

Utility Network C# SDK

Beta 1

February 2, 2017

Introduction

1. Overview
2. Other Ways to Program Against the Utility Network
3. Organization of the Utility Network SDK

1. Overview

- The Utility Network C# SDK is a managed .NET SDK that provides access to the utility network
 - It is an object-oriented SDK that aligns with modern C# practices and existing frameworks
 - It adheres to the principles and architecture of the general [Pro SDK](#)
 - This document assumes a basic understanding of the Utility Network information model
-
- Architectural Topics
 - DML-only (Data Manipulation Language)
 - Threading
 - Garbage Collection
 - Services-Only
-
- Namespaces

Architecture:

DML-only (Data Manipulation Language)

- The utility network API is a DML-only (Data Manipulation Language) API
 - Schema creation and modification operations such as creating domain networks, adding and deleting rules, etc., need to be performed using Python
 - This is in alignment with the rest of the [Geodatabase API](#)
 - Python can be called from C# by using the [Geoprocessing API](#)

```
var args = Geoprocessing.MakeValueArray(utilityNetworkPath, @"ALL", @"rules.csv");  
var result = Geoprocessing.ExecuteToolAsync("un.ImportRules", args);
```

Architecture: Threading

- Almost all of the methods in the utility network API should be called on the MCT (Main CIM Thread)
 - The API reference documentation on the methods that need to run on the MCT are specified as such
 - These method calls should be wrapped inside the `QueuedTask.Run` call
 - Failure to do so will result in `ConstructedOnWrongThreadException` being thrown
- Read [Working with multithreading in ArcGIS Pro](#) to learn more

```
Task t = QueuedTask.Run(() =>
{
    //put utility network code here
});
```

Architecture:

Garbage Collection

- By default, locks and connections on database objects held by .NET objects are released at non-deterministic times (when garbage collection runs)
 - As a consequence, connections and locks on any dataset referenced by the .NET object will also be held
- Using blocks or calling Dispose on objects after use provides greater control on when connections and file system locks are released

■ Example

```
UtilityNetwork utilityNetworkIndeterministic = geodatabase.OpenDataset<UtilityNetwork>(utilityNetworkName);  
  
using (UtilityNetwork utilityNetworkDeterministic = geodatabase.OpenDataset<UtilityNetwork>(utilityNetworkName))  
{  
    // more code here  
}
```

- Locks acquired by `utilityNetworkIndeterministic` are released when garbage collection runs
- Locks acquired by `utilityNetworkDeterministic` are released at the end of the using block

Architecture: Services-only

- In standard usage, access to a utility network takes place via services and not client-server
- The utility network SDK is likewise based on a services-only architecture and is designed accordingly
- Implications of this architecture are described throughout this document as appropriate
- The Beta release currently provides client-server access to the utility network. This will be removed before final release.

Namespaces

- The main items of the utility network are included within the `ArcGIS.Core.Internal.Data.UtilityNetwork` namespace, unless otherwise noted
 - Tracing items are included within `ArcGIS.Core.Internal.Data.UtilityNetwork.Trace`
- This will be changed to `ArcGIS.Core.Data.UtilityNetwork` before shipping the first release.

2. Other Ways To Access the Utility Network

- In addition to the ArcGIS Pro Managed SDK, there are other ways to program against a utility network:
 - Geoprocessing models and Python scripts
 - Directly coding against the REST APIs
 - SOEs and SOIs may be written with ArcObjects may be written to run on the server

```
#Update the Subnetworks
arcpy.AddMessage("Update Subnetworks")
arcpy.UpdateSubnetwork_un(utilityNetwork, domainNetworkName, "Subtransmission", "ALL")
arcpy.UpdateSubnetwork_un(utilityNetwork, domainNetworkName, "Medium Voltage", "ALL")
arcpy.UpdateSubnetwork_un(utilityNetwork, domainNetworkName, "Medium Voltage Multifeed", "ALL")
arcpy.UpdateSubnetwork_un(utilityNetwork, domainNetworkName, "Low Voltage Radial", "ALL")
arcpy.UpdateSubnetwork_un(utilityNetwork, domainNetworkName, "Low Voltage Mesh", "ALL")
arcpy.AddMessage("Finished updating subnetworks")
```

validateNetworkTopology

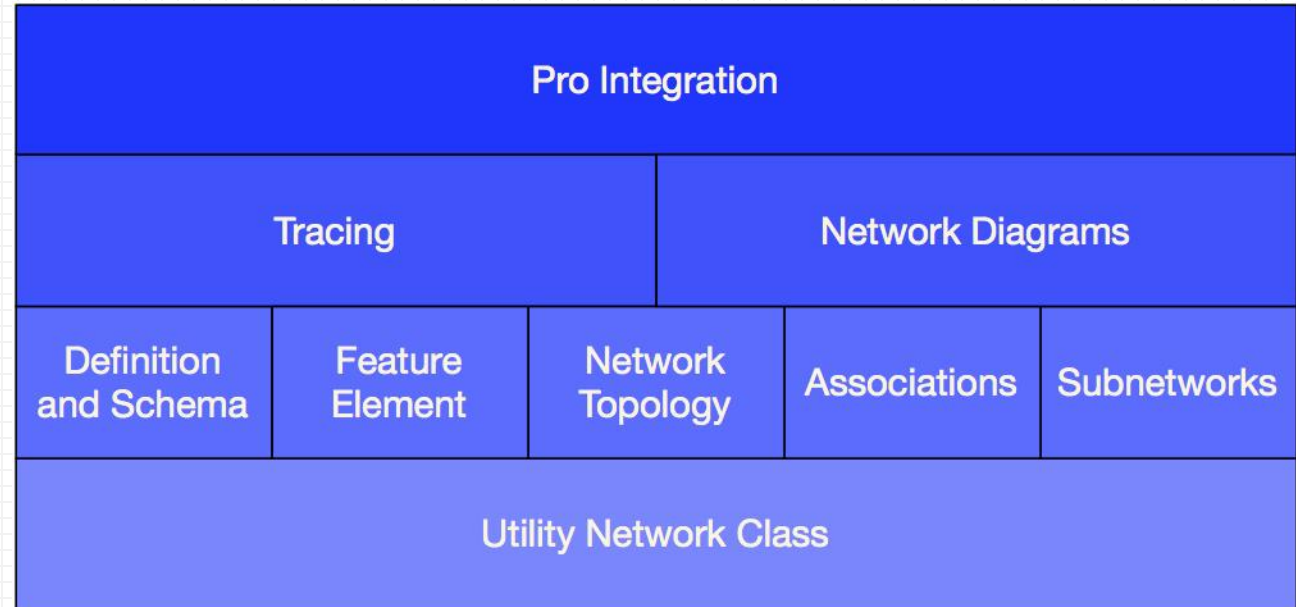
POST only

Validating the network topology for a utility network maintains consistency between feature editing space and network topology space. Validating a network topology may include all or a subset of the dirty areas present in the network.

Parameter	Details
f	Description: Optional parameter representing the output format of the response (default is JSON).
gdbVersion	Description: The name of the GDB version. Syntax: gdbVersion=<version>
sessionId	Description: The token (guid) used to lock the version. Syntax: sessionId=<guid>
validateArea	Description: The envelope of the area to validate.

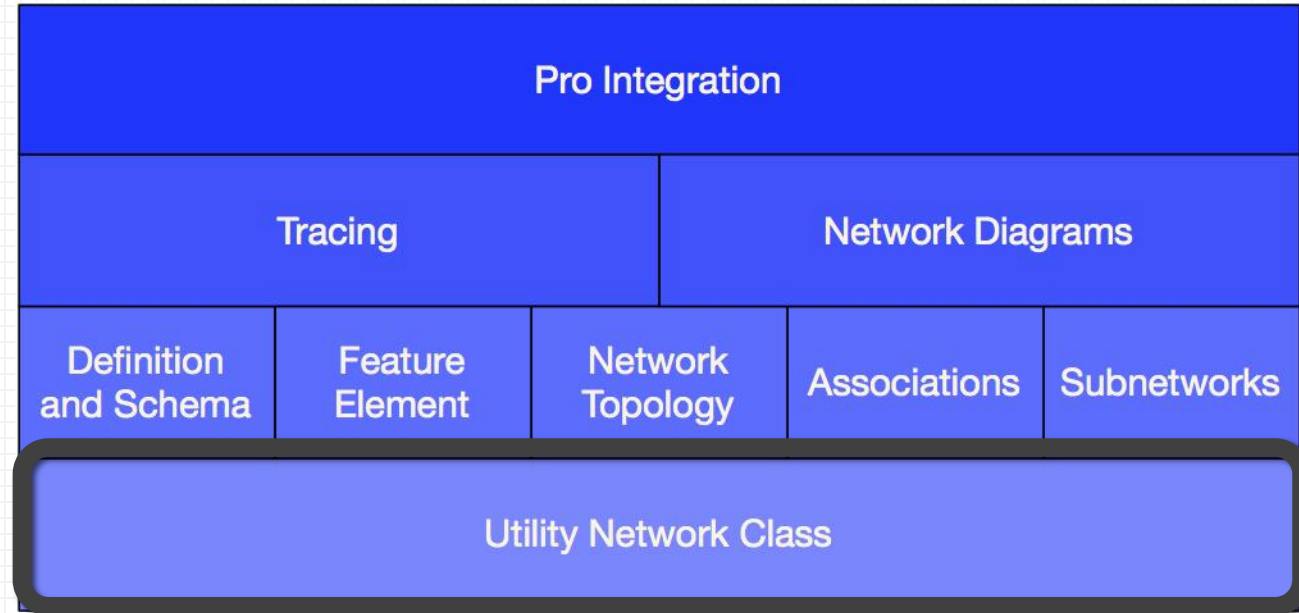
3. Organization of the Utility Network SDK

- This document logically divides the SDK into nine different sections
- The diagram at right provides a **functional** organization of the SDK
 - Strictly speaking, the SDK is a collection of classes
 - Not a layered architecture



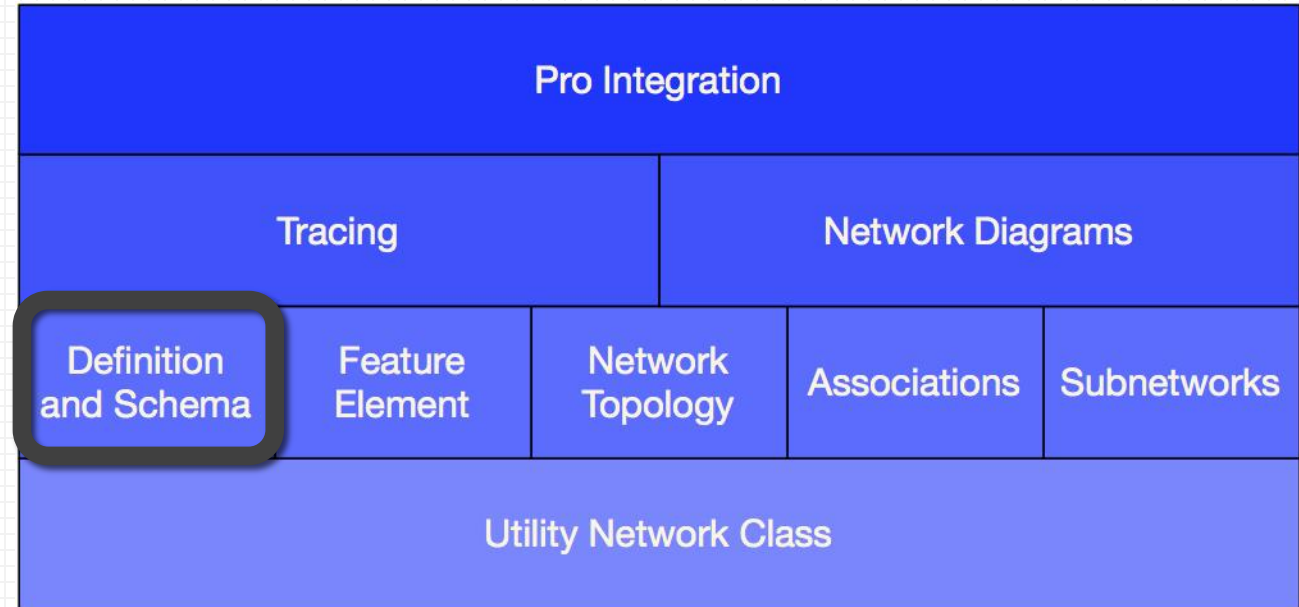
3. Organization of the Utility Network SDK

- **Utility Network Class** covers the root object that provides access to the Utility Network SDK



