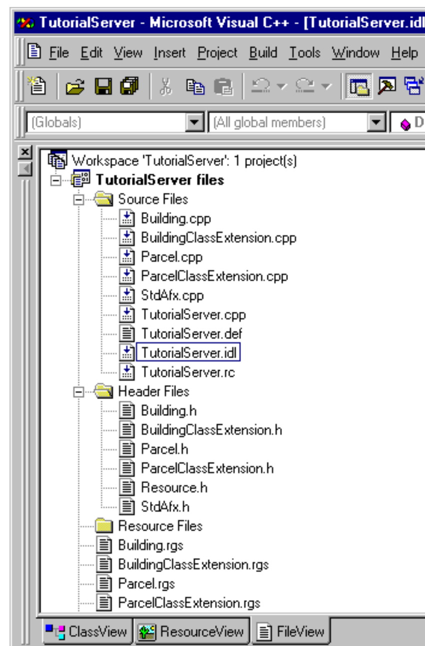
At this point the wizard will generate a Developer Studio project and workspace you can open. The compilation of the project will generate a COM DLL that contains COM classes for the classes in the model: Parcel, Parcel class extension, Building, and Building class extension. The name of the DLL will be TutorialServer, the name of the C++ project.

Adding custom behavior

The following pages will help you add C++ code to implement the behavior for parcels, buildings, and their class extensions. Comments and lengthy lines may appear differently in the book than in the files on disk due to restrictions imposed by the book format.

Generated code

Let's start by browsing through the code just generated. In Developer Studio click the FileView tab and expand the tree so all files are visible. Here is a small description of them (for further details on the project outline, refer to the ATL documentation or any of the books available on ATL).



The code generated is a Developer Studio C++ workspace and project.

A number of support files are created such as TutorialServer.cpp, TutorialServer.rc, and TutorialServer.def. These files provide services such as registration and unregistration of the COM classes in your DLL.

The file StdAfx.h contains a *#import* statement that incorporates information from the esriCore type library. The content of the type library is converted into C++ classes that ease the use of ESRI COM classes and interfaces when writing the behavior.

It is through these wrapper classes that we use the COM classes of the geodata access components. For example, we can write the following code to create an instance of a field object:

```
IFieldPtr ipField(CLSID_Field);        // IFieldPtr wrapper class
```

Another interesting file in your project is TutorialServer.idl. This file has metadata of the contents of your project. Notice the interfaces are defined, and then, inside a type library, all

the COM classes are defined too, along with the interfaces they implement. Notice also the custom features all the interfaces implemented by feature, although you can't tell the difference between aggregated and contained interfaces. This is how a client will see your custom features.

For each COM class a registration file is created, for example, Parcel.rgs. This script creates the registry keys and values so the COM class is correctly registered in the system's registry. It includes registering the parcel COM class in a component category: the custom features' component category. Class extensions COM classes (for example, the parcel class extension) are registered in the class extensions component category in the same way.

You also get the header and source files for the classes in the UML model, for example, the header file Parcel.h and the source file Parcel.cpp. In the header file you can recognize a number of ATL macros being used, among them the ones defining the COM_MAP. It is apparent the parcel COM class implements IParcel and IRelatedObjectEvents locally and aggregates any other interface implemented by the inner object (Feature).

```
BEGIN_COM_MAP(Parcel)
   COM_INTERFACE_ENTRY(IParcel)
   COM_INTERFACE_ENTRY(IRelatedObjectEvents)
   COM_INTERFACE_ENTRY_AGGREGATE_BLIND(m_pInnerUnk)
END_COM_MAP()
```

Open the source code for the parcel (Parcel.cpp). The method *FinalConstruct* is called by ATL as the final step of the COM class creation procedure. An instance of the feature COM class will be created in this method. Parcel holds a reference to the inner feature through the member variable m_pInnerUnk, a pointer to Feature's IUnknown interface.

Also, a query interface is done to get a pointer to the only contained interface, IRelatedObjectEvents.

```
HRESULT Parcel::FinalConstruct()
{
   IUnknown * pOuter = GetControllingUnknown();

   if (FAILED (CoCreateInstance(__uuidof(Feature),  /* create inner feature  */
            pOuter,
            CLSCTX_INPROC_SERVER,
            IID_IUnknown,
            (void**) &m_pInnerUnk)))                    /* hold it */
      return E_FAIL;

   // QI for IRelatedObjectEvents
   if (FAILED(m_pInnerUnk->QueryInterface(IID_IRelatedObjectEvents,
                                          (void**)&m_pIRelatedObjectEvents)))
      return E_FAIL;
   pOuter->Release();

   return S_OK;
}
```

Then stub code is created for methods of the local interface IParcel. All of them return E_NOTIMPL, as the developer has to provide the implementation. An example is the accessor for ParcelValue:

```
STDMETHODIMP Parcel::get_ParcelValue(long* pParcelValue)
{
   return E_NOTIMPL;
}
```

And lastly, methods of the only contained interface, IRelatedObjectEvents, are created. An example is:

```
STDMETHODIMP Parcel::RelatedObjectChanged(IRelationshipClass* pRelationshipClass,
    IObject* pobjectThatChanged)
{
        return E_NOTIMPL;
}
```

Notice the methods you decided to override return E_NOTIMPL, while the others are just forwarding the call to the same interface implemented by the inner COM class.

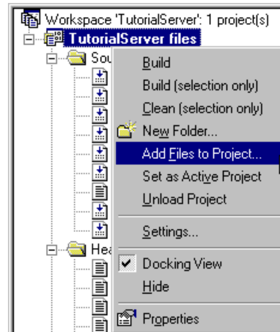
It is in the implementation of these local and contained interfaces where the behavior of custom features is placed.

Although it might look a little intimidating at this point, the good news is all that stuff is already there! The CASE tools allow you to design your objects using a graphical language and then generate a lot of boilerplate code that takes care of a lot of details needed for the creation of custom features using C++. You can (almost) forget entirely about those details and concentrate on what you really want: writing the behavior of your custom features.

Inserting helper files

A couple of files containing helper functions were already in the source directory. To add the files to the project:

1. Right-click the TutorialServer project, then click Add Files to Project.
2. Select the file databaseTools.h.
3. Repeat steps 1 and 2 to add the source file databaseTools.cpp.



Adding utility files to the C++ project.

4. Double-click Building.cpp and add #include “databaseTools.h” after the last #include in the file.

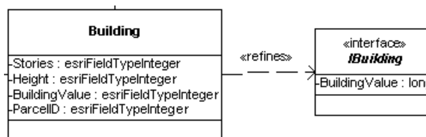
```
// Building.cpp : Implementation of Building
// Generated 9/22/99 11:28:15 AM
//
#include "stdafx.h"
#include "TutorialServer.h"
#include "Building.h"
#include "databaseTools.h"
…
```

5. Repeat step 4 for Parcel.cpp, ParcelClassExtension.cpp, and BuildingClassExtension.cpp.

Double-click databaseTools.cpp to open the source code. Two of the functions in databaseTools.cpp help you read and write values to and from fields in feature classes (GetFieldValue and PutFieldValue). The third function tells if a relationship class has a specific name (IsRelationshipClass).

