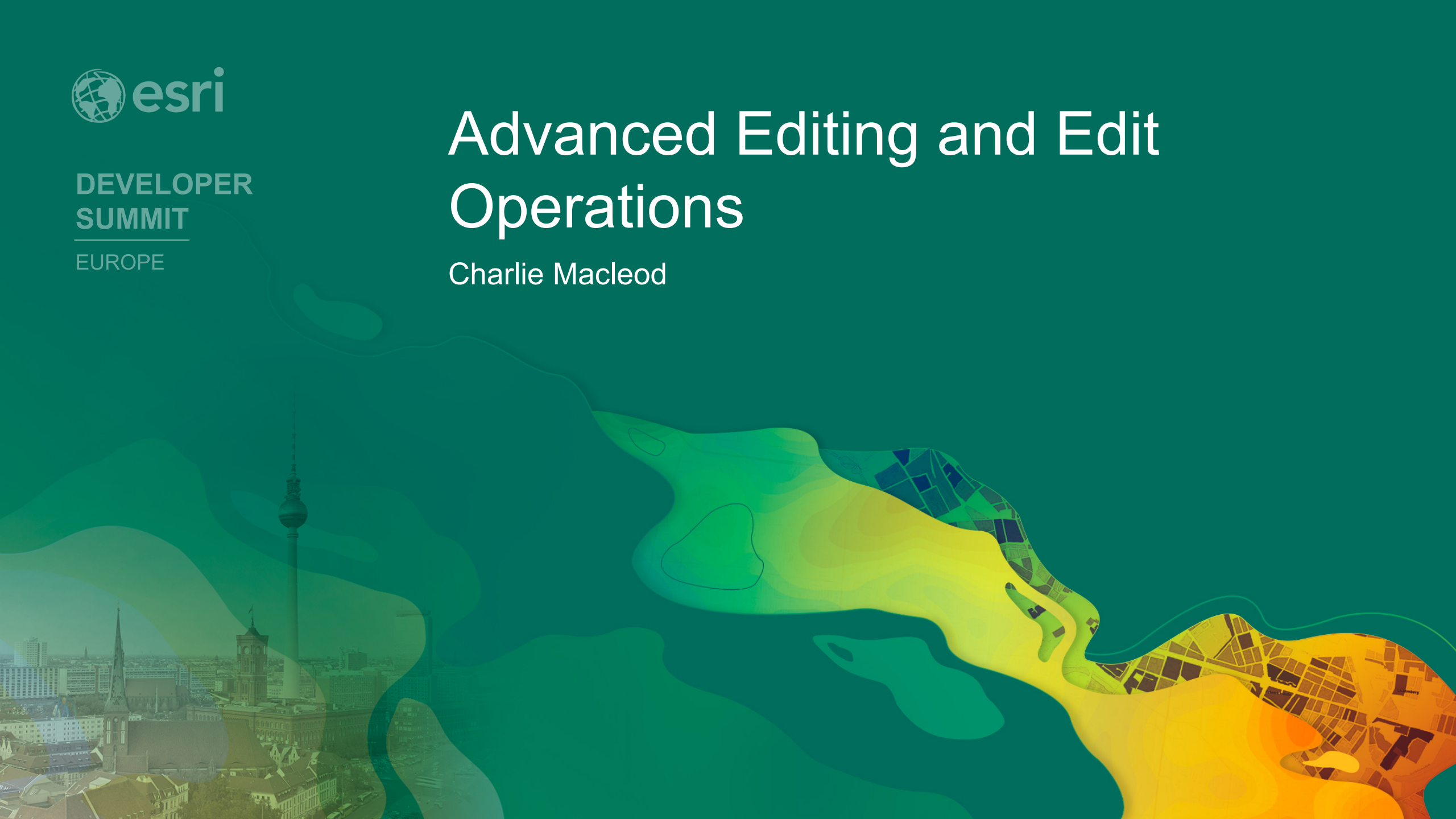
esri

DEVELOPER SUMMIT

EUROPE

Advanced Editing and Edit Operations

Charlie Macleod



Edit Operations - Session Overview

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

Edit Operations – Basic Workflow

- Recap:
- 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 MapMember caches