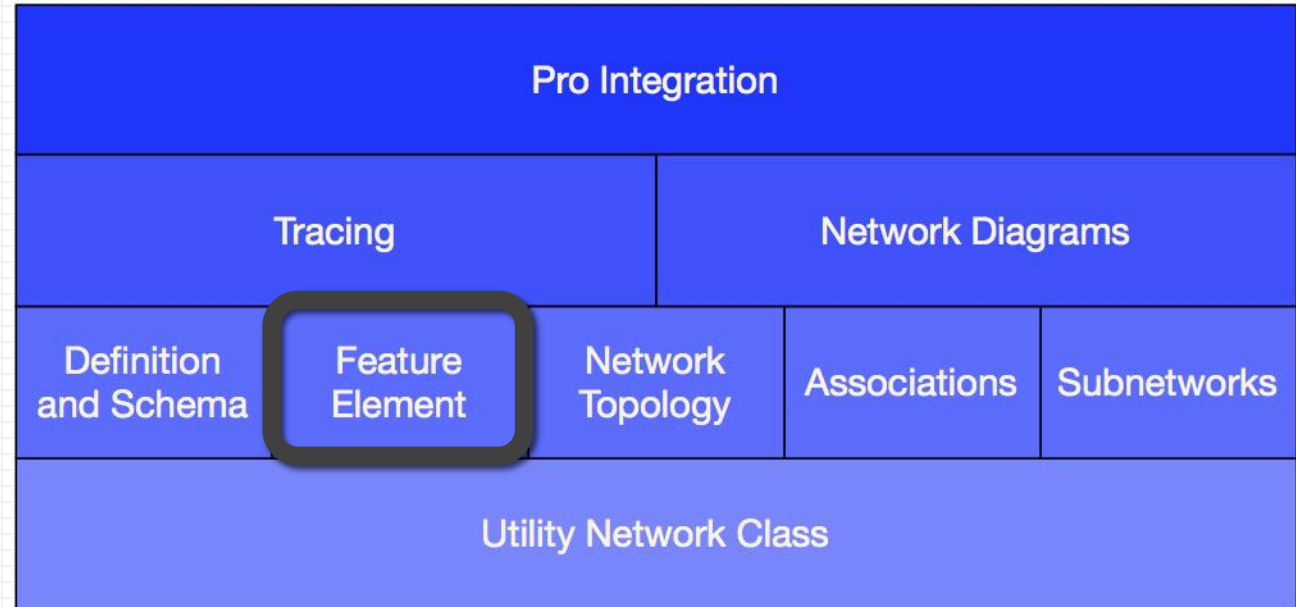
3. Organization of the Utility Network SDK

- **Definition and Schema** describes the classes and methods that provide information about the utility network schema



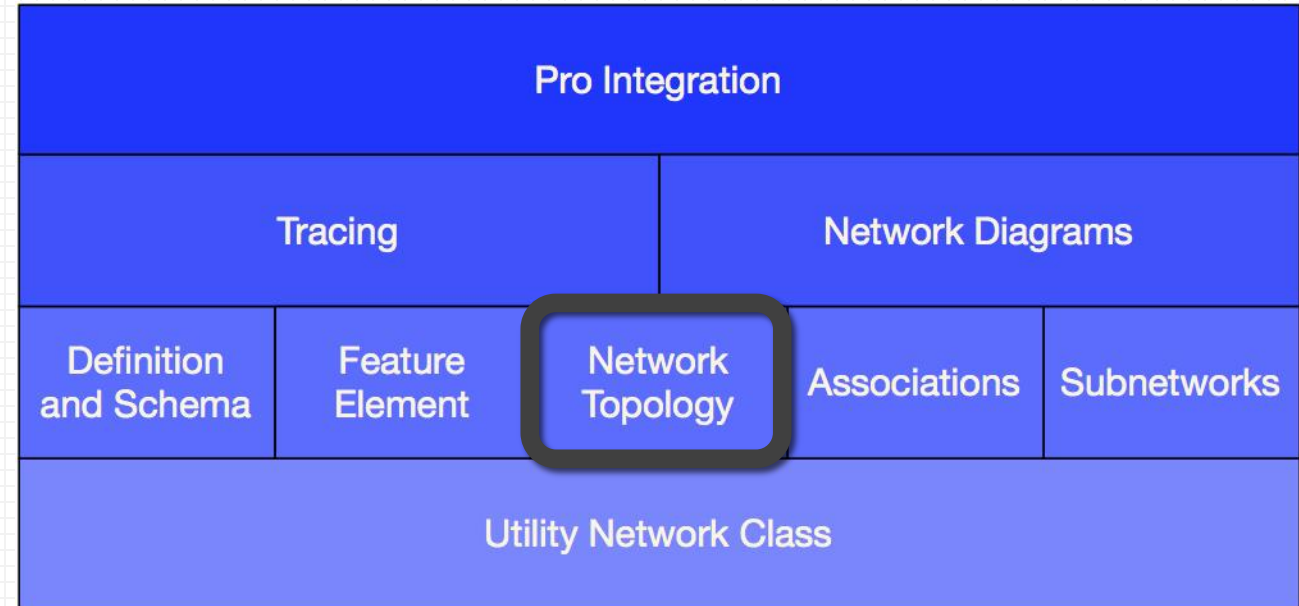
3. Organization of the Utility Network SDK

- **Feature Element** covers the basic encapsulation of a feature in the utility network SDK



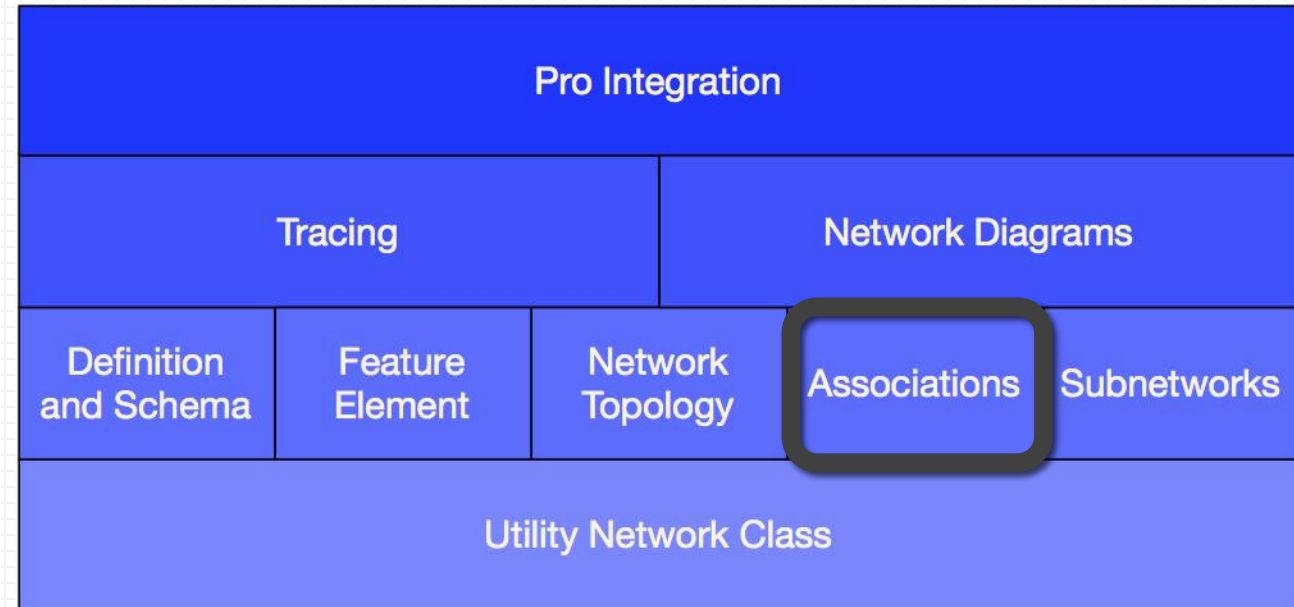
3. Organization of the Utility Network SDK

- **Network Topology** covers routines that query the topological index



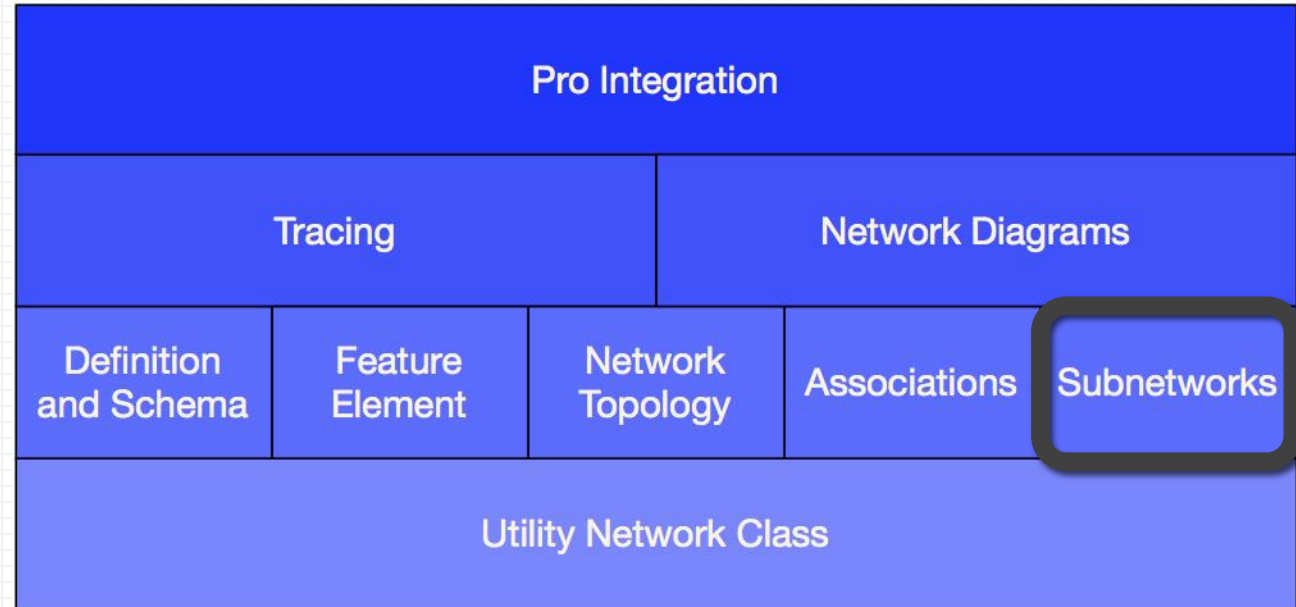
3. Organization of the Utility Network SDK

- **Associations** covers routines that query and edit associations between utility network features



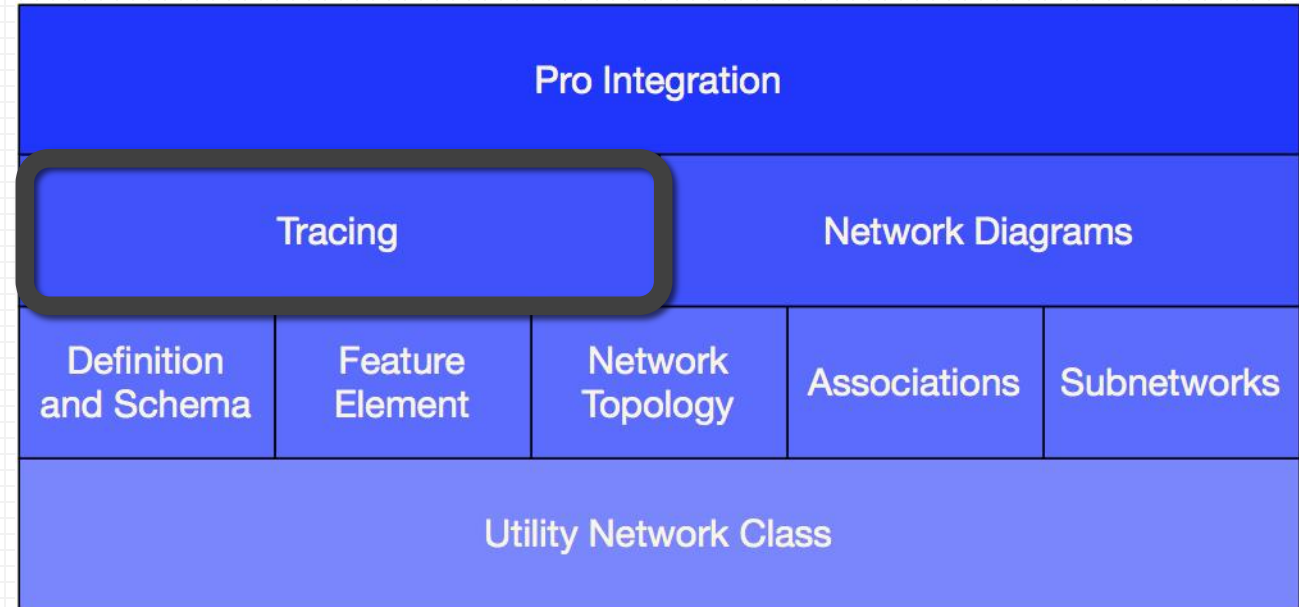
3. Organization of the Utility Network SDK

- **Subnetworks** provides classes and routines to query and edit utility network subnetworks



