Advanced Editing and Edit Operations

Charles Macleod, Sean Jones

2018 **Esri DEV**Summit **Conference** | Palm Springs, CA

Edit Operations - Session Overview

- Edit Operations
  - Basic workflow
  - Chaining operations
  - EditOperation Callback
- Edit Operation Type
  - Dataset compatibility – short and long modes
    - Toward consistent Undo/Redo, Save/Discard, Cancel edit behavior
- Row–level events
- EditCompletedEvent

Edit Operations – Basic Workflow

- **Best Practice: Use Edit Operations for feature edits (or Inspector)**
  - **Coarse-grained API**
    - **Individual methods for:**
      - **Create, Modify, Delete, Cut, Clip, Merge, Reshape, Rotate, Split, Scale, Move,…**
  - **Executes edits against the underlying datastores**
  - **Adds undo/redo on the Operation Manager***
  - **Invalidates affected layer caches**

  - **Does NOT need to be run on the QueuedTask…..but….many of the API methods you likely will use (to perform an edit) do.**

- ***Non-versioned edits cannot be undone**

Edit Operations – Basic Workflow

- **Example combining many coarse-grained methods (on different datasets):**

```
var editOp = new EditOperation();
editOp.Name = "Simple edit operation";
//Add three points - 1 point to each layer
editOp.Create(pointsLayer1, start_pt);
editOp.Create(pointsLayer2, GeometryEngine.Instance.Move(start_pt, distance, 0.0));
editOp.Create(pointsLayer3, GeometryEngine.Instance.Move(start_pt, distance * 2, 0.0));

//Modify two polygons - 1 polygon in two layers
editOp.Modify(polyLayer2, 1, GeometryEngine.Instance.Buffer(boundary, distance));
editOp.Modify(polyLayer3, 1, GeometryEngine.Instance.Buffer(boundary, distance * 2));

//Execute the operations
editOp.ExecuteAsync();
```

Edit Operations – Execute() and ExecuteAsync()

- **On Execute:**
  - Edits are executed sequentially in a predefined order regardless of how they are declared
    - Creates first, Updates, Deletes, then reshape, clip, etc. Order is not important
    - A session is started per datastore (i.e. “workspace” - if not already started)
  - Individual edits must be independent of one another
    - Dependent edits require *chaining*
  - On success an undo operation is placed on the undo stack (if undo-able)
  - On failure (of any edit), all preceding edits for *that* edit operation are rolled back

- **On Save or Cancel all sessions are ended**

```
if (Project.Current.HasEdits)
   await Project.Current.SaveEditsAsync();
```

  - Save: Edits saved across all datastores (workspaces)
  - Cancel: Edits rolled back across all datastores (workspaces)
  - Undo/redo is cleared

Edit Operations

- Demo
- Simple edit operation

Edit Operations - Chaining

- A chained operation is an edit operation that links to a previous EditOperation
  - Both operations become part of the same undo/redo item
    - Normally, each edit operation has its own undo/redo item

- Use a chained operation when the edit you require is dependent on the results of a previous EditOperation being commited.
  - The classic scenario is adding a feature attachment as part of feature creation.
    - To add the attachment, the OID of the feature must already exist (i.e. the feature must have *been* ~created~ and not be *being* created…)

- Call `editOp.CreateChainedOperation()` to create a chained operation
  - (Instead of "`new EditOperation`")

EditOperations - To Chain Edits:

- Create the “initiating” edit operation (this is the one that shows up in the undo/redo stack)
- Execute it
- If it succeeds, call editOp.CreateChainedOperation() to create the dependent /”chained” operation

```
var editOp = new EditOperation();
editOp.Name = "Create new facility record"; //←this is what shows up as the undo/redo

var attribs = new Dictionary<string, object>();
attribs["SHAPE"] = ...;

long new_oid = -1;
editOp.Create(facilities, attribs, oid => new_oid = oid); //capture oid
if (editOp.Execute()) {//do the create. Note: Execute() needs a QueuedTask
   //create succeeded so chain a new operation to add the attachment
   var chained_op = editOp.CreateChainedOperation();
   chained_op.AddAttachment(facilities, new_oid, @"E:\Attachments\Hydrant_1.jpg");
   chained_op.Execute();
 }
```

Edit Operations

- Chained operation demo

Edit Operations - Callback

- Certain *specialized* scenarios require “custom” handling beyond what the coarse-grained API provides. They are:
  - An edit that must span both GIS and Non-GIS data (non-registered tables)
  - An edit that (needs to) edit data that is not in the project
  - An edit for Annotation text symbol properties (at 2.1)

- You must:
  - Define your edits as part of a *callback*.
  - The callback is executed when the editOp.Execute() is called.
  - Pass in, as parameters, the GIS (and Non-GIS data) to be edited