Adding behavior for building

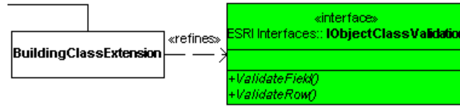
We will start implementing the functionality of the Building feature. Open Building.cpp at this point. Only two methods require implementation, the accessor and mutator for the FieldValue property. This is the implementation of the methods:



The Building COM class will implement the IBuilding interface.

```
STDMETHODIMP Building::get_BuildingValue(long* pBuildingValue)
{
   HRESULT        hr;
   CComBSTR    fieldName = L"BuildingValue";
   CComVariant value;

   IRowPtr    ipRow(GetControllingUnknown());
   if (ipRow == 0)
      return E_FAIL;

   if (FAILED(hr = GetFieldValue(fieldName, ipRow, &value)))
      return hr;

   *pBuildingValue = value.lVal;

   return hr;
}


STDMETHODIMP Building::put_BuildingValue(long BuildingValue)
{
   CComBSTR      fieldName = L"BuildingValue";
   CComVariant value(BuildingValue);

   IRowPtr    ipRow(GetControllingUnknown());
   if (ipRow == 0)
      return E_FAIL;

   return PutFieldValue(fieldName, ipRow, value);
}
```

That completes the code for the building COM class. Now we also want to create a custom validation rule that verifies the height is at least the number of floors times 10 feet. The code for the validation rule will be implemented inside the class extension for the building. Open BuildingClassExtension.cpp and insert code for the ValidateRow method:



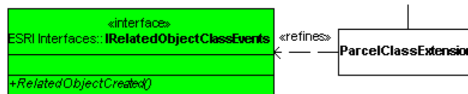
The BuildingClassExtension will implement the methods in the optional interface IObjectClassValidation.

```
STDMETHODIMP BuildingClassExtension::ValidateRow(IRow* pRow, BSTR* pValidateRow)
{
   CComVariant height, stories;
   CComBSTR      bstrFieldName(L"Height");

   if (FAILED(GetFieldValue(bstrFieldName, pRow, &height)))
      return E_FAIL;

   bstrFieldName = L"Stories";

   if (FAILED(GetFieldValue(bstrFieldName, pRow, &stories)))
      return E_FAIL;

   CComBSTR bstrErrMsg;
   bstrErrMsg = L"The number of stories x 10ft exceeds the building height";

   if (height.lVal < (stories.lVal * 10))
      *pValidateRow = bstrErrMsg.Copy();

   return S_OK;
}
```

When the user triggers the validation for a building, this code will be executed and a message will be shown for those buildings that are invalid.

Adding behavior for parcel

Now let's go to the parcel. First, look at the code that keeps an eye on new buildings and creates relationships between the new building and the parcel containing it. This behavior is implemented in the Parcel class extension, which receives a notification each time a new related object is created. Open ParcelClassExtension.cpp and insert the following code as the implementation of the RelatedObjectCreated method.



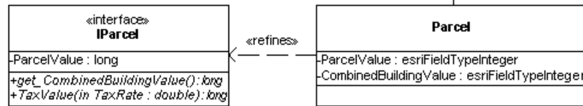
The ParcelClassExtension will implement the method RelatedObjectCreated, in the optional interface IRelatedObjectClassEvents.

```
STDMETHODIMP ParcelClassExtension::RelatedObjectCreated(
                                    IRelationshipClass* pRelationshipClass,
                                    IObject* pobjectThatWasCreated)
{
   // first verify the relationship is "Contains"
   if (!IsRelationshipClass(pRelationshipClass,CComBSTR(L"Contains")))
      return S_OK;

   // the relationship is "Contains", thus the created object must be a building.

   IGeometryPtr ipBuildingGeometry;

   ((IFeaturePtr) pobjectThatWasCreated)->get_Shape(&ipBuildingGeometry);
```

```
    if (ipBuildingGeometry == 0)
        return E_FAIL;

    // Search for any parcel whose geometry contains that
    // of the newly created building.

    IFeatureClassPtr          ipParcelFeatureClass(m_pClass);
    IFeatureDatasetPtr     ipFeatureDataset;
    ISpatialReferencePtr ipSpatialReference;

    ipParcelFeatureClass->get_FeatureDataset(&ipFeatureDataset);
    ((IGeoDatasetPtr) ipFeatureDataset)->get_SpatialReference(&ipSpatialReference);

    // Prepare the spatial filter

    ISpatialFilterPtr ipSpatialFilter(CLSID_SpatialFilter);

    ipSpatialFilter->putref_Geometry(ipBuildingGeometry);
    ipSpatialFilter->put_SpatialRel(esriSpatialRelIntersects);
    CComBSTR shapeField(L"Shape");

    ipSpatialFilter->put_GeometryField(shapeField);
    ipSpatialFilter->putref_OutputSpatialReference(shapeField, ipSpatialReference);

    // Do the query

    HRESULT hr;
    CComQIPtr<IParcel> ipParcel;
    IFeatureCursorPtr        ipParcels;

    if (FAILED(hr = ipParcelFeatureClass->Search(ipSpatialFilter, VARIANT_FALSE, &ipParcels)))
        return hr;

    // check a parcel was found

    IFeaturePtr ipFeature;

    if (ipParcels->NextFeature(&ipFeature) != S_OK)
        return S_OK;

    ipParcel = ipFeature;

    // Create an instance of the relationship.

    IObjectClassPtr  ipBuildingObjectClass;
    IRelationshipPtr  ipRelationship;

    pobjectThatWasCreated->get_Class(&ipBuildingObjectClass);

    return pRelationshipClass->CreateRelationship((IObjectPtr) ipParcel,
                                                pobjectThatWasCreated,
                                                &ipRelationship);
}
```

The code first verifies the relationship class is “Contains” because parcels could have relationships with other feature classes. Then it grabs the geometry of the object that was created (building) and searches for the parcel underneath it. If a parcel is found, a new contains relationship is created between the new building and the parcel.

Now let’s go to the parcel object itself. Open Parcel.cpp at this point. We have two interfaces to implement: IParcel and IRelatedObjectEvents (the latter because we decided to contain it during code generation). The code behind the ParcelValue property and the read-only property CombinedBuildingValue is pretty much the same as the one used for the building value property.