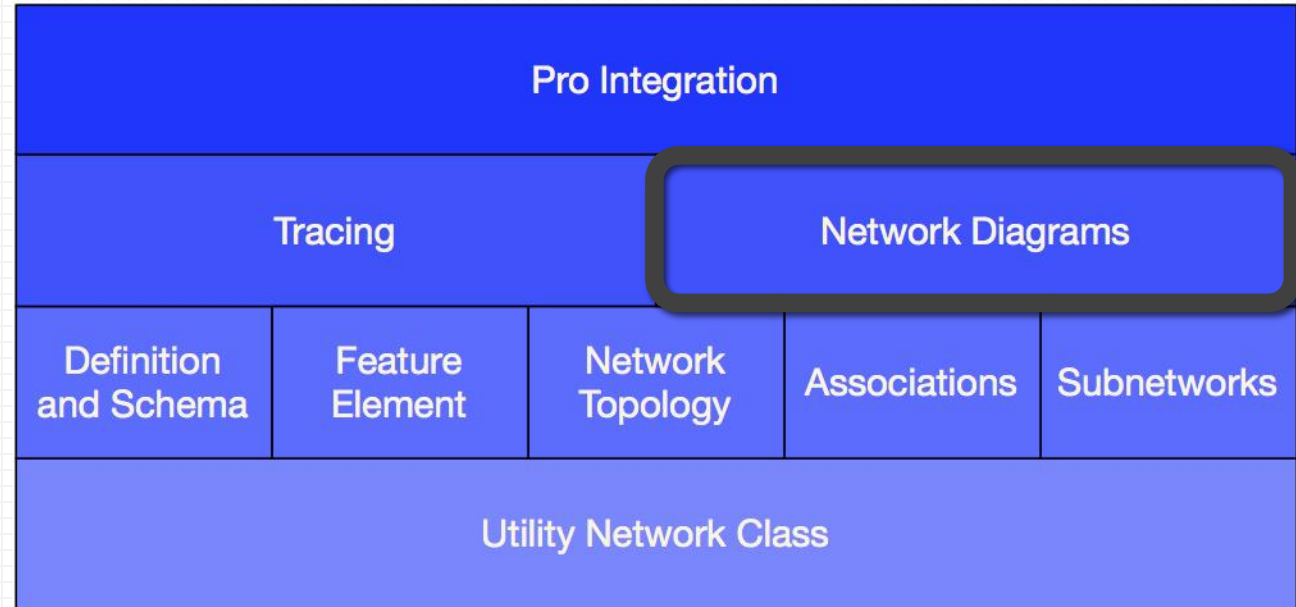
3. Organization of the Utility Network SDK

- **Tracing** provides tracing functionality



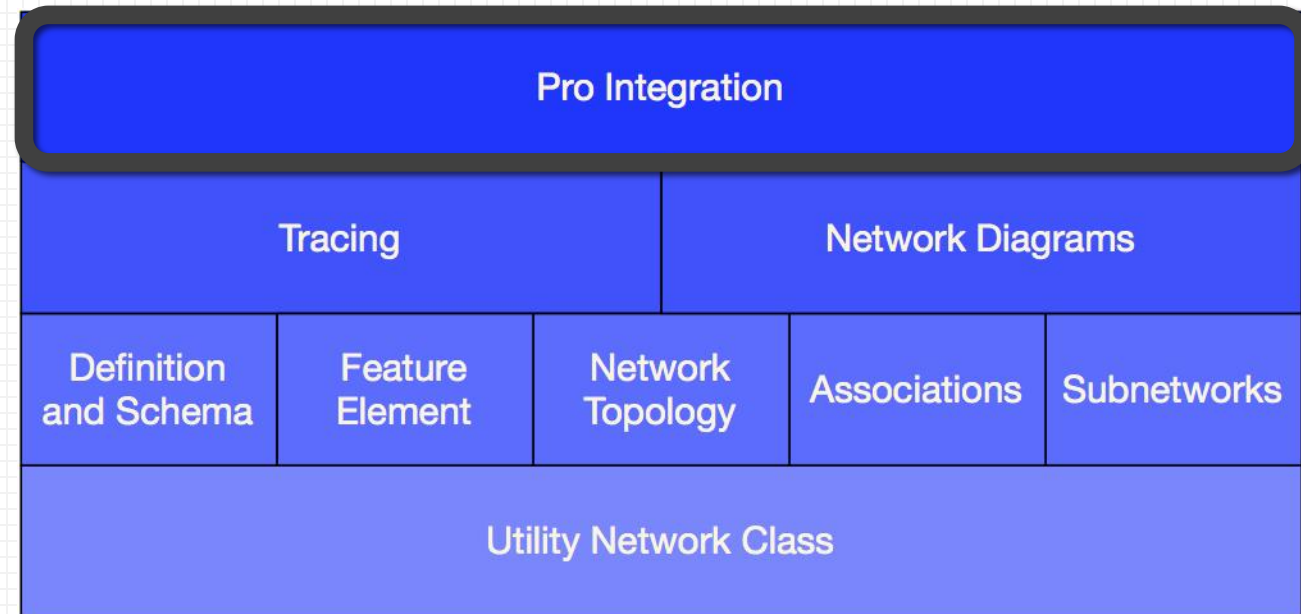
3. Organization of the Utility Network SDK

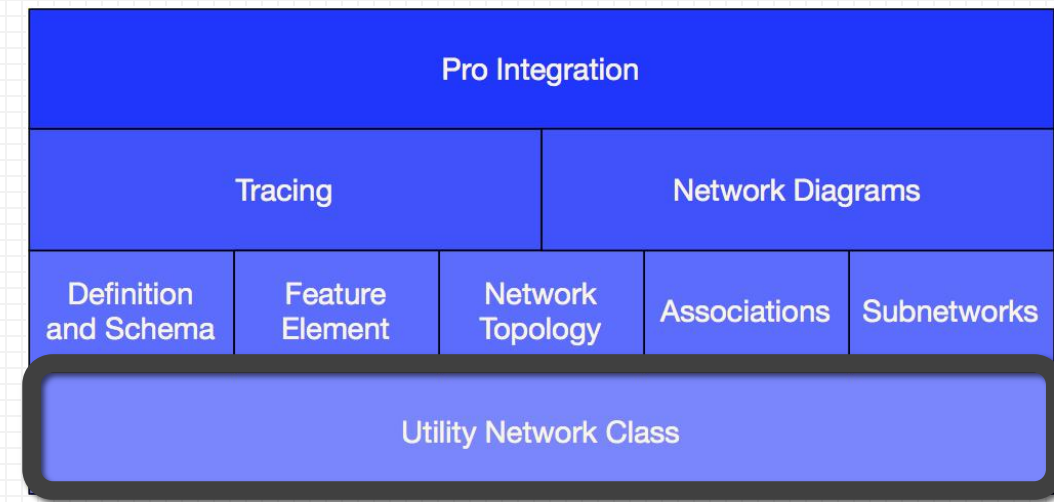
- **Network Diagrams** allows the developer to query and edit network diagrams



3. Organization of the Utility Network SDK

- Finally, **Pro Integration** describes how the utility network SDK integrates with other parts of the Pro SDK





Utility Network Class

The `UtilityNetwork` class

- Utility networks are implemented in the geodatabase as controller datasets
 - Other controller datasets in ArcGIS include Network Datasets for transportation networks, and Topology for managing coincident features
- The `UtilityNetwork` class provides an abstraction of this controller dataset
- Methods on this class provide an entry point to the other areas of the utility network SDK

Tracing			Network Diagrams	
Definition and Schema	Feature Element	Network Topology	Associations	Subnetworks

- These are documented in the corresponding sections of this document

UtilityNetwork — Dataset Routines

- While most of the methods of the UtilityNetwork class are covered in other areas of this document, these are general to all Datasets

Type : DatasetType

- Returns DatasetType.UtilityNetwork

GetExtent() : Envelope

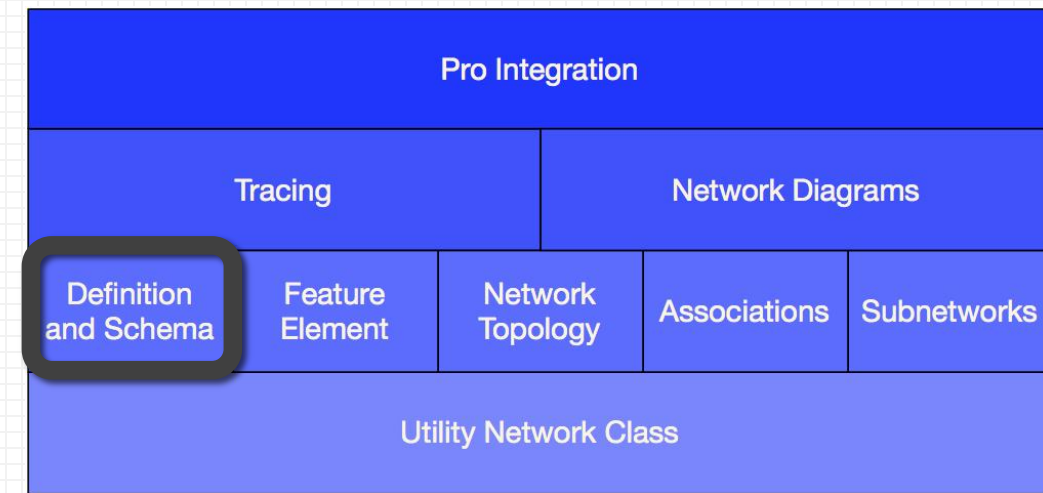
- This routine returns the extent of the feature classes within the network (same as other controller datasets in ArcGIS)

Obtaining a UtilityNetwork object

- Like other datasets, in the geodatabase, a UtilityNetwork can be obtained by calling Geodatabase.OpenDataset
 - `Geodatabase.OpenDataset<UtilityNetwork>(string datasetName) : UtilityNetwork`
- The UtilityNetwork can also be obtained from a table or feature class that belongs to a utility network
 - `Table.IsControllerDatasetSupported() : bool`
 - `Table.GetControllerDataset() : IReadOnlyList<Dataset>`
- Note that a particular feature class can belong to multiple controller datasets

Code Snippet: Getting a UtilityNetwork from a Table

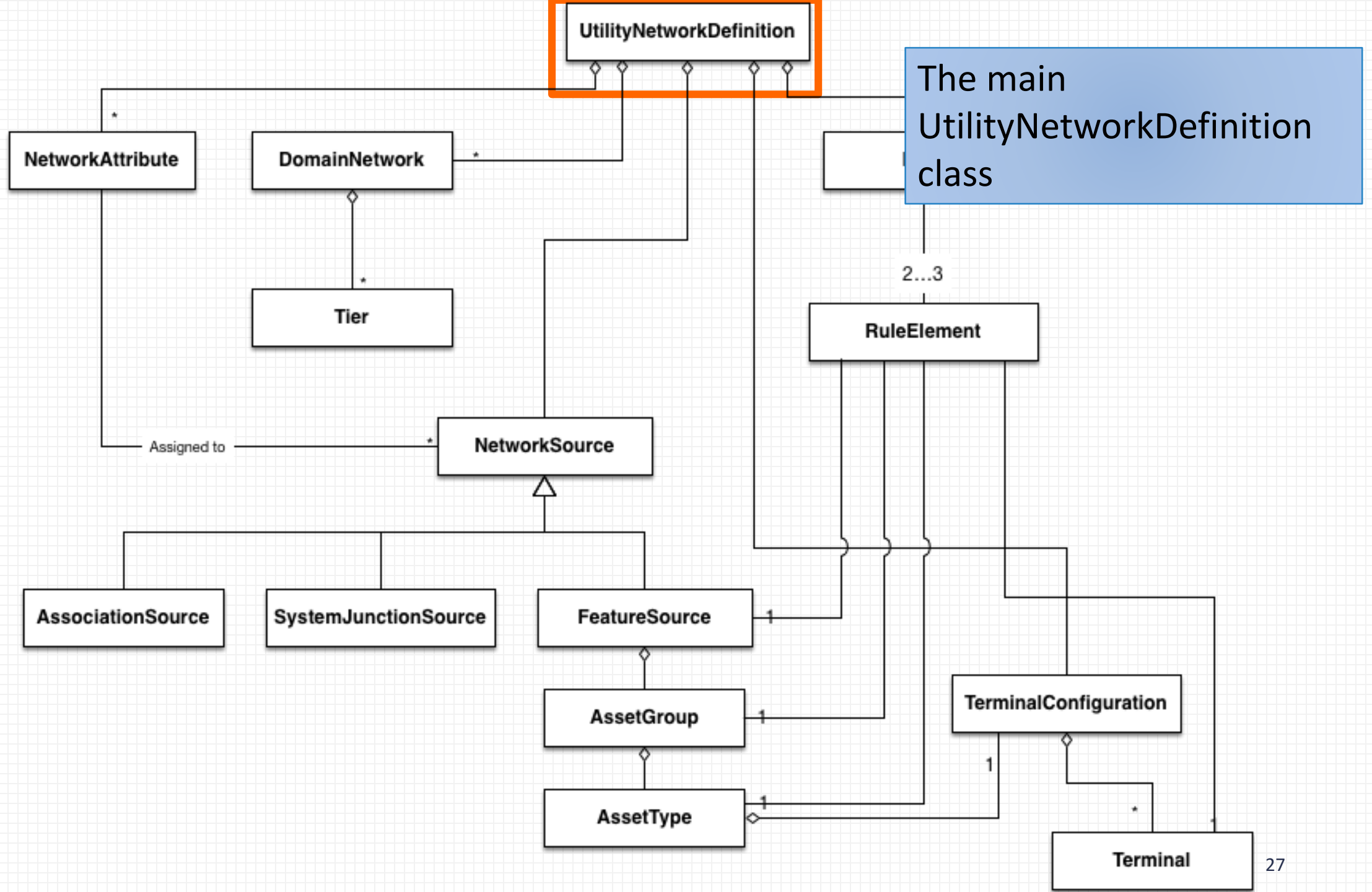
```
public static UtilityNetwork GetUtilityNetworkFromFeatureClass(FeatureClass featureClass)
{
    if (featureClass.IsControllerDatasetSupported())
    {
        IReadOnlyList<Dataset> controllerDatasets = featureClass.GetControllerDatasets();
        foreach (Dataset controllerDataset in controllerDatasets)
        {
            if (controllerDataset is UtilityNetwork)
            {
                return controllerDataset as UtilityNetwork;
            }
        }
    }
    return null;
}
```