Edit Operations - Callback

- **Considerations:**
  - **All edits must be done “by hand”**
    - **Cannot mix in edit operation calls within the callback *itself***
  - **Use non-recycling cursors to update rows**
  - **You are responsible for calling Store, CreateRow, etc**
  - **You are responsible for feature or feature dataset invalidation**
    - **Use the provided context that is passed in to you in your callback**

Edit Operations Callback – General Pattern

```
var editOp = new EditOperation();
editOp.Name = “Do callback";
var featLayer1 = ...

editOp.Callback((context) => {
  //Do all your edits here – use non-recycling cursors
  var qf = ...;
  using (var rc = featLayer1.GetTable().Search(qf, false)) {
    while (rc.MoveNext()) {
        context.Invalidate(rc.Current);//Invalidate the row before
        //Do edits...
        rc.Current.Store(); //call store
        context.Invalidate(rc.Current);//Invalidate the row after
    }
  }
  //Edit the non_versioned_table here, etc.

  //Note: can also invalidate any datasets (instead of rows)
 //context.Invalidate(featLayer1.GetTable()); - simpler but less efficient
}, featLayer1.GetTable(), non_versioned_table,...); //Pass as parameters the datasets
                                                    //and tables you will be editing
editOp.Execute();
```

Edit Operations Callback – Key points for Annotation

- Use a non-recycling cursor to access the annotation feature(s)
- Use annotationFeature.GetGraphic() and SetGraphic() to access the stored CIMTextGraphic
  - Avoid trying to modify attributes on the annotationFeature (eg via the Inspector)
  - Experience will be enhanced in 2018 to avoid having to use the callback and CIMTextGraphic to update overrides like Text, Color, Font, Justification, etc.

Edit Operations

- Edit anno Callback demo

Edit Operations

- **Toward Guidelines :**

- **Generally speaking, avoid mixing short and long edit operation types:**
  - **Will provide for a consistent Undo/Redo experience (by extension, Save and Cancel):**

  - **Less stringent requirement for consistent Abort/Cancel Edit behavior**
    - **Direct edits can be aborted (but Direct edits *cannot* be undone/redone)**
    - **Definitely avoid mixing in hosted feature services and DEFAULT on branch versioned**

Edit Operations – Edit Operation Types

- Recap – Edit Operation Types
  - Long – FileGDB, File (shape), and Versioned* data:
    - Starts an underlying edit session
    - Supports Undo/Redo
    - All edits are either saved or discarded at the end of the session
  - Short - Enterprise non-versioned and Hosted feature services:
    - No edit session
    - No Undo/Redo
    - All edits are saved or discarded immediately
  - *DEFAULT on branch-versioned exhibits short transaction behavior

Edit Operations – Edit Operation Type

- By default, `EditOperationType` property returns null or “mixed mode”
  - long and short in the same operation is ok
  - Explicitly set `EditOperationType` = to Long or Short to force dataset compatibility
    - Setting `EditOperationType` = `Long` disallows datasets with “short” semantics
    - And vice versa for Setting `EditOperationType` = `Short`

- If the `EditOperationType` is explicitly set, Execute will fail if your operation mixes datasets with “short” and “long” semantics.
  - `editOp.Execute()` will return `false`

EditOperations - Undo and Redo

- For Undo and Redo
- Either:
  - Call Undo/Redo on the *last edit operation* (assumes you acquired a reference!)
    - `editOp.UndoAsync()` + overloads, `RedoAsync()` + overloads
- Or:
  - Access the Map's OperationManager and call *its* undo/redo
    - `Map.OperationManager.UndoAsync()` + overloads, `RedoAsync()` + overloads

EditOperations - Save or Discard

- For Save or Discard:
  - Call `SaveEditsAsync()` or `DiscardEditsAsync()` on the current project
    - Use `HasEdits` to check if there are pending edits

```
if (Project.Current.HasEdits)
    await Project.Current.SaveEditsAsync();
```

EditOperations - RowEditEvents and CancelEdit

- For Row-level events use:
  - `RowCreatedEvent` – raised whenever a row is (being) created
  - `RowChangedEvent` – raised whenever a row is (being) modified
  - `RowDeletedEvent` – raised whenever a row is (being) deleted

  - Call `CancelEdit` on the `RowChangedEventsArgs` to abort the transaction*

```
public sealed class RowChangedEventArgs  {
  public Row Row { get; }
  public Guid Guid { get; }
  //CancelEdit + overloads...
  public void CancelEdit(string errorMessage, bool canOverride = false);
}
```