The Parcel feature will implement the IParcel interface.

```
STDMETHODIMP Parcel::get_ParcelValue(long* pParcelValue)
{
  HRESULT     hr;
  CComBSTR    fieldName = L"ParcelValue";
  CComVariant value;

  IRowPtr     ipRow(GetControllingUnknown());
  if (ipRow == 0)
    return E_FAIL;

  if (FAILED(hr = GetFieldValue(fieldName, ipRow, &value)))
    return hr;

  *pParcelValue = value.lVal;

  return hr;
}

STDMETHODIMP Parcel::put_ParcelValue(long ParcelValue)
{
  CComBSTR      fieldName = L"ParcelValue";
  CComVariant value(ParcelValue);

  IRowPtr     ipRow(GetControllingUnknown());
  if (ipRow == 0)
    return E_FAIL;

  return PutFieldValue(fieldName, ipRow, value);
}

STDMETHODIMP Parcel::get_CombinedBuildingValue(long* pCombinedBuildingValue)
{
  HRESULT hr;
  CComBSTR      fieldName = "CombinedBuildingValue";
  CComVariant value;

  IRowPtr     ipRow(GetControllingUnknown());
  if (ipRow == 0)
    return E_FAIL;

  if (FAILED(hr = GetFieldValue(fieldName, ipRow, &value)))
    return hr;

  *pCombinedBuildingValue = value.lVal;

  return hr;
}
```

The code for the last method of IParcel calculates the tax value of a parcel, based on a given tax rate, the parcel's value, and the combined value of the buildings in the parcel.

```
STDMETHODIMP Parcel::TaxValue(double TaxRate, long* pTaxValue)
{
  // Dynamically calculate the tax value

  if (TaxRate < 0)
    return E_FAIL;

  long parcelValue, buildingValue;
```

```
    get_ParcelValue(&parcelValue);
    get_CombinedBuildingValue(&buildingValue);

    *pTaxValue = static_cast<long>((buildingValue + parcelValue) * TaxRate);

    return S_OK;
}
```

We want to keep the parcel's field CombinedBuildingValue updated if the value of a building changes or if a building is moved outside of a parcel. To implement this functionality, we will first create a couple of helper functions. The first function calculates the combined value of the buildings in a parcel. The second verifies a building is still contained in a parcel.

Open Parcel.h and insert the following prototypes in the private section of the class:

```
private:
    // helper functions
    HRESULT RecalculateCombinedBuildingValue(IRelationshipClass* pRelClass);
    HRESULT CheckBuildingContainment(IRelationshipClass* pRelClass, IObject* pBuildingObject);
```

Now open Parcel.cpp. The following routine maintains the combined building value updated. It loops through the related buildings calculating the total value. Then it writes the value to the database. Insert the code in your file.

```
HRESULT Parcel::RecalculateCombinedBuildingValue(IRelationshipClass* pRelClass)
{
    IRowPtr ipRow(GetControllingUnknown());
    if (ipRow == 0)
        return E_FAIL;

    HRESULT    hr;
    ISetPtr       ipRelatedBuildings;

    if (FAILED(hr = pRelClass->GetObjectsRelatedToObject((IObjectPtr) ipRow, &ipRelatedBuildings)))
        return hr;

    IUnknownPtr                 ipUnk;
    IRowPtr                         ipBuildingRow;
    CComQIPtr<IBuilding> ipBuilding;
    long                               combinedBuildingValue = 0;

    while (ipRelatedBuildings->Next(&ipUnk) == S_OK)
    {
        long value;

        ipBuildingRow = ipUnk;

        ipBuilding = ipBuildingRow;
        ipBuilding->get_BuildingValue(&value);

                combinedBuildingValue += value;
    }

    // Persist the new combined building value.

    CComBSTR      fieldName = "CombinedBuildingValue";
    CComVariant value(combinedBuildingValue);

    if (FAILED(hr = PutFieldValue(fieldName, ipRow, value)))
        return hr;

    return ipRow->Store();
}
```

The second routine breaks the relationship between a building and a parcel if the building is moved and is no longer contained in the parcel. Notice that besides removing the relationship it also recalculates the combined building value.

```
HRESULT Parcel::CheckBuildingContainment(IRelationshipClass* pRelClass, IObject* pBuildingObject)
{
   // If the building is no longer contained within the parcel,
   // break the relationship and recalculate the combined building value.

   IFeaturePtr ipFeature(GetControllingUnknown());
   if (ipFeature == 0)
      return E_FAIL;

   IGeometryPtr ipBuildingGeometry, ipParcelGeometry;

   ((IFeaturePtr) pBuildingObject)->get_Shape(&ipBuildingGeometry);
   ipFeature->get_Shape(&ipParcelGeometry);

   VARIANT_BOOL contains;

   ((IRelationalOperatorPtr) ipParcelGeometry)->Contains(ipBuildingGeometry, &contains);

   if (contains == VARIANT_TRUE)
      return S_OK;

   // INVARIANT: The parcel no longer contains the building.

   HRESULT hr;

   if (FAILED(hr = pRelClass->DeleteRelationship((IObjectPtr) ipFeature, pBuildingObject)))
      return hr;

   return RecalculateCombinedBuildingValue(pRelClass);
}
```

Now let's work on the actual events received by the parcel through IRelatedObjectEvents. The first event is triggered when a single related object is changed. Either the shape or the value of one of its properties could have changed.

If the shape has changed, we test if the building is still contained within the parcel. If it is not contained we break the relationship and calculate the combined building value.

If the shape did not change, then perhaps the value of a building might have. In this case, we calculate the combined building value of the parcel.

Insert the following code as the implementation of the method RelatedObjectChanged.

```
STDMETHODIMP Parcel::RelatedObjectChanged(IRelationshipClass* pRelationshipClass,
                                  IObject* pobjectThatChanged)
{
   if (!pRelationshipClass || !pobjectThatChanged)
      return E_POINTER;

   if (!IsRelationshipClass(pRelationshipClass, CComBSTR(L"Contains")))
      return S_OK;

   HRESULT            hr;
   VARIANT_BOOL  shapeChanged;
   IFeaturePtr         ipFeature(pobjectThatChanged);

   if (ipFeature == 0)
      return S_OK;

   IFeatureChangesPtr ipFeatureChanges(ipFeature);
```

```
   ipFeatureChanges->get_ShapeChanged(&shapeChanged);

   if (shapeChanged)
   {
      if (FAILED(hr = CheckBuildingContainment(pRelationshipClass, pobjectThatChanged)))
         return hr;
   }
   else
   {
      if (FAILED(hr = RecalculateCombinedBuildingValue(pRelationshipClass)))
         return hr;
   }

   return m_pIRelatedObjectEvents->RelatedObjectChanged(pRelationshipClass,
                                            pobjectThatChanged);
}
```

The second event we will handle is triggered when a set of buildings is moved. This case is a bit more complex because the geodata access components send this message only once to a parcel in the set of parcels whose buildings were moved. The parcel must handle the movement of its own buildings, remove itself from the set of affected parcels, and send the message to the next parcel in the set. Eventually, all parcels will handle the movement of their buildings.

In this implementation the parcel doesn't really remove itself from the affected parcels set. Instead, it creates a clone of the set and doesn't include itself in it. It is not fair to assume the routine can change the set passed in because the caller routine may use the set later on and would be really surprised to find an empty set.