- *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
    - 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

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 completed.
  - 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`")
- Note: Utility Network users use CreateEx overloads and *Associations*

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 which there is no coarse-grained API
    - Use of editOp.Modify or Inspector is also possible
  - At 2.1, editing Edit Annotation text properties

- For these scenarios, use a Callback
- The callback is executed when the editOp.Execute() is called.
  - Pass in, as parameters, at least one dataset per datastore that will 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
    - Retrieve all necessary fields in your query filters
    - (Safest to use qf.SubFields = “*”)
  - 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 demo

Edit Operations – Dataset Compatibility and EditOperationType

- public Nullable<EditOperationType> EditOperationType { get; set;}

Visual Basic (Declaration) C#

```
public enum EditOperationType : System.Enum, System.IComparable,
```

Members

Member	Description
Long	A long transaction. Edits can be placed on the undo stack.
Short	A short transaction. Edits are committed immediately.

- By default, `EditOperationType` property returns null or “mixed mode”
  - long and short in the same operation is ok

Edit Operations – EditOperationType

- In mixed mode (`editOp.EditOperationType == null`)
  - Dataset Compatibility is not checked (ok to mix)
  - Long transactions go on the undo/redo stack
    - Edit sessions established on underlying datastores
  - Short transactions are committed immediately
    - No edit session

Edit Operations – EditOperationType

- To enforce dataset compatibility explicitly set `EditOperationType` = to Long or Short
  - Setting `EditOperationType = Long` disallows datasets with “short” semantics
  - And vice versa for Setting `EditOperationType = Short`
  - Generally speaking, setting EditOperationType provides for a consistent Undo/Redo experience (by extension, Save and Discard):

- Execute will fail if your operation mixes datasets with “short” and “long” semantics.
  - `editOp.Execute()` will return `false`

Edit Operations – Determining Edit Operation Type

- Check underlying `GeodatabaseType` and `RegistrationType` of the feature class:

```
((Geodatabase)featLayer.GetFeatureClass().GetDatastore()).GetGeodatabaseType();
 featLayer.GetFeatureClass()?.GetRegistrationType();
```

GeodatabaseType	RegistrationType	Version	EditOperationType	Example
FileSystem	N/A	N/A	LONG	Shape file
LocalDatabase	N/A	N/A	LONG	File GDB
RemoteDatabase	Versioned	N/A	LONG	Enterprise
RemoteDatabase	NonVersioned	N/A	SHORT	Enterprise (Direct)
Service	NonVersioned	N/A	SHORT	Hosted or “Standard”
Service	Versioned	Default	SHORT	Branch Versioned
Service	Versioned	Named	LONG	Branch Versioned

Edit Operations – Edit Operation Types

- Summary of characteristics by Long and Short type

GeodatabaseType	RegistrationType	CancelEdit	Undo/Redo	Save/Discard
FileSystem	N/A	YES	YES	YES
LocalDatabase	N/A	YES	YES	YES
RemoteDatabase	Versioned	YES	YES	YES
RemoteDatabase	NonVersioned	YES	NO	NO
Service	NonVersioned	PARTIAL*	NO	NO
Service	Versioned (Default)	YES	NO	NO
Service	Versioned (Named)	YES	YES	YES

LONG SHORT

*2.2 and earlier - Create cannot be canceled

EditOperations - Undo and Redo

- For datasets with LONG semantics…
  - Call Undo/Redo on the edit operation
    - `editOp.UndoAsync()` + overloads, `RedoAsync()` + overloads
- 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();
    //await Project.Current.DiscardEditsAsync();
```

EditOperations - Demo

- Demo Dataset Compatibility

EditOperations - RowEditEvents

- Row Events available for:
  - Creates - `RowCreatedEvent`
  - Changes (updates) - `RowChangedEvent`
  - Deletes - `RowDeletedEvent`

- Events are on a per-dataset basis
  - Broadcast only for those datasets for which you registered for the given event
  - Use Subscribe and Unsubscribe to register/unregister