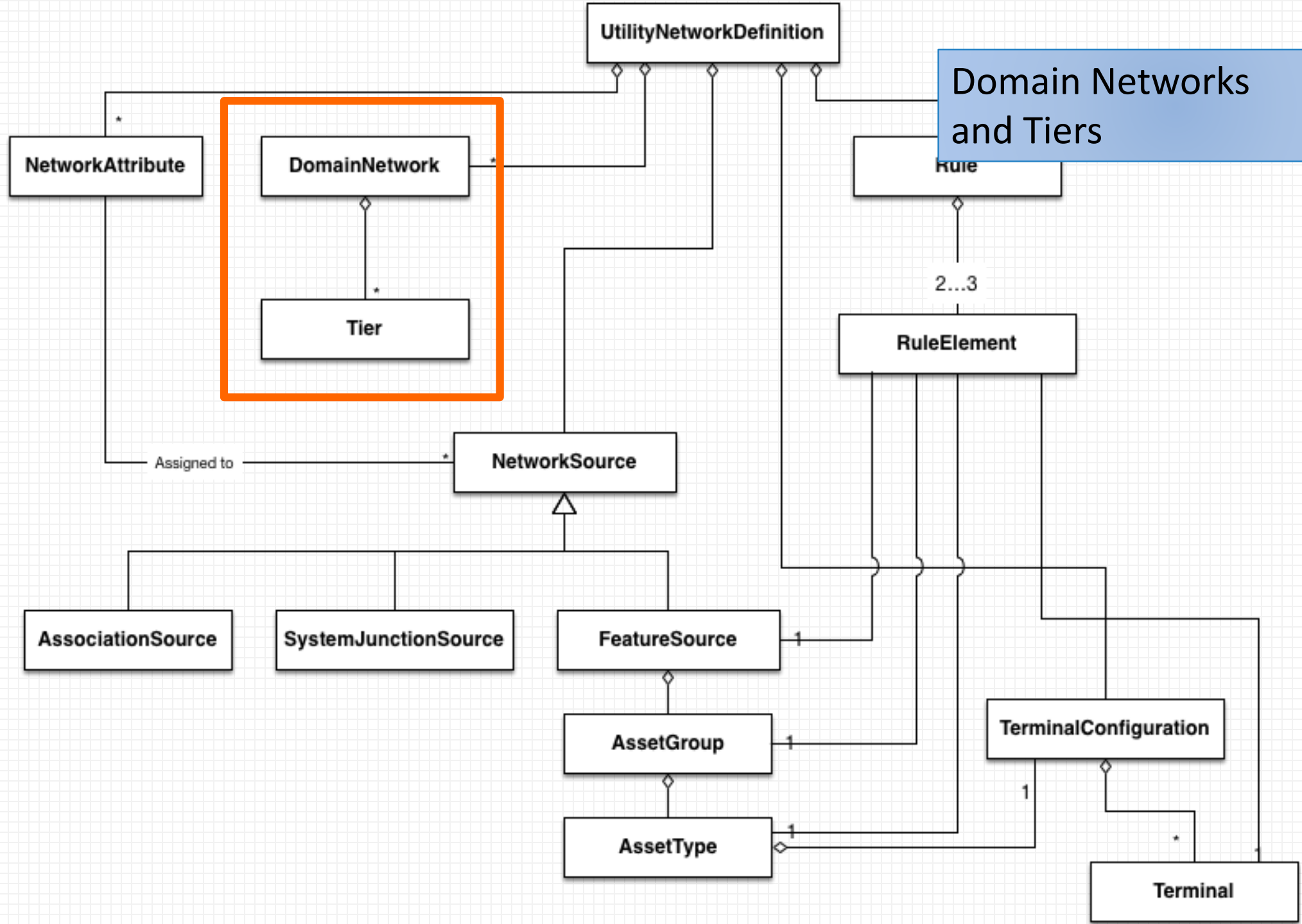



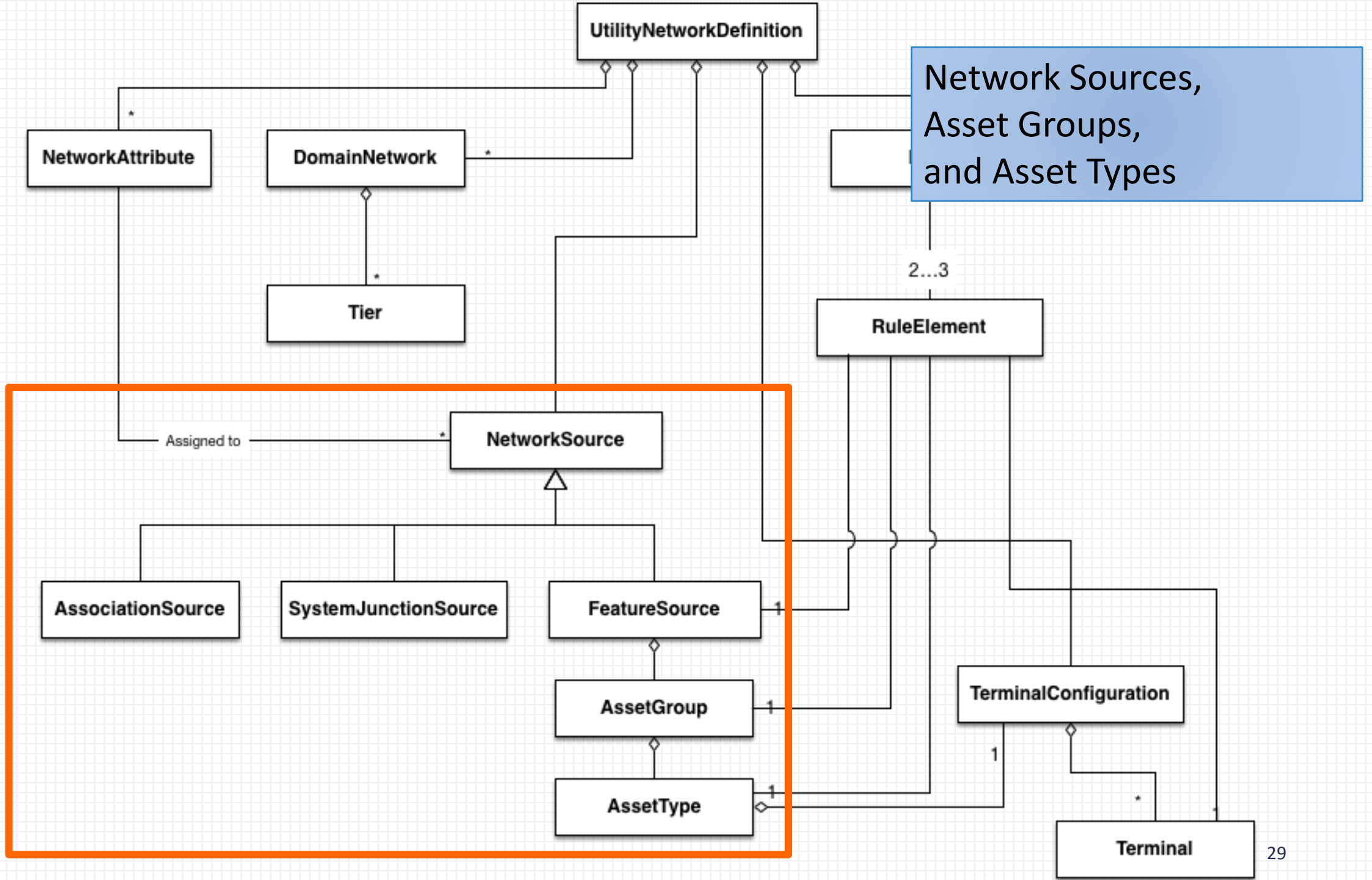
Definition and Schema

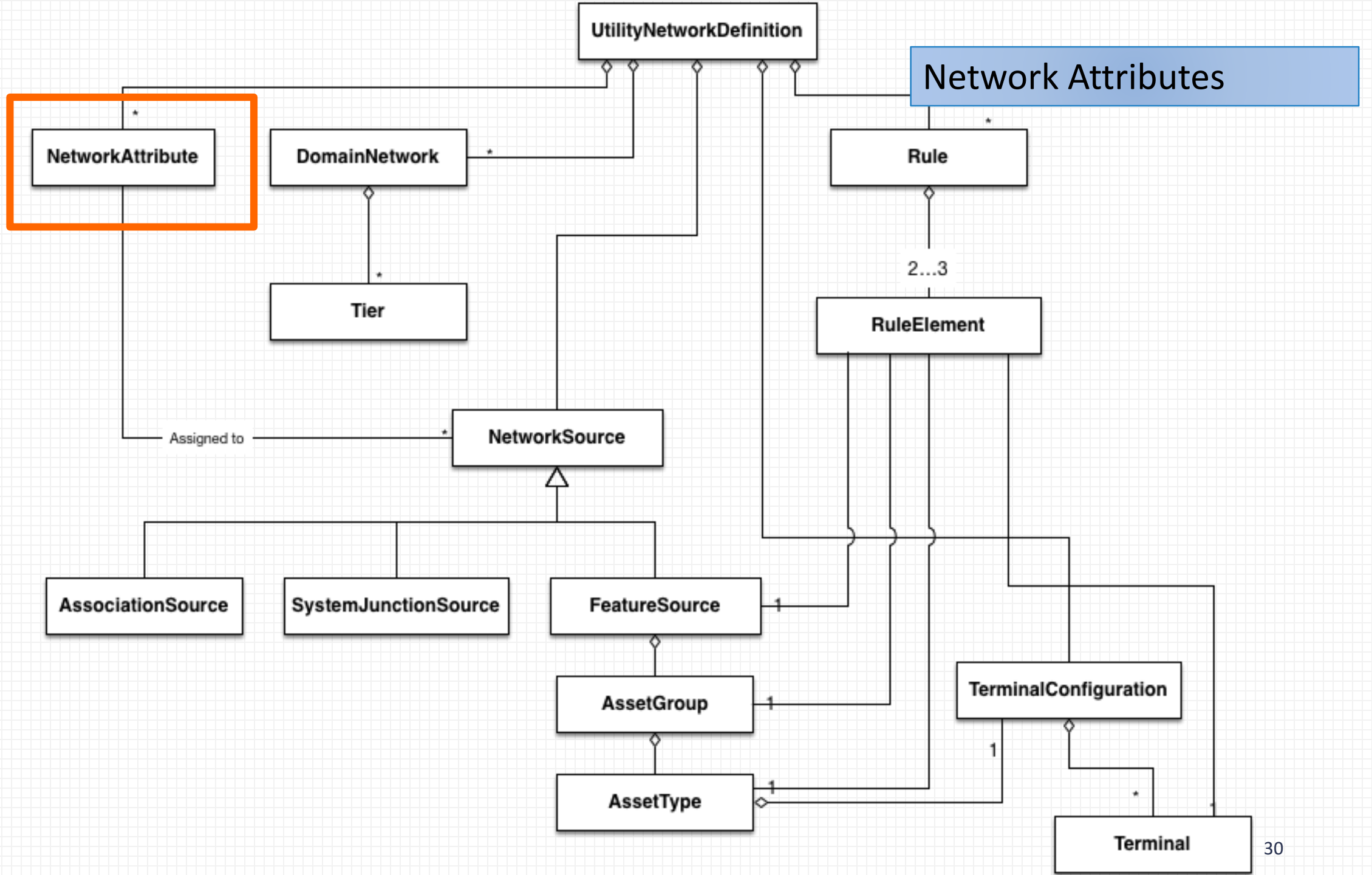
Definition and Schema

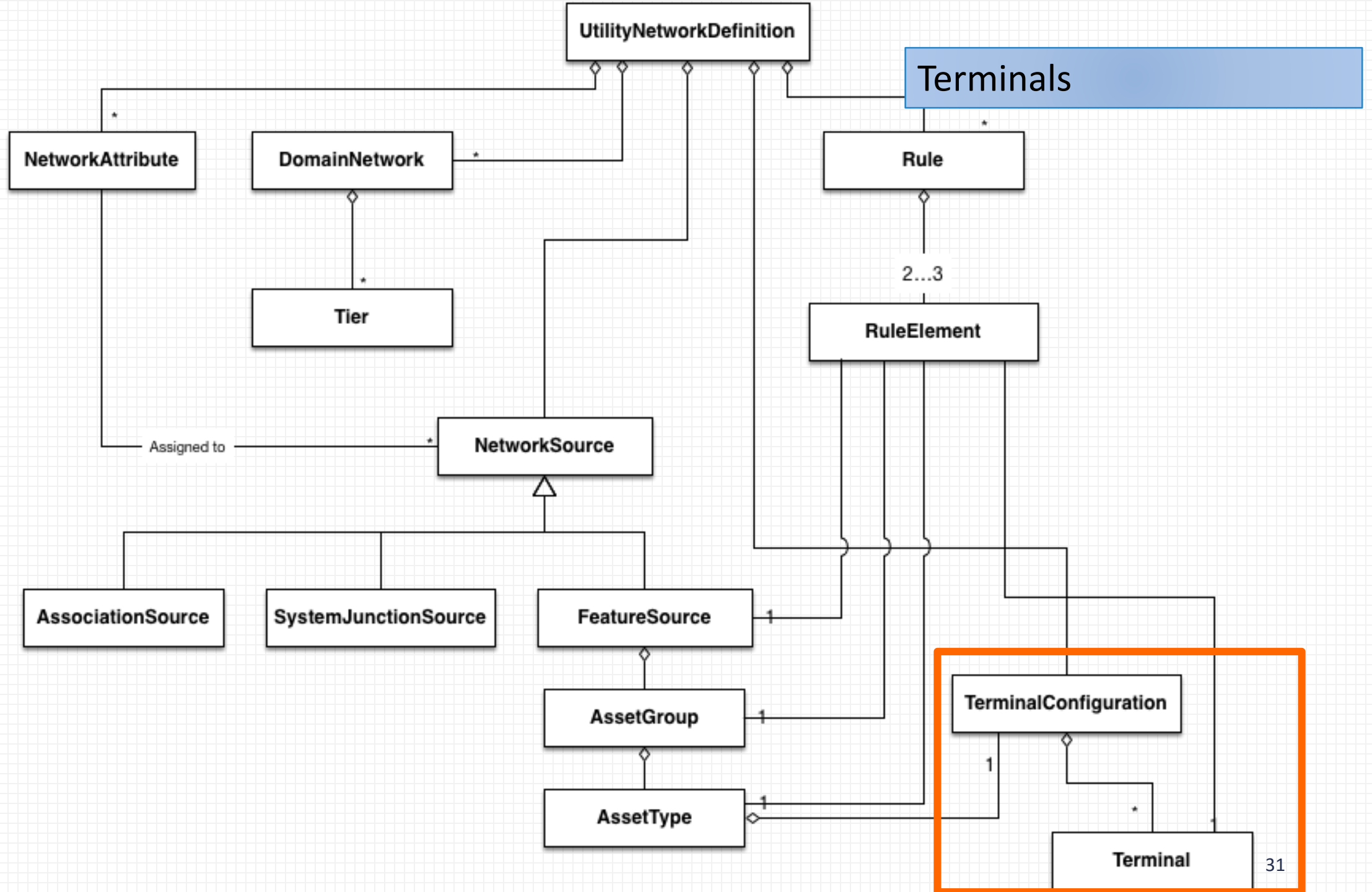
- The `UtilityNetworkDefinition` class provides metadata information about the utility network
- Opening the Definition is a lightweight operation when compared to opening a dataset
- The `UtilityNetworkDefinition` class is the entryway to a constellation of additional classes that provide metadata about elements inside the utility network, such as domain networks and rules
- The classes described in this section are all value-based objects that are derived from information cached with the feature service
- These additional classes are grouped as shown on the following slides
- For more information on the concepts implemented by these objects, see the [online help](#)