Insert the following code as the implementation of the method RelatedObjectSetMoved.

```
STDMETHODIMP Parcel::RelatedObjectSetMoved(IRelationshipClass* pRelationshipClass,
                                 ISet*  pobjectsThatNeedToChange,
                                 ISet*  pobjectsThatChanged,
                                 ILine* pMoveVector)
{
   // check it is the Contains relationship
   if (!IsRelationshipClass(pRelationshipClass,CComBSTR(L"Contains")))
      return S_OK;

   IObjectPtr ipObject(GetControllingUnknown());
   if (ipObject == 0)
      return E_FAIL;

   // find the buildings related to the parcel

   ISetPtr ipRelatedBuildingObjects;

   pRelationshipClass->GetObjectsRelatedToObject(ipObject, &ipRelatedBuildingObjects);

   pobjectsThatChanged->Reset();

   HRESULT       hr;
   ISetPtr          ipPrunedRelatedSet(CLSID_Set);
   IRowPtr        ipRelatedBuildingRow;
   IUnknownPtr ipUnknown, ipUnkRelatedBuilding;
   IObjectPtr     ipRelatedBuildingObject;

   // match buildings that changed to those really related to this parcel

   while (pobjectsThatChanged->Next(&ipUnknown) == S_OK)
   {
      bool matchFound = false;

      while (!matchFound && ipRelatedBuildingObjects->Next(&ipUnkRelatedBuilding) == S_OK)
      {
```

```
            if (ipUnknown == ipUnkRelatedBuilding)
            {
               // building that changed found in the related buildings
               // stop looking for it in the related buildings
               matchFound = true;

               // check changed building is still contained by parcel
               ipRelatedBuildingObject = ipUnknown;
               if (FAILED(hr = CheckBuildingContainment(pRelationshipClass, ipRelatedBuildingObject)))
                  return hr;
            }
         }
      }


   // Clone the parcel set, removing self.

   bool              remainingParcels = false;
   ISetPtr           ipPrunedParcelSet(CLSID_Set);
   IUnknownPtr  ipUnkParcel, ipUnkSelf(ipObject);

   IRelatedObjectEventsPtr ipNextParcel;

   pobjectsThatNeedToChange->Reset();

   while (pobjectsThatNeedToChange->Next(&ipUnkParcel) == S_OK)
   {
      if (ipUnkParcel != ipUnkSelf)
      {
         remainingParcels = true;
         ipNextParcel = ipUnkParcel;
         ipPrunedParcelSet->Add(ipUnkParcel);
      }
   }

   if (!remainingParcels)
      return S_OK;

   // Recursively call this method on one item in the cloned parcel set.
   // If there are no remaining parcels, simply return.

   return ipNextParcel->RelatedObjectSetMoved(pRelationshipClass,
                                       ipPrunedParcelSet,
                                       ipPrunedRelatedSet,
                                       pMoveVector);
}
```

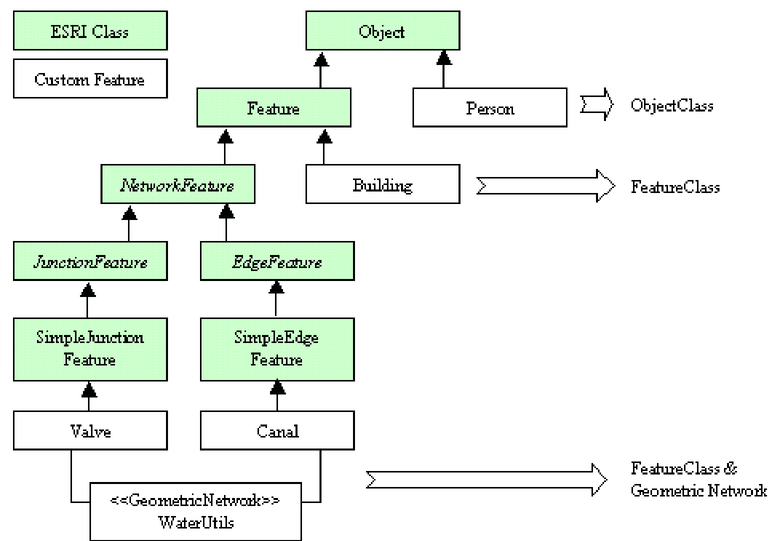
We will leave other methods in the IRelatedObjectEvents untouched. They will be forwarded to the inner feature. However, some functionality is still to be developed for a fully functional model. For example, when a building is moved from one parcel to a different parcel, the relationship with the original parcel is broken, but the relationship with the new parcel is never created. As always, there is room for more development.

In Developer Studio, click the Build menu and then click Build TutorialServer.dll to compile the project. The library TutorialServer.dll will be created and registered in your system upon compilation.

Creating the schema

The second wizard of the CASE tools will help you create a schema for your UML models in a geodatabase. For each UML class in the model, a table will be created. For each UML attribute in the class, a field in the table will be created.

Depending on the parent class, an object class or a feature class is created. Object classes will be created for custom features that inherit from the Object class. Feature classes will be created for custom features inheriting from any other concrete class in the ESRI Classes Package. Feature classes can be further divided into simple feature classes—those inheriting from Feature, and network feature classes —those inheriting from junction or edge features. In the latter case a geometric network will be created as well.

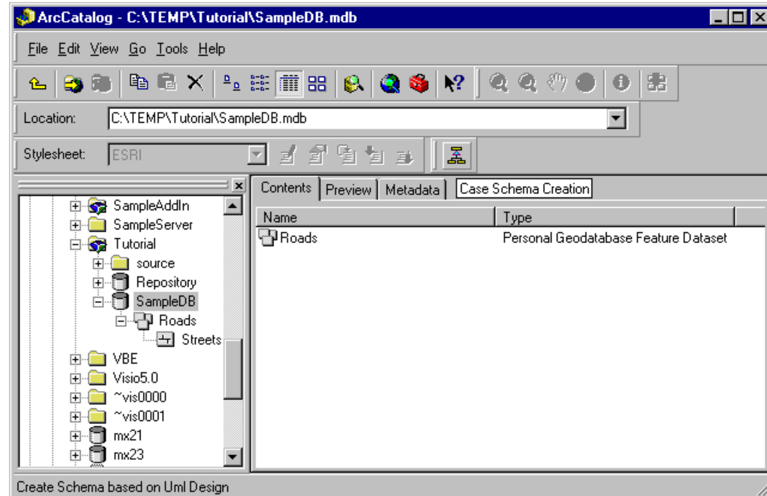


UML object models can be used to create geodatabase schemas. The feature type of the feature classes is determined based on the ESRI class they inherit from.

The wizard is a command in ArcCatalog. You can launch the wizard when a geodatabase has been selected on the left-hand side panel of ArcCatalog. This implies the feature datasets, feature classes, and other model elements will be created within the target geodatabase.

To add the wizard command to ArcCatalog

1. Start ArcCatalog.
2. Click Tools, then click Customize.
3. Click the Commands tab.
4. Click CASE Tools in the categories list.
5. Drag the Schema Wizard command onto a toolbar.
6. Click Close.



The Schema Wizard command in ArcCatalog.

Overview of the schema creation

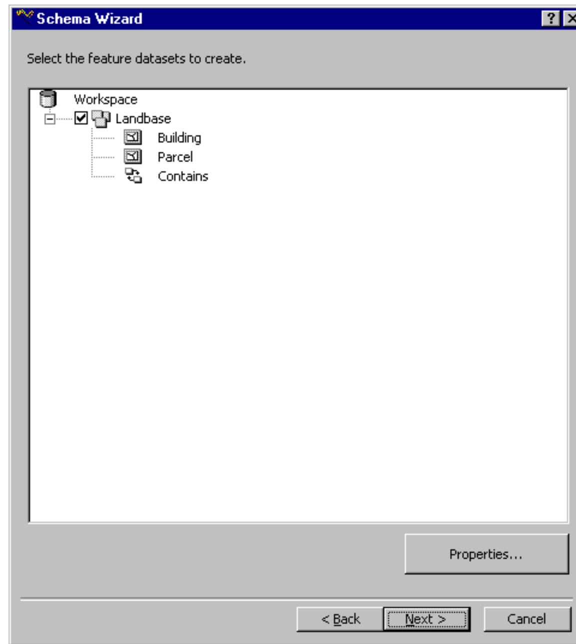
You will create a schema for the tutorial model. Under a selected geodatabase, the wizard will create a feature dataset (Landbase), two feature classes (building and parcel), and a composite relationship class (contains). The wizard will guide you through the following steps:

- Connect to the Repository.
- Select the object model.
- Define schema properties for each feature class.
- Create the schema.