```
var fc = featLayer.GetFeatureClass();

SubscriptionToken token = RowCreatedEvent.Subscribe((rc) => RowEventHandler(rc), fc);
RowCreatedEvent.Unsubscribe(token);
```

EditOperations - RowEditEvents

- Use Row Events 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
    - Changes to other tables must use Core.Data API within row events

EditOperations - RowEditEvents

- Use the RowChangedEventArgs parameter to:
  - Access the Row being edited
  - EditType (Create, Change, Delete)
  - Call `CancelEdit` on the `RowChangedEventsArgs` to abort the transaction

```
public sealed class RowChangedEventArgs  {
  public Row Row { get; }
  public EditType EditType { get; }
  public Guid Guid { get; }

  //CancelEdit + overloads...
  public void CancelEdit(string errorMessage, bool canOverride = false);
}
```

EditOperations - RowEditEvents

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

Edit Operations

- Demo row events

Advanced Editing - Summary

- EditOperations
  - 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
- Dataset compatibility and EditOperationType implications
  - Explicitly set for consistent undo/redo, save/discard behavior
- Row events for validation, cancel edit

Thank You to Our Sponsors

con·terra

geocom


esri
THE
SCIENCE
OF
WHERE

EditOperations - EditCompletedEvent

- **Subscribe for notifications of all transactions on all datasets in Pro session.**
  - **Global: Across any Map in Project**
  - **All Creates, Modify/Updates, Deletes, Undo/Redo, Save, Discard, Reconcile, Post, Change Version**

```
ArcGIS.Desktop.Editing.Events.EditCompletedEvent.Subscribe(HandleEdits);

private Task HandleEdits(EditCompletedEventArgs ec) {
  ...
```

EditOperations - EditCompletedEvent

- **Check the incoming EditCompletedEventArgs.CompletedType for an indication of the initiating edit type:**

```
private Task HandleEdits(EditCompletedEventArgs ec) {
    ...
    switch (ec.CompletedType) {
        case EditCompletedType.Operation:
          //Create, Modify, Delete, Reconcile, Post
          //Change Version
          break;
        case EditCompletedType.Undo:
        case EditCompletedType.Redo:
        case EditCompletedType.Save:
        case EditCompletedType.Discard:
          //TODO
          break;
    }
```

EditOperations - EditCompletedEvent

- For edits that Create, Modify, Delete features, check the corresponding Creates, Modifies, Deletes collections…
  - 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 that triggered the event.
  - Includes:
    - All Create, Modify/Update, Delete edit operations
    - Undo, Redo

Edit Operations - EditCompletedEvent

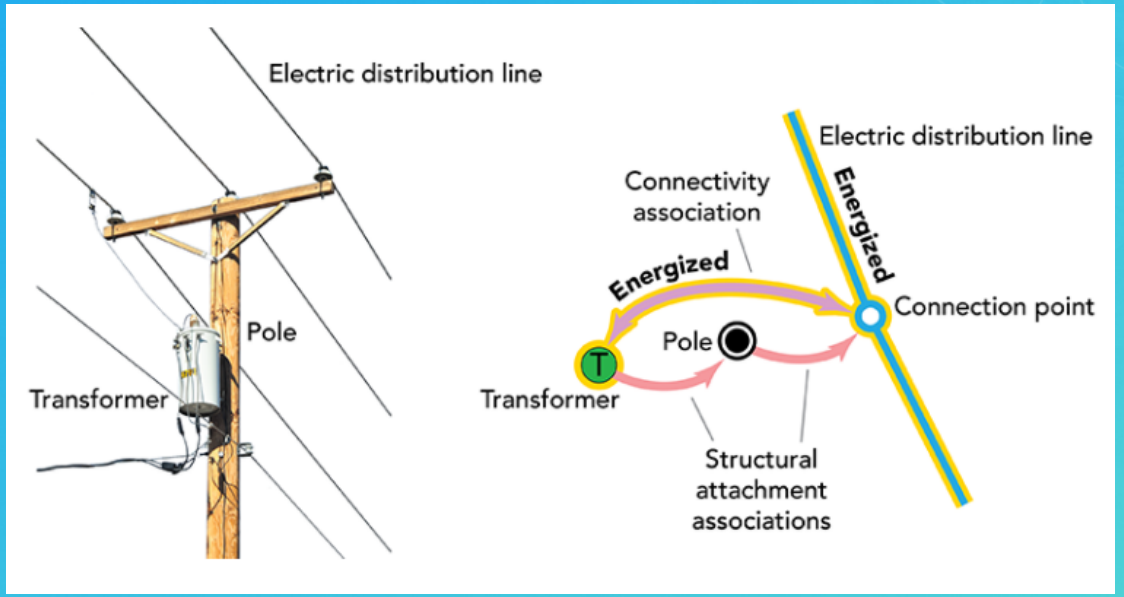
```
EditCompletedEvent.Subscribe(HandleEdits);