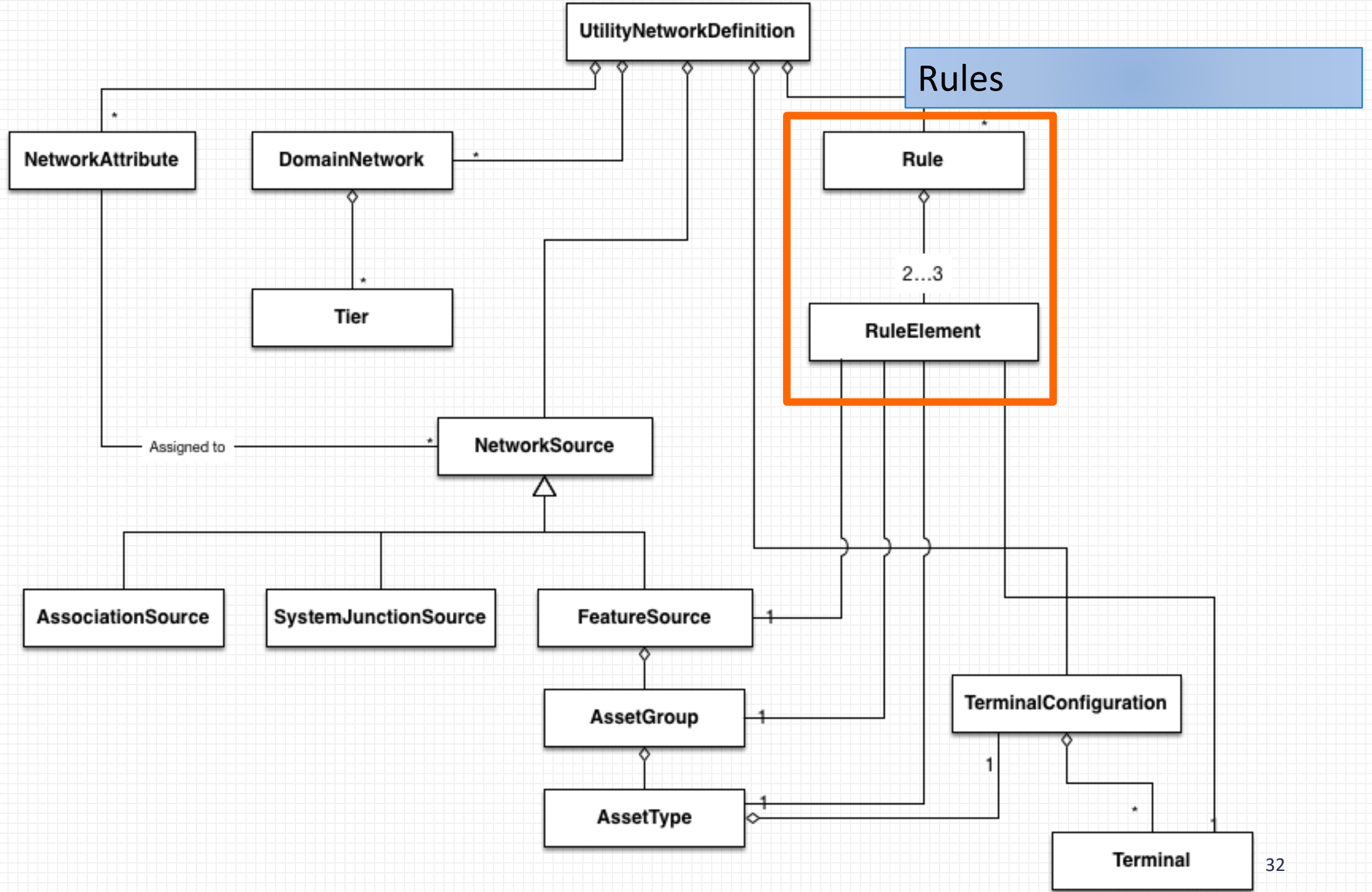












Accessing UtilityNetworkDefinition

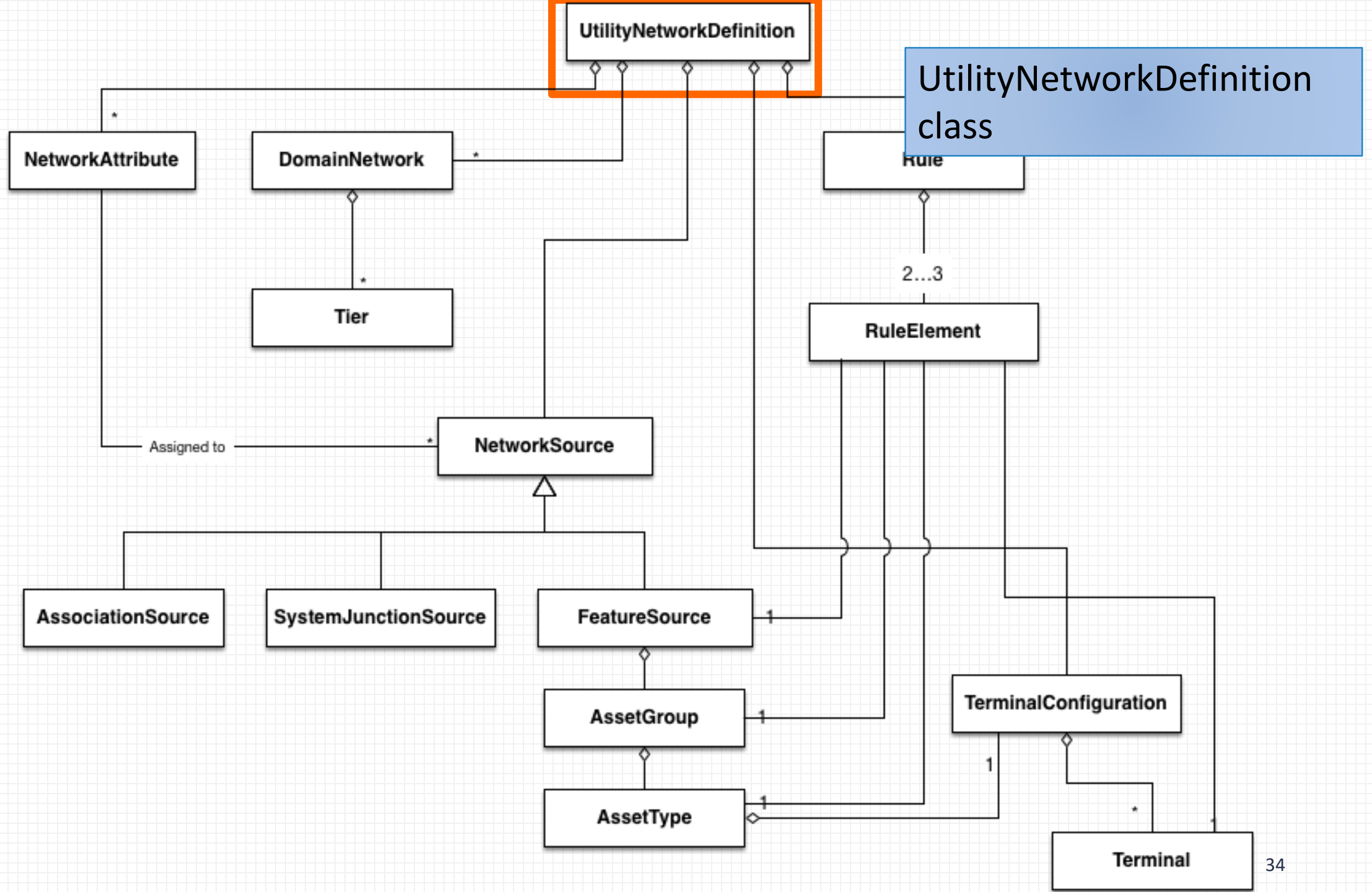
- There are two options for accessing `UtilityNetworkDefinition` classes.
- Open a `UtilityNetworkDefinition` from a `Geodatabase` — typically this is used when it is not anticipated that the utility network will be opened.

```
UtilityNetworkDefinition definition = geodatabase.GetDefinition<UtilityNetworkDefinition>("UtilityNetworkName");
```

- Open the `UtilityNetworkDefinition` directly from the `UtilityNetwork` — this is used when the utility network is already open

```
UtilityNetworkDefinition definition = utilityNetwork.GetDefinition();
```


- This is the same access pattern used by other Definition classes within the Geodatabase API












UtilityNetworkDefinition

UtilityNetworkDefinition
Sealed Class
→ Definition

Properties

 `DatasetType { get; } : DatasetType`

Methods

-  `GetAvailableCategories() : IReadOnlyList<string>`
-  `GetDomainNetworks() : IReadOnlyList<DomainNetwork>`
-  `GetNetworkAttribute(string networkAttributeName) : NetworkAttribute`
-  `GetNetworkAttributes() : IReadOnlyList<NetworkAttribute>`
-  `GetNetworkSource(string networkSourceName) : NetworkSource`
-  `GetNetworkSources() : IReadOnlyList<NetworkSource>`
-  `GetRules() : IReadOnlyList<Rule>`
-  `GetServiceTerritoryEnvelope() : Envelope`
-  `GetTerminalConfigurations() : IReadOnlyList<TerminalConfiguration>`

UtilityNetworkDefinition: General Routines

■ `DatasetType` : `DatasetType`

- This is identical to the value returned by `UtilityNetwork.Type`. Both properties are included to conform with the convention established by other dataset types

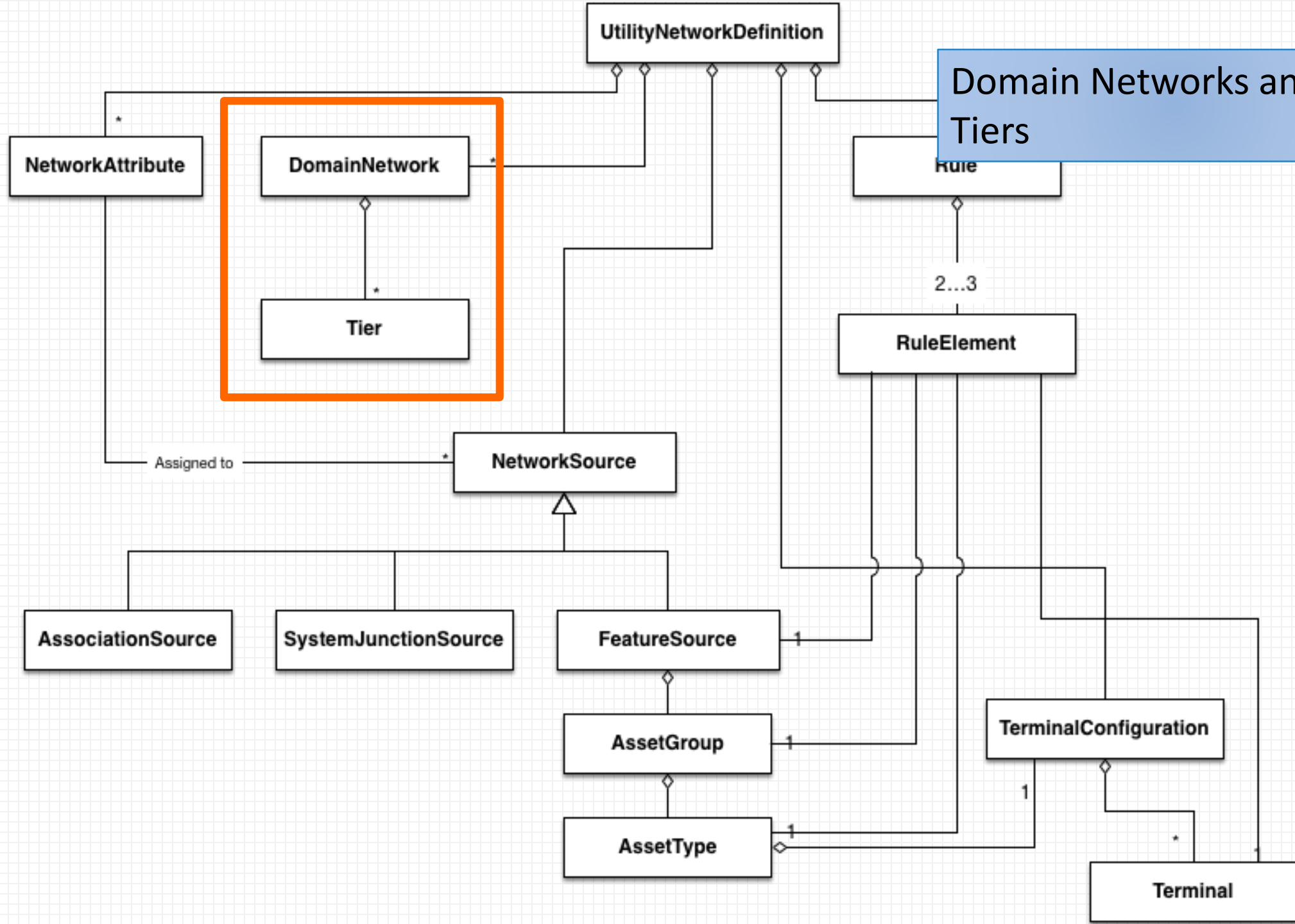
`GetServiceTerritoryEnvelope()` : `Envelope`