To start the wizard, select the geodatabase you want to use as a target for the creation of the schema (for example, C:\Tutorial\SampleDB).

Creating the schema

The first three steps are exactly the same as those of the Code Generation Wizard, described previously. Start the wizard, connect to the repository, and select the tutorial model now.



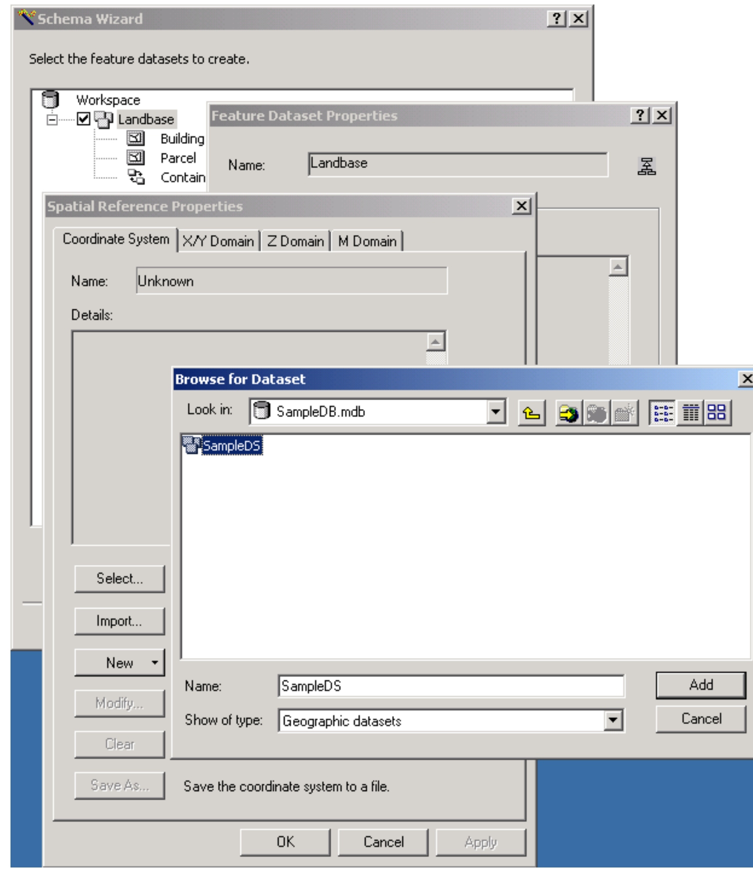
The Schema Wizard interprets UML elements as geodatabase elements, such as feature datasets, feature classes, and relationship classes.

The hierarchy of the model shown by the schema wizard is similar to that shown by ArcCatalog. You can identify the workspace (the geodatabase), the land base feature dataset, the feature classes, and the relationship class.

For each element you can open a properties dialog. Select the Landbase feature dataset, then click Properties. The feature dataset's only property is its spatial reference. To set it:

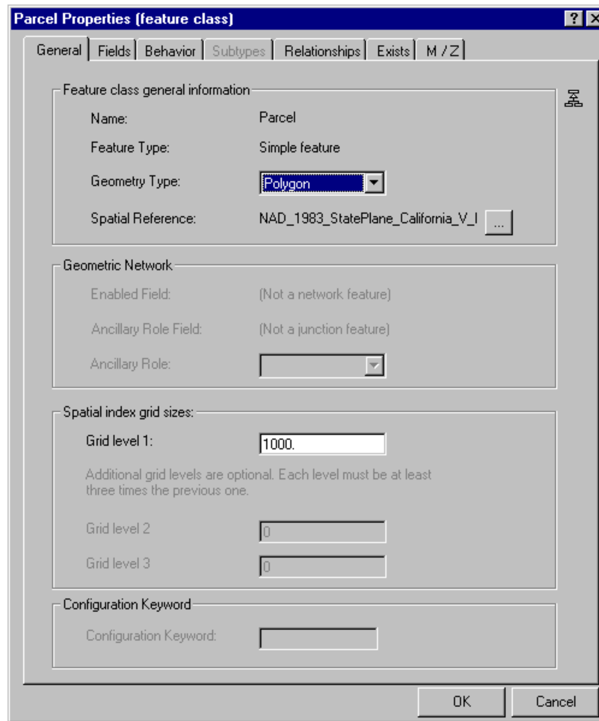
1. In the feature dataset properties dialog, click Edit.
2. In the spatial reference properties dialog, click Import.
3. Browse to the existing feature dataset under SampleDB (C:\Temp\Tutorial\SampleDB\SampleDS).
4. Click Add.
5. Click OK to accept the changes to the spatial reference.
6. Click OK to accept the changes to the feature dataset in the model.

The Landbase feature dataset will have the same spatial reference as the existing Roads feature dataset.



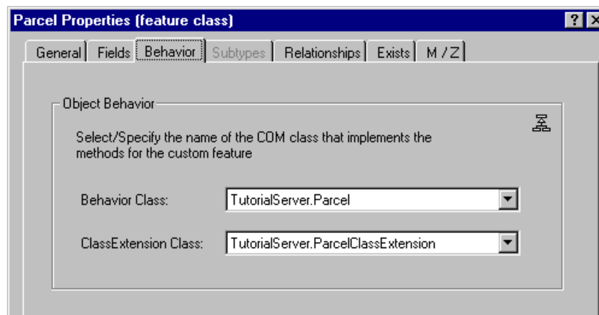
Setting then spatial reference for a feature dataset while running the Schema Wizard.

Double-click Parcel to open its properties. This dialog shows the properties of the feature class that will be created including geometry type, fields, relationships, and behavior COM classes. Notice Polygon is selected in the geometry type dropdown menu because it was specified as a tagged value of the corresponding parcel UML class in the model. Change the grid size to be 1.



Properties of a feature class in the Schema Wizard.