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();
  }
```

Edit Operations and Utility Network Associations

- In a Utility Network Topology features are related to each other through geometric coincidence and association mechanisms
- An association is used when features to be "*associated*" are not necessarily geometrically coincident. One feature can have many associations.
- To create an association:
  - Create or select the features to be associated
  - Create the association description
  - Create the association



Edit Operations and CreateEx

- It is (usually) desirable to create the features and association in a single transaction
  - Similar to attachments, the challenge is we need the features before we can associate them
- Use EditOperation.CreateEx (not “Create”)
  - Returns a RowToken or “placeholder” that can be used in lieu of the feature (“to be” created) to create the association (via RowHandle)

```
public RowToken CreateEx(Layer layer, Geometry geometry, Dictionary<string, object> values);
public RowToken CreateEx(Table table);
public RowToken CreateEx(FeatureLayer layer, Geometry geometry);
public RowToken CreateEx(Table table, Dictionary<string, object> values);
public RowToken CreateEx(MapMember member);
public RowToken CreateEx(MapMember member, Dictionary<string, object> values);
public RowToken CreateEx(EditingTemplate template, Geometry geometry);
```

- Call EditOperation.Execute to create the features and the association in one shot.

Edit Operations and CreateEx

```
var editOp = new EditOperation();
editOp.Name = “Create pole + transformer bank. Attach bank to pole";

//Create the transformer and pole
RowToken transToken = editOp.CreateEx(transformerLayer, transAttrib);
RowToken poleToken = editOp.CreateEx(poleLayer, poleAttrib);

//Create association using the row tokens not features
var poleAttachment = new StructuralAttachmentAssociationDescription(
     new RowHandle(poleToken), new RowHandle(transToken));

editOp.Create(poleAttachment);

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

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 Discard):**

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

EditOperations – CancelEdit Considerations for Feature Services

- **Recall - CancelEdit varies across feature service types**
  - All support cancel of modify and delete
  - Create cannot be cancelled for Hosted and Standard
    - Without a VMS, the feature OID must be acquired before the event.
    - Hence, when the event fires, the feature has already been created…
      - ….and so cannot be cancelled

	Cancel Edit
File (Shape)	Yes
File Geodatabase	Yes
Enterprise Versioned	Yes
Feature Service* Branch Versioned	Yes
Enterprise Direct	Yes
Feature Service Hosted	Partial
Feature Service "Standard"	Partial

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 – 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
   }
```

EditOperations - RowEditEvents - Special Considerations for Feature Services

- Difference between Create for Branch vs Create for Hosted/Standard
  - Branch:
    - The VMS can pre-allocate OIDs so the Object ID of the feature to-be-created is already known
    - OID is provided to RowCreate ~before~ the applyEdits call (hence it can be cancelled)
  - Hosted/Standard
    - For non-VMS, the row has to be created to acquire its OID (no pre-allocation)
    - OID is provided to RowCreate ~after~ the applyEdits call (hence it cannot be cancelled)

Edit Operations

- Questions?