- *CancelEdit supported per dataset characteristics previously discussed

EditOperations - RowEditEvents

- **Subscribe for row events on a “per” feature class/table basis to:**
  - **Make changes to *the* row being edited (passed in the event argument)**
    - **Changes to the row become part of the on-going transaction**
  - **Validate row attributes that have been changed**
  - **Cancel row transactions (e.g. those that fail validation)**
  - **Log edits**

```
RowChangedEvent.Subscribe((rc) => {
  //Validate any change to “police district”
  if (rc.Row.HasValueChanged(rc.Row.FindField(“POLICE_DISTRICT"))) {
     if (FailsValidation(rc.Row[“POLICE_DISTRICT"])) //Cancel edits with invalid “police district” values
        rc.CancelEdit($"Police district {rc.Row[“POLICE_DISTRICT"]} is invalid");
  }
 }, crimes_fc);
```

EditOperations - RowEditEvents

- **Note:**
  - **If you wish to edit a separate table in the event callback (eg for audit trial) you cannot use an edit operation.**
    - **Use the ArcGIS.Core.Data API. For example:**

```
var rowBuffer = history_table.CreateRowBuffer();
rowBuffer[“DESCRIPTION”] = “Edit description here”;
history_table.CreateRow(rowBuffer)
```

  - **Other:**
    - **RowEvent callbacks are called on the QueuedTask thread.**
    - **You do NOT need to wrap code in a QueuedTask within your row event callback**
      - **It is redundant**

Edit Operations

- Demo row events and non-versioned and versioned edits

EditOperations - EditCompletedEvent

- Global event – once you subscribe, you receive it for all transactions on all datasets in the Pro session:
  - EditOperations, Save edits, Cancel edits, Undo, Redo
- Check the passed-in EditCompletedEventArgs and its…:
  - public IReadOnlyDictionary<MapMember, IReadOnlyCollection<long>> Creates,
  - public IReadOnlyDictionary<MapMember, IReadOnlyCollection<long>> Modifies and
  - public IReadOnlyDictionary<MapMember, IReadOnlyCollection<long>> Deletes
  - …to determine the editing activity for the given operation, save, undo, etc. that triggered the event.

Edit Operations - EditCompletedEvent

```
EditCompletedEvent.Subscribe(HandleEdits);

private Task HandleEdits(EditCompletedEventArgs e) {
    var selSet = new SelectionSet(MapView.Active.Map);
    foreach (var pair in e.Creates) {// handle adds. In your code check e.Creates != null
      MapMember member = pair.Key;
      var oids = pair.Value;
      selSet.Add(member, oids);
    }

    foreach (var pair in e.Modifies) {// handle modifies
      MapMember member = pair.Key;
      var oids = pair.Value;
      selSet.Add(member, oids);
    }
    // assign it to the map
    return selSet.Set();
  }
```

EditOperations - Summary

- EditOperations are the preferred editing API for Pro
  - Coarse-grained methods that can be combined into a single transaction
  - Use ChainedOperations when one operation is dependent on another
  - Use Callback in special scenarios only
- Avoid mixing short and long edit operation types to provide consistent undo/redo, save/discard, and cancel edit behavior
  - OperationManager for undo/redo
  - Project for save/discard
- Row events for validation, cancel edit, logging, etc.
- EditCompletedEvent global event fired after all edit operations, save, cancel, undo, etc.

ArcGIS Pro SDK for .NET Tech Sessions

Date	Time	ArcGIS Pro SDK for .NET Tech Sessions	Location
Tue, Mar 06	1:00 pm - 2:00 pm	An Overview of the Geodatabase API	Mojave Learning Center
	2:30 pm - 3:30 pm	Beginning Pro Customization and Extensibility	Primrose A
	5:30 pm - 6:30 pm	Beginning Editing and Editing UI Patterns	Mojave Learning Center
Wed, Mar 07	10:30 am - 11:30 am	Mapping and Layout	Pasadena/Sierra/Ventura
	1:00 pm - 2:00 pm	Advanced Pro Customization and Extensibility	Santa Rosa
	2:30 pm - 3:30 pm	Pro Application Architecture Overview & API Patterns	Mesquite G-H
	4:00 pm - 5:00 pm	Advanced Editing and Edit Operations	Santa Rosa
Thu, Mar 08	9:00 am - 3:30 pm	Getting Started Hands-On Training Workshop	Mojave Learning Center
	5:30 pm - 6:30 pm	Working with Rasters and Imagery	Santa Rosa
Fri, Mar 09	8:30 am - 9:30 am	An Overview of the Utility Network Management API	Mesquite G-H
	10:00 am - 11:00 am	Beginning Pro Customization and Extensibility	Primrose A
	1:00 pm - 2:00 pm	Advanced Pro Customization and Extensibility	Mesquite G-H