- Returns the extent of the utility network (i.e., the extent of the service territory + 10%).
- Note that this can be slightly different than the value returned by `UtilityNetwork.GetExtent()`

`GetAvailableCategories()` : `ReadOnlyList<string>`

- Returns a list of all the categories that have been registered with the utility network.

Domain Networks and Tiers

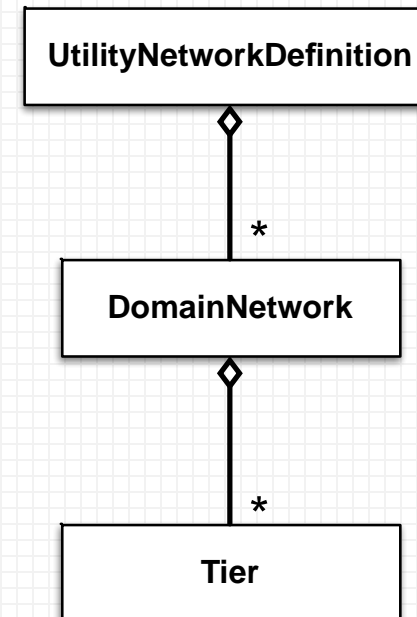


Domain Networks

- A domain network represents an industry domain
- Some examples include electric distribution, electric transmission, gas, and water

`UtilityNetworkDefinition.GetDomainNetworks() : IReadOnlyList<DomainNetwork>`

- Returns a list of domain networks included with this utility network



Domain Network

Alias : string

- A user-readable name

ID : int

- An identifier for the domain network. This identifier is unique within a single utility network

Name : string

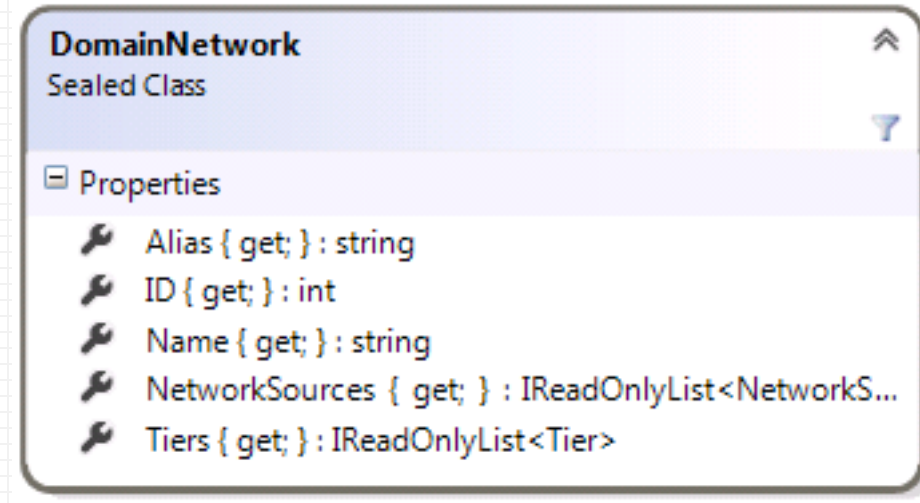
- The name is prepended to the feature classes in the domain network

NetworkSources : IReadOnlyList<NetworkSource>

- The list of network sources for this domain network.

Tiers : IReadOnlyList<Tier>

- Returns the tiers in this domain network.



Tiers - 1

- Tiers demarcate a logical level within the network
 - E.g., in an electric distribution network, there may be Subtransmission, MediumVoltage, and LowVoltage tiers
 - Some networks may only have one tier, such as electric transmission

IsSourceNameRequired : bool

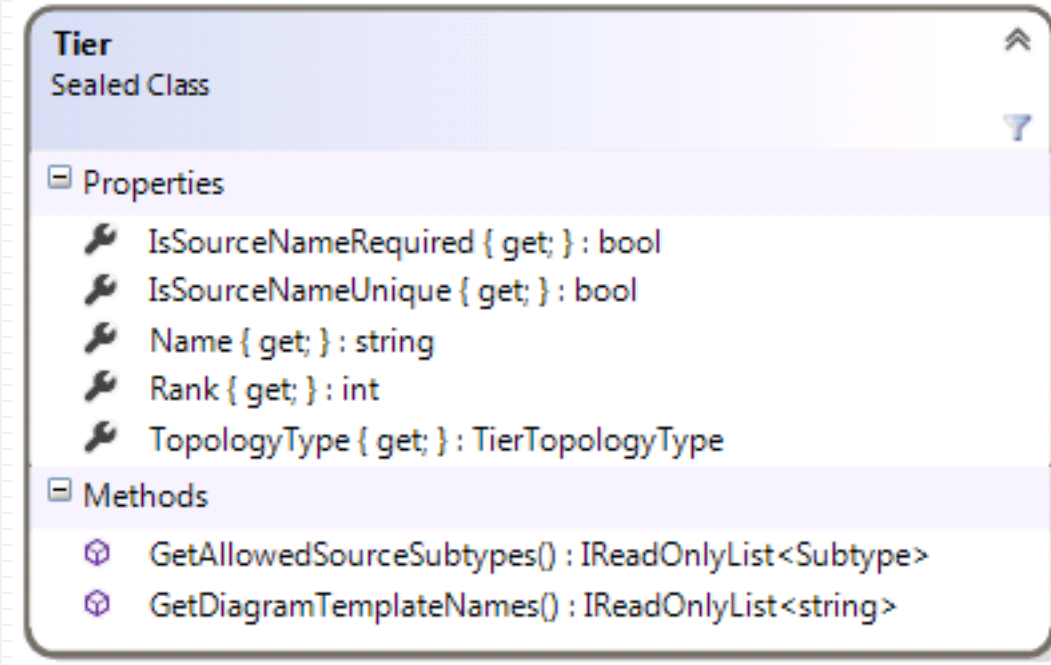
- Determines whether subnetwork sources within this tier require a name

IsSourceNameUnique : bool

- Determines whether subnetwork sources within this tier require unique names

GetAllowedSourceSubtypes() : IReadOnlyList<Subtype>

- Returns a list of subtypes that can be subnetwork sources within this tier
- This routine will be changed in a future alpha release



Tiers - 2

`GetDiagramTemplateNameNames() :`
`IReadOnlyList<string>`

- Returns the default diagram templates for this tier

`Name : string`

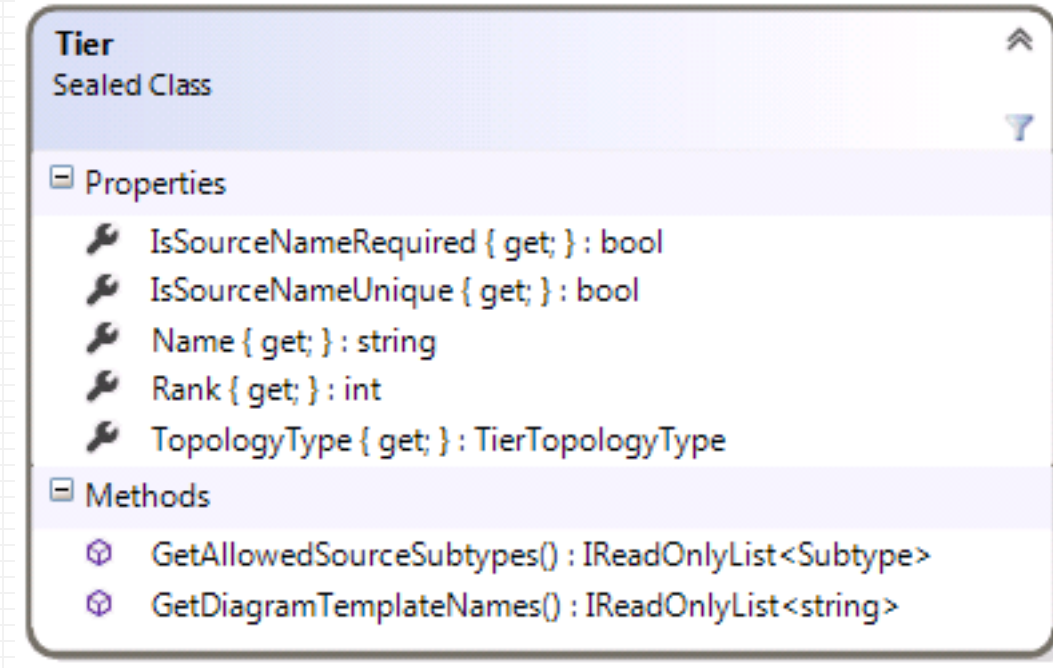
- The name of the tier (e.g., “Medium voltage”)

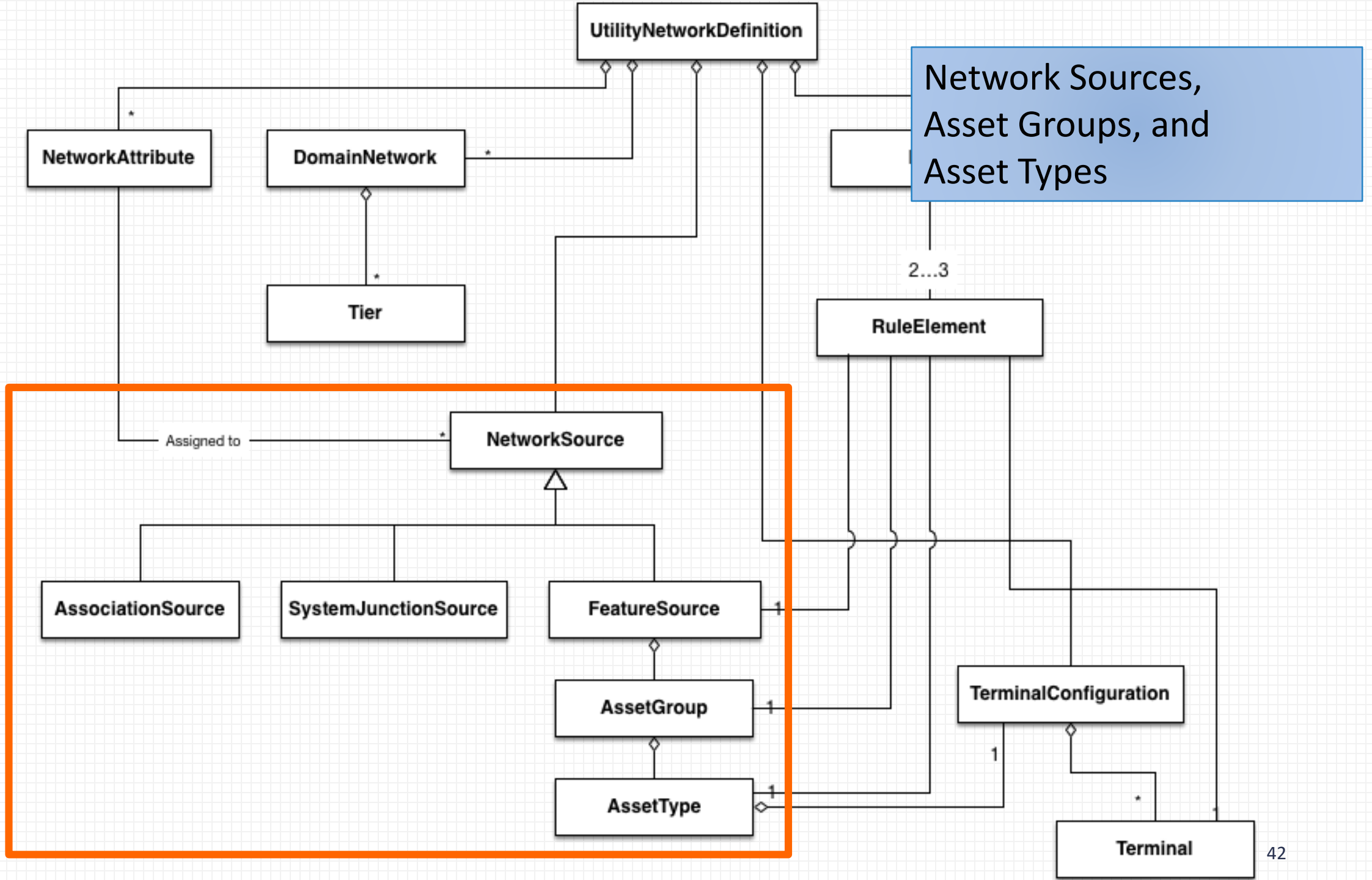
`Rank : int`

- The numeric rank of the tier.