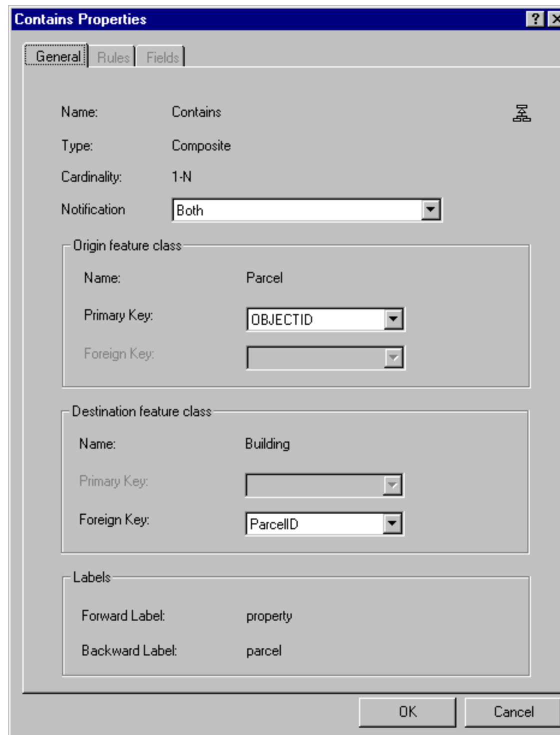
Click the Behavior tab and notice the parcel custom feature and the associated class extension have been selected automatically. The schema wizard looks for the Behavior COM classes in the system's registry, and therefore it is necessary to register the dynamic link library beforehand (compiling in Developer Studio registered the TutorialServer.dll). Click OK to dismiss the dialog.



Behavior classes that will be assigned to the feature class upon creation.

Use the same procedure to verify the definition of the building feature class. Change the grid size to 1 and check the geometry type and behavior classes have been correctly read from the model.

Double-click contains to open the properties dialog. Notice all the information is read from the association in the UML model including notification and primary and foreign keys. Click OK to dismiss the dialog and then click Next to continue with the wizard.

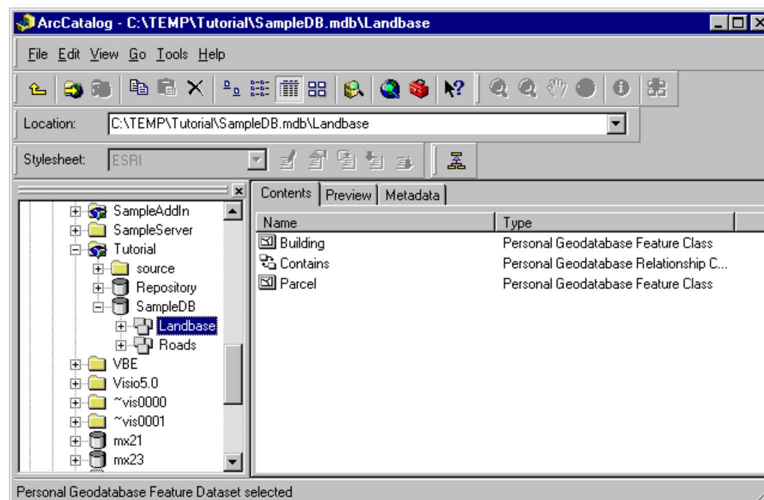


Properties of a relationship class in the Schema Wizard.

Before creating the schema, the wizard will display a summary of the custom features and options you selected. Click Finish to create the schema.

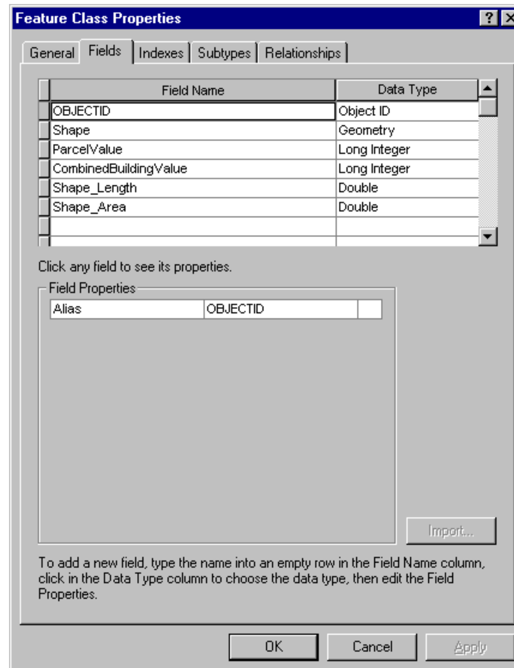
Verifying the schema

In ArcCatalog, click the Landbase feature dataset. Notice the two polygon feature classes have been added along with the Contains relationship class.



The Schema Wizard has created the feature dataset, two feature classes, and a relationship class.

Double-click the parcel feature class to open the properties dialog. Click the Fields tab and notice the fields correspond to those defined in the UML model for the parcel class. The shape_length and shape_area fields are automatically added and maintained by the geodatabase for any polygon feature class. Click OK to close the dialog.



Fields of the Parcel feature class in ArcCatalog.

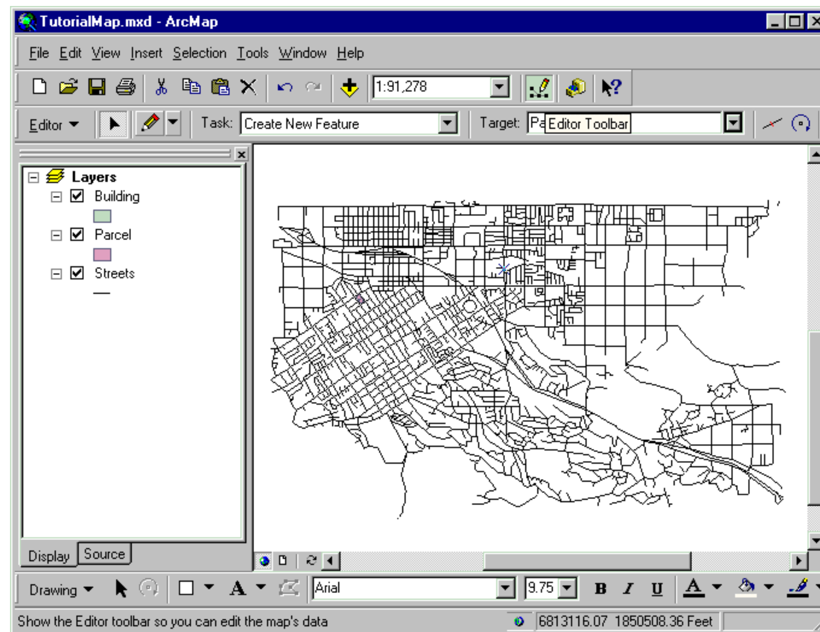
Double-click the Contains relationship class and verify the properties, as seen by ArcCatalog. When done, click OK to dismiss the dialog.

Properties of the Contains relationship class in ArcCatalog.

Using the custom features in ArcMap

In this step of the tutorial, you will create a few parcels and buildings to test the schema and the functionality implemented in the behavior COM classes.

Start ArcMap and drag and drop both feature datasets in SampleDB from ArcCatalog onto ArcMap. Make sure the editor toolbar is available. Save the map as TutorialMap under the tutorial directory.