ArcGIS Pro SDK for .NET Demo Theater Sessions

Date	Time	ArcGIS Pro SDK for .NET Demo Theater Presentation	Location
Tue, Mar 06	1:00 pm - 1:30 pm	Getting Started	Demo Theater 1 - Oasis 1
	4:00 pm - 4:30 pm	Custom States and Conditions	Demo Theater 2 - Oasis 1
Wed, Mar 07	5:30 pm - 6:00 pm	New UI Controls for the SDK	Demo Theater 2 - Oasis 1
	6:00 pm - 6:30 pm	Raster API and Manipulating Pixel Blocks	Demo Theater 2 - Oasis 1

ArcGIS Pro Road Ahead Sessions

Date	Time	ArcGIS Pro SDK for .NET Demo Theater Presentation	Location
Tue, Mar 06	4:00 pm – 5:00 pm	ArcGIS Pro: The Road Ahead	Oasis 4
Thu, Mar 08	4:00 pm – 5:00 pm	ArcGIS Pro: The Road Ahead	Primrose B


esri
THE
SCIENCE
OF
WHERE

Edit Operations – Toward Guidelines and Dataset Compatibility

- How to determine dataset compatibility – (*if* your goal *is* a consistent edit experience)

- Analyze the underlying layer dataset and connection properties
  - Geodatabase type – FileSystem, LocalDatabase, RemoteDatabase, Service
  - Registration type – Versioned, Non-versioned, VersionedWithMoveToBase
  - Connection properties – Database connection, service connection, etc

- Can infer versioned/non-versioned, undo/redo, cancelable, etc. from the combination of these properties and if it is ok to mix

Edit Operations - Determining Dataset Compatibility – Basic Functions

```
public static class FeatureLayerExtensions {

   public static RegistrationType GetRegistrationType(this FeatureLayer featLayer) {
     //Will throw CalledOnWrongThreadException if called from the UI!
     return featLayer.GetFeatureClass()?.GetRegistrationType() ?? RegistrationType.Nonversioned;
   }

   public static GeodatabaseType GetGeodatabaseType(this FeatureLayer featLayer) {
     return ((Geodatabase)featLayer.GetFeatureClass()?.GetDatastore()).GetGeodatabaseType();
   }

   public static Connector GetConnnectionProperties(this FeatureLayer featLayer) {
     return featLayer.GetFeatureClass()?.GetDatastore().GetConnector();
   }
```

Edit Operations - Determining Dataset Compatibility – Versioned, Non-Versioned

```
public static class FeatureLayerExtensions {
   //FeatureLayerExtensions Continued (1)…
   public static bool IsVersioned(this FeatureLayer featLayer) {
     return featLayer.GetRegistrationType() != RegistrationType.Nonversioned;
   }

   public static bool IsBranchVersioned(this FeatureLayer featureLayer) {
     //does the underlying Geodatabase support versioning?
     var gdb = featureLayer.GetFeatureClass().GetDatastore() as Geodatabase;
     if (!gdb?.IsVersioningSupported() ?? false)
       return false;
     var cprops = featureLayer.GetConnnectionProperties();
     if (cprops is DatabaseConnectionProperties) {//Utility network fc only
       return ((DatabaseConnectionProperties)cprops).Branch.Length > 0;
     }
     return cprops is ServiceConnectionProperties;
   }
```

Edit Operations - Determining Dataset Compatibility – Undo,Redo

```
public static class FeatureLayerExtensions {

   //FeatureLayerExtensions Continued (3)…
   public static bool SupportsUndoRedo(this FeatureLayer featLayer) {

     var gdbType = featLayer.GetGeodatabaseType();
     if (gdbType == GeodatabaseType.FileSystem || gdbType == GeodatabaseType.LocalDatabase)
       return true;//File GDB and Shape Files
     if (gdbType == GeodatabaseType.RemoteDatabase && featLayer.IsVersioned())
       return true;
     //Branch versioned is special case
     if (featLayer.IsBranchVersioned()) {
       var vmgr = ((Geodatabase) featLayer.GetFeatureClass().GetDatastore()).GetVersionManager();
       return vmgr.GetCurrentVersion().GetParent() != null;//non-default support undo/redo
     }
     return false;
   }
```

Edit Operations - Determining Dataset Compatibility – CancelEdit

```
public static class FeatureLayerExtensions {

   //FeatureLayerExtensions Continued (4)…
   public static bool SupportsCancelEdit(this FeatureLayer featLayer) {

     if (featLayer.GetGeodatabaseType() != GeodatabaseType.Service)
       return true;
     //Branch versioned is special case
     if (featLayer.IsBranchVersioned()) {
       var vmgr = ((Geodatabase)featLayer.GetFeatureClass().GetDatastore()).GetVersionManager();
       return vmgr.GetCurrentVersion().GetParent() != null;//non-default supports cancel
     }
     return false;//Hosted fs do not support cancel
   }
```

Edit Operations

- Questions?