`TopologyType : TierTopologyType`

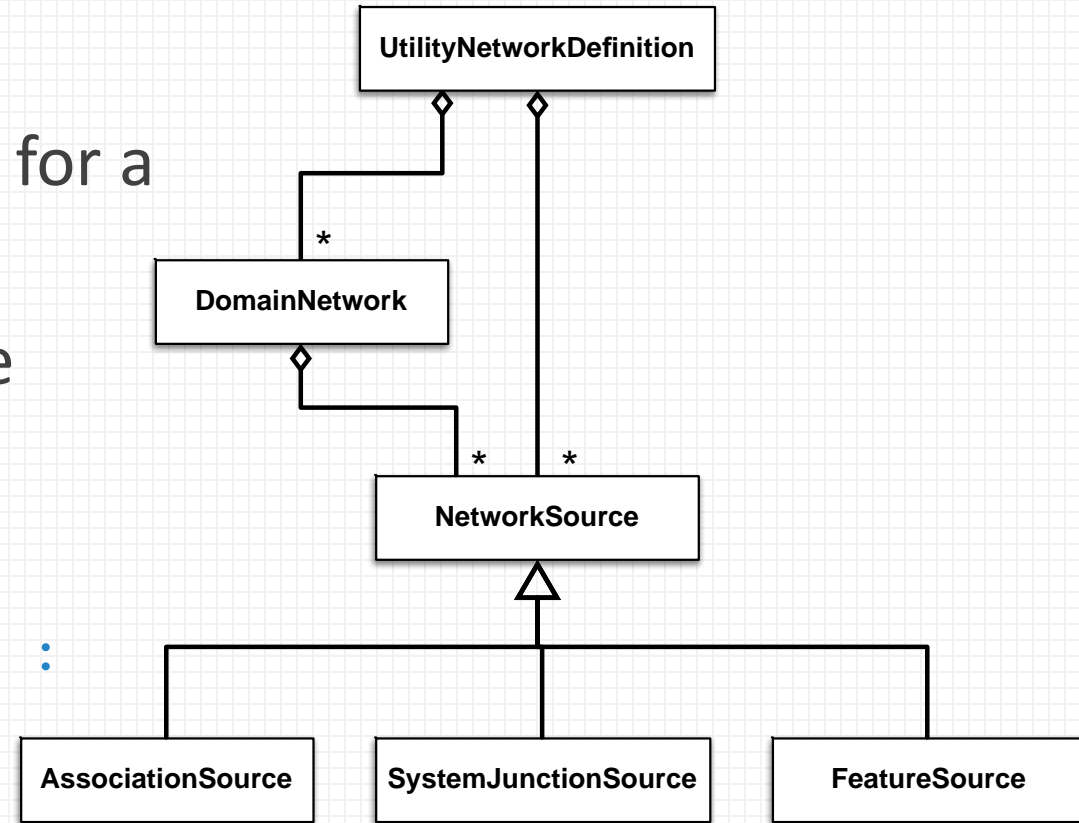
- The type of subnetworks that are supported
- TierTopologyType is an enum with the values `Radial`, `MultifeedRadial` and `Mesh`





Network Sources

- A Network Source is a source of information for a utility network
- Network Sources can be obtained with three different routines:
 - `UtilityNetworkDefinition.GetNetworkSource(string networkSourceName) : NetworkSource`
 - `UtilityNetworkDefinition.GetNetworkSources() : IReadOnlyList<NetworkSource>`
 - `DomainNetwork.NetworkSources : IReadOnlyList<NetworkSource>`
- The most commonly used are **FeatureSource** objects, which correspond to Feature Classes within the utility network



NetworkSource

- NetworkSource is the abstract base class for the Network Source class family

ID : int

- The numeric ID of the network source

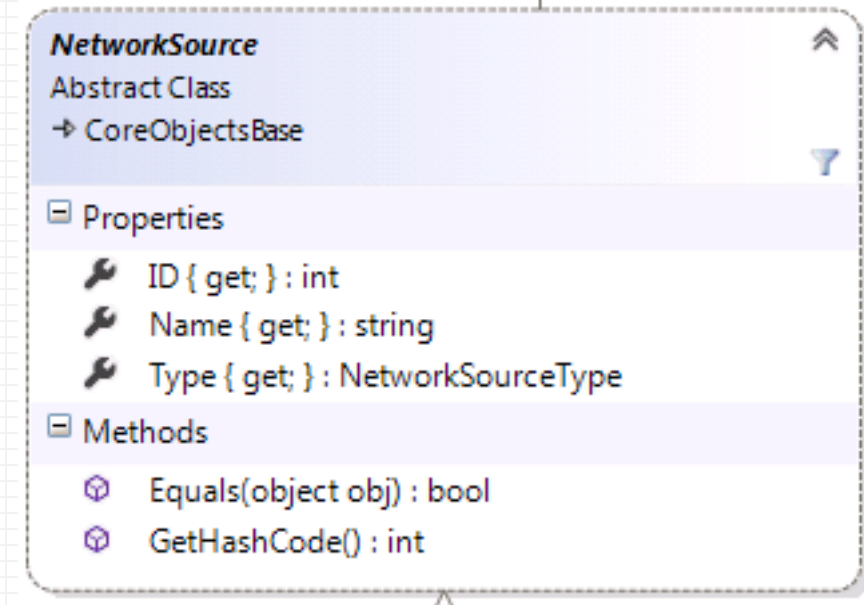
Name : string

- The name of the network source. This is the name of the underlying table, not a user-readable name.

Type : NetworkSourceType

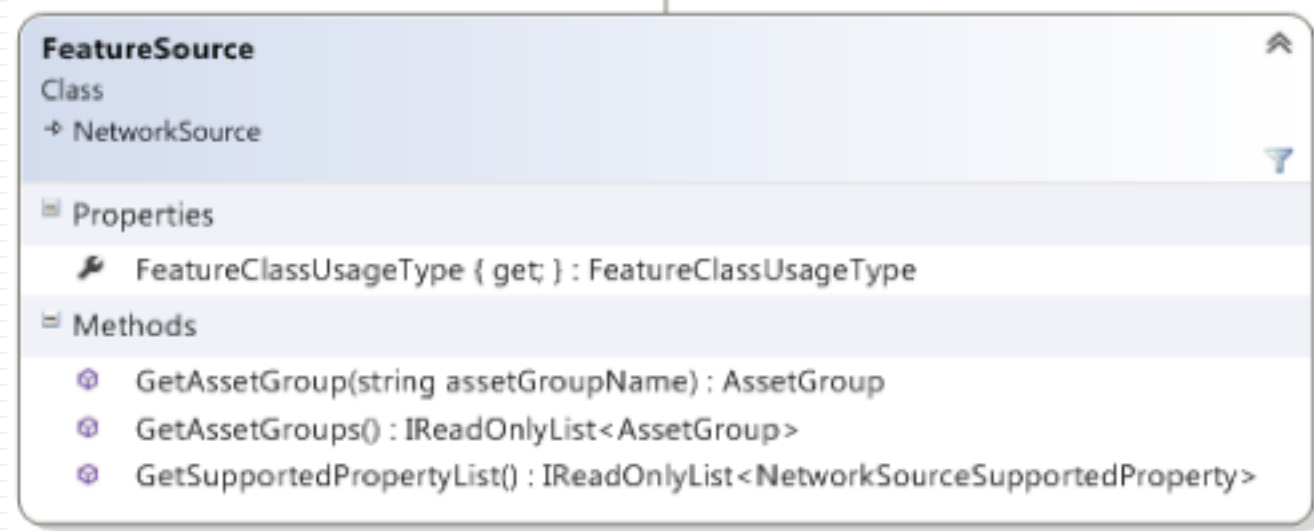
- The values for the NetworkSourceType enum are as follows: [SystemJunction](#), [Boundary](#), [Junction](#), [Edge](#), [Association](#)

- NetworkSource objects can be compared with the comparison operator



FeatureSource

- FeatureSource represents a source that is created from a user feature class



FeatureClassUsageType : FeatureClassUsageType

- This property returns how the feature class is used.
- Possible values are **Device**, **Junction**, **Line**, **Assembly**, **SubnetLine**, **StructureJunction**, **StructureLine**, and **StructureBoundary**.

GetAssetGroup(string assetGroupName) : AssetGroup

GetAssetGroups() : IReadOnlyList<AssetGroup>

- Returns one or more AssetGroups (subtypes) for this particular feature class

GetSupportedPropertyList() : IReadOnlyList<NetworkSourceSupportedProperty>

- Returns the properties that this network source supports. Possible values are **Containment**, **StructuralAttachments**, **Categories**, **Terminals**, **NetworkAttributes**.

System Sources



- **SystemJunctionSource**

- Concrete class that represents system junctions (marker interface)

- **AssociationSource**

- Concrete class that represents associations (marker interface)

AssetGroup

- The AssetGroup class provides information about Asset Groups (subtypes) within the utility network. In the core geodatabase, they are implemented as subtypes.

Code : int

- The subtype code

FeatureSource : FeatureSource

- The parent FeatureSource

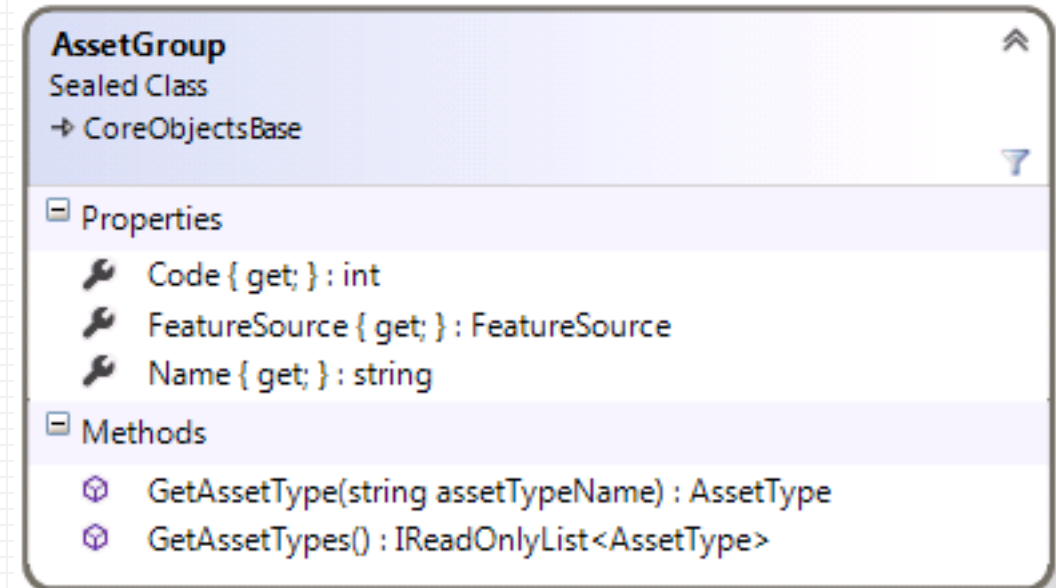
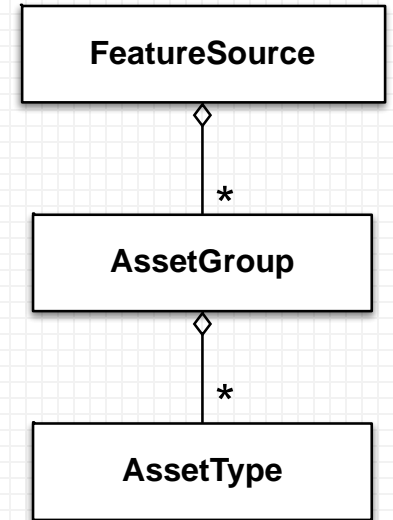
Name : string