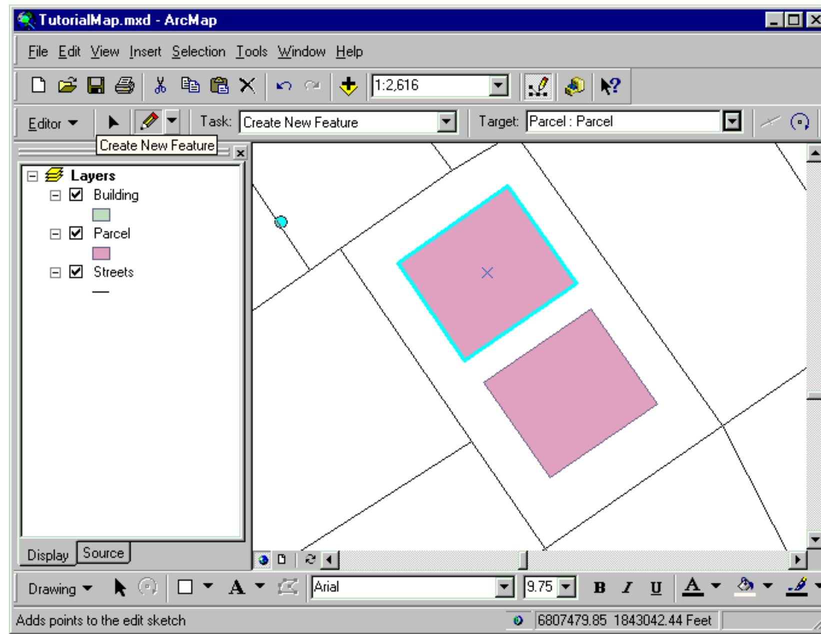


Preparing ArcMap for editing features in the created schema.

Testing the custom feature behavior

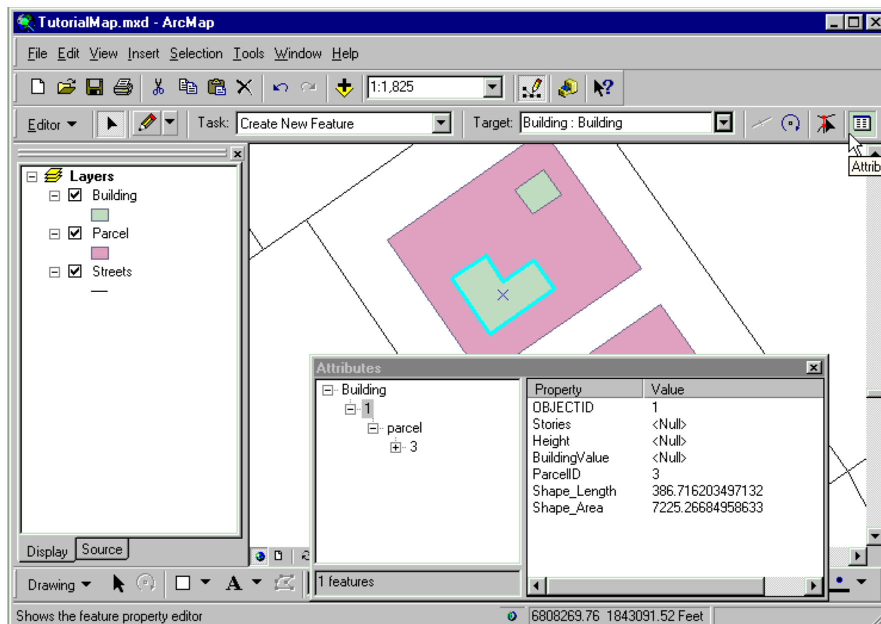
You will create a couple of parcels and buildings in this step. Zoom in to a single block in the map and create the parcels:

1. In the editor toolbar click Editor, then click StartEditing.
2. Verify the current Task is Create New Feature, click the Target dropdown, and click Parcel.
3. Click the Create New Feature tool and use the mouse to digitize two parcels.



Adding parcel features.

Now you'll create the buildings. Click the Target dropdown arrow and click Building. Digitize two buildings inside a parcel. With a building selected, click the Attributes tool to open the Property Inspector. Click the plus sign by the building ID and notice it has a related parcel. The relationship was created by the Parcel class extension in response to the creation of a related object, the building.



Adding buildings and setting attributes.

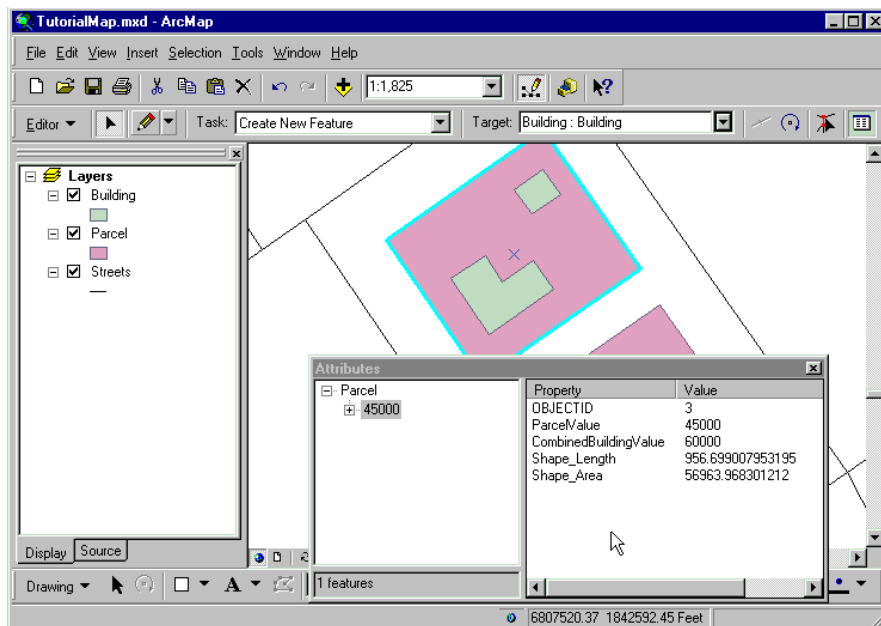
In the Property Inspector click the building ID again and fill in values for the fields as indicated in the following table:

Field	Value
Stories	1
Height	12
BuildingValue	50,000

Select the second building in the parcel and specify the following attributes:

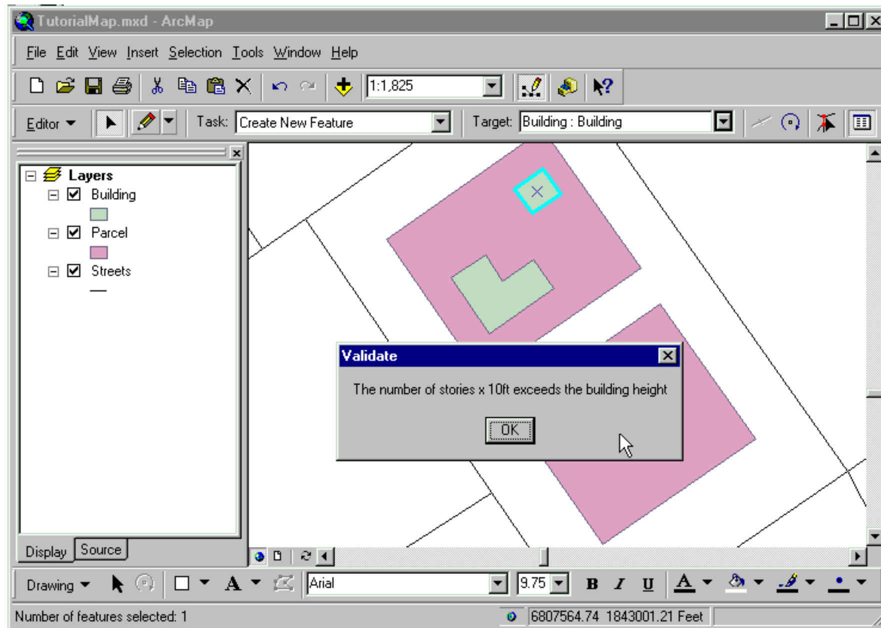
Field	Value
Stories	1
Height	8
BuildingValue	10,000

The parcel custom feature recalculates the combined building value each time a related building field value changes. In the map, select the parcel to display its attributes. Notice how the current combined building value is 60,000, the sum of the value of the two buildings. The parcel itself does not have a value yet, so change the parcel value to 45,000.



Parcel attributes, showing the combined building value calculated by the Parcel custom feature.

Close the Attributes dialog and select the two buildings in the parcel. Click the Editor menu and then click Validate Selection to verify the selected buildings. For one of the buildings the number of stories is 1 and the height is 8, which violates the custom validation rule implemented in the building class extension. Click OK to dismiss the message.

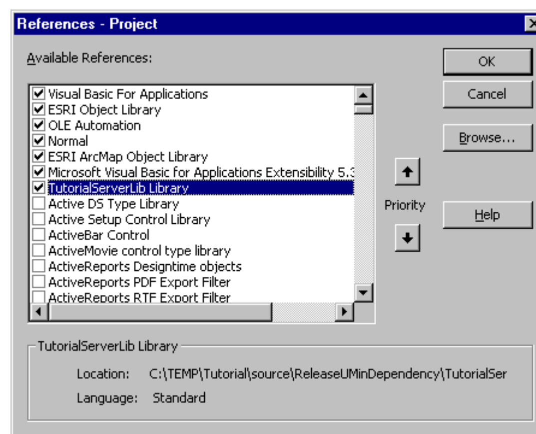


Testing the custom validation implemented by the building class extension.

The last step is to use the services of the IParcel interface programmatically, specifically the calculation of the tax value given a tax rate. We will create a Visual Basic for Applications (VBA) macro that loops through the selected features, asks for the IParcel interface, and uses the method.

To create the VBA macro:

1. In the Tools menu, click Macros, then click Visual Basic Editor.
2. In the Visual Basic Editor, click Tools, then click References.
3. Scroll down and select the TutorialServerLib Library as a reference.



Adding a reference in VBA to the DLL where custom features and class extensions are hosted.

The tutorial server library provides Visual Basic with the types inside the TutorialServer.dll created with ATL/C++. Types such as IParcel or Parcel are defined there.

4. Click OK to accept changes to the references.
5. In the Project Explorer, click the plus sign by the “Project(TutorialMap.mxd)”.
6. Click the plus sign by ArcMap Objects.
7. Double-click ThisDocument.
8. Copy the following code:

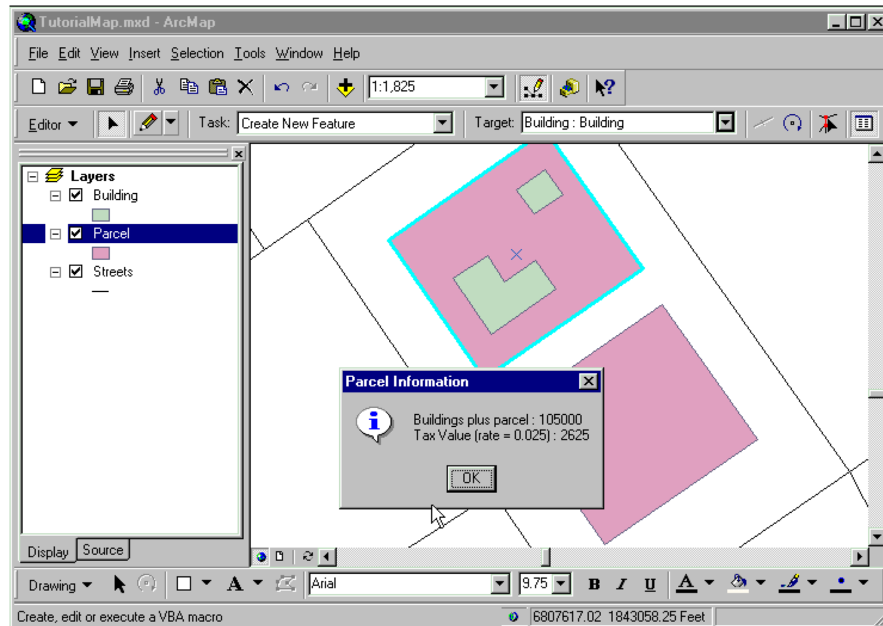
```
Public Sub ParcelInformation()
  Dim pDoc                  As IMxDocument
  Dim pMap                  As IMap
  Dim pLayer                As ILayer
  Dim pFeatLayer  As IFeatureLayer
  Dim pFeatSelection        As IFeatureSelection
  Dim pSelectionSet         As ISelectionSet
  Dim pRow                  As IRow
  Dim pCursor               As ICursor
  Dim pParcel               As IParcel
  Dim sMsg                  As String
  Dim nTotal                As Long

  Set pDoc = ThisDocument
  Set pMap = pDoc.FocusMap
  Set pLayer = pDoc.SelectedLayer

  If pLayer Is Nothing Then
    MsgBox "Please select the parcel layer", vbInformation
    Exit Sub
  End If

  Set pFeatLayer = pLayer
  Set pFeatSelection = pFeatLayer
  Set pSelectionSet = pFeatSelection.SelectionSet

  pSelectionSet.Search Nothing, True, pCursor
  Set pRow = pCursor.NextRow
  If pRow Is Nothing Then MsgBox "Please select a parcel", vbInformation

  Do While Not pRow Is Nothing
    sMsg = ""
    If TypeOf pRow Is IParcel Then
      Set pParcel = pRow
      nTotal = pParcel.CombinedBuildingValue + pParcel.ParcelValue
      sMsg = "Buildings plus parcel : " & nTotal & vbNewLine
      sMsg = sMsg & "Tax Value (rate = 0.025) : " & pParcel.TaxValue(0.025)
      MsgBox sMsg, vbInformation, "Parcel Information"
    End If
    Set pRow = pCursor.NextRow
  Loop
End Sub
```

The macro first finds the selected layer. Then it loops through the selected features and tries to get the IParcel interface. If successful, it uses the interface on the current parcel to get the tax value and report it to the user.

To run the macro:

1. Close the Visual Basic Editor.
2. Select the parcel layer in the TOC.

3. Select the parcel that contains the buildings.
4. In the Tools menu, click Macros, then click Macros.
5. Select the ParcelInformation macro, then click Run.



Using the method TaxValue in the IParcel interface.

To see the results of the rest of your work, try the following:

1. In ArcMap, start editing again, select the parcel with buildings, and move it. The buildings move with the parcel.
2. Select a building and move it outside the parcel. Select the parcel and open the Property Inspector. Notice that the combined building value has changed for the parcel.
3. In the Editor menu, select Stop Editing and then answer Yes to save your edits.