- The subtype name

GetAssetType(string assetTypeName) : AssetType

GetAssetTypes() : IReadOnlyList<AssetType>

- Returns Asset Type definitions



AssetType - 1

- Provides information about the definition of an AssetType

AssetGroup : AssetGroup

- The parent AssetGroup

Code : int

- The code of the AssetType domain

Name: string

- The string value of the AssetType domain

IsLinearConnectivityPolicySupported() : bool

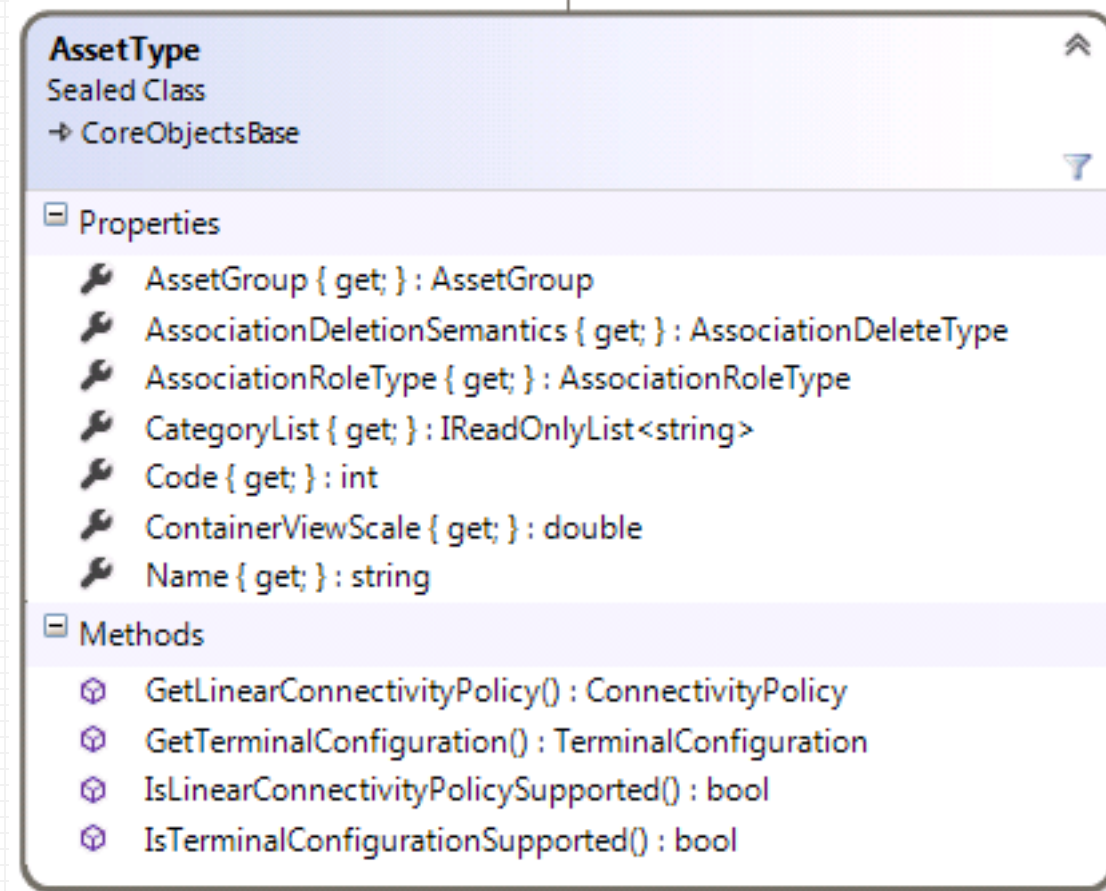
- Returns whether or not the **GetConnectivityPolicy()** routine is applicable.

GetLinearConnectivityPolicy() : ConnectivityPolicy

- Returns whether connectivity for this asset type can be established at any vertex (**AnyVertex**) or only at end points (**EndVertex**)

AssociationDeletionSemantics : AssociationDeleteType

- The deletion type for this asset class (**Cascade, None, Restricted**)



AssetType - 2

ContainerViewScale : double

- Returns the default scale of any containers created from this asset type.
- If the asset type is not a container, this routine returns 0.0.

AssociationRoleType : AssociationRoleType

- Returns whether the asset type can be a **Container**, **Structure**, or neither (**None**).

GetCategoryList() : IReadOnlyList<string>

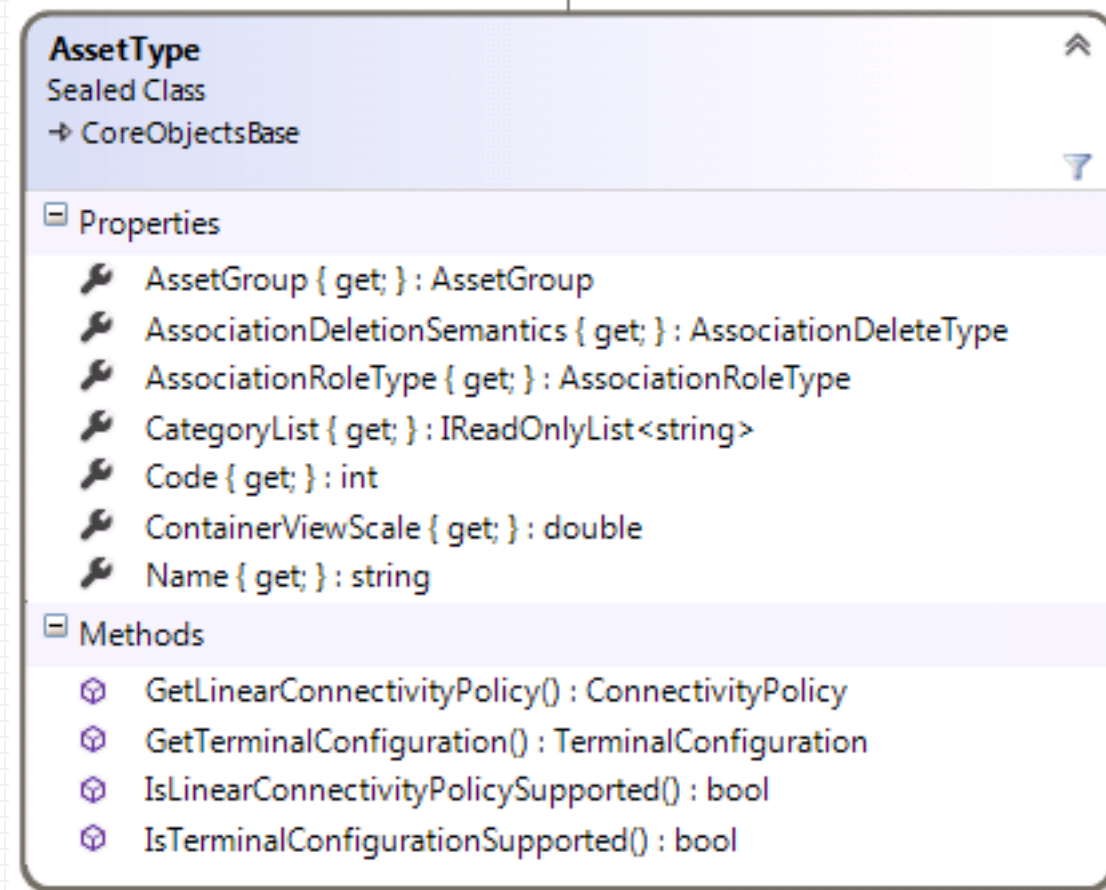
- Returns the categories supported

IsTerminalConfigurationSupported() : bool

- Returns whether or not the **GetTerminalConfiguration()** routine is applicable

GetTerminalConfiguration() : TerminalConfiguration

- Returns the **TerminalConfiguration** for this asset type



The screenshot shows the definition of the **AssetType** class, which is a sealed class inheriting from **CoreObjectsBase**. It lists several properties and methods.

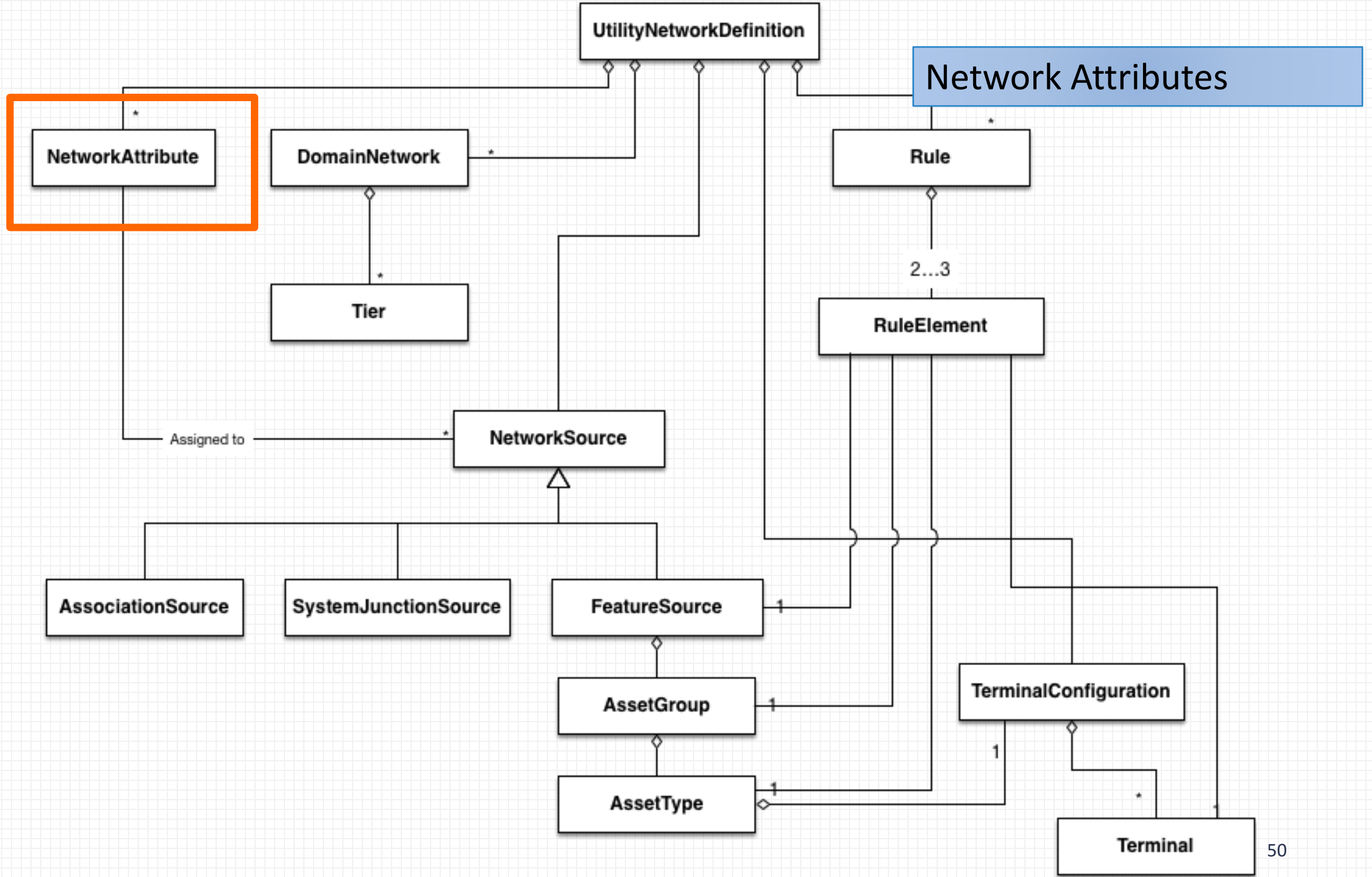
AssetType
Sealed Class
→ CoreObjectsBase

Properties

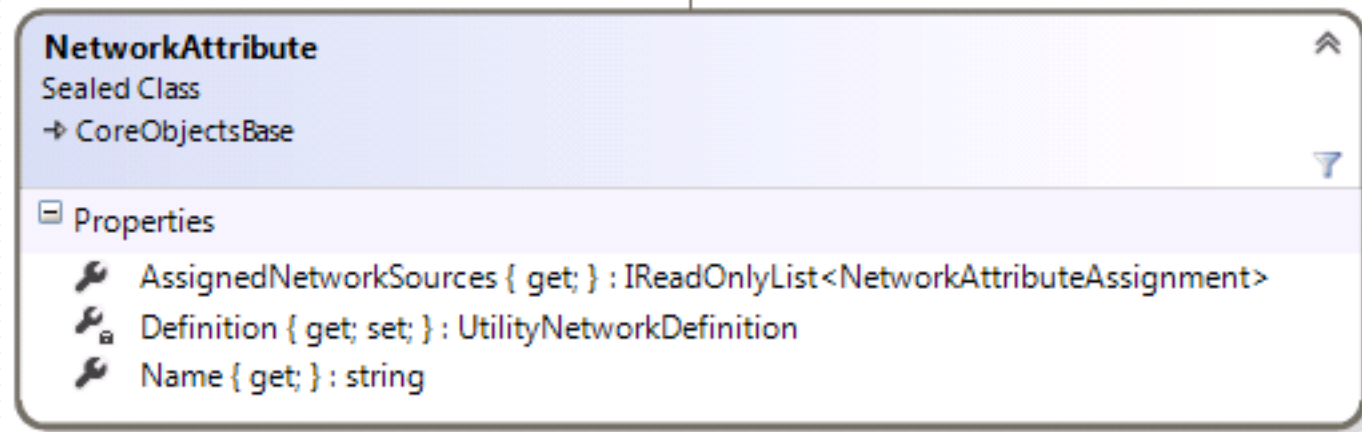
- AssetGroup { get; } : AssetGroup
- AssociationDeletionSemantics { get; } : AssociationDeleteType
- AssociationRoleType { get; } : AssociationRoleType
- CategoryList { get; } : IReadOnlyList<string>
- Code { get; } : int
- ContainerViewScale { get; } : double
- Name { get; } : string

Methods

- GetLinearConnectivityPolicy() : ConnectivityPolicy
- GetTerminalConfiguration() : TerminalConfiguration
- IsLinearConnectivityPolicySupported() : bool
- IsTerminalConfigurationSupported() : bool



Network Attributes



- The `NetworkAttribute` class represents a network attribute (logically equivalent to a weight in the geometric network)
- NetworkAttributes can be obtained by calling `UtilityNetworkDefinition.GetNetworkAttributes()` : `ReadOnlyList<NetworkAttribute>`

`UtilityNetworkDefinition.GetNetworkAttribute(string networkAttributeName):`
`NetworkAttribute`

`AssignedNetworkSources : IReadOnlyList<NetworkSourceAssignment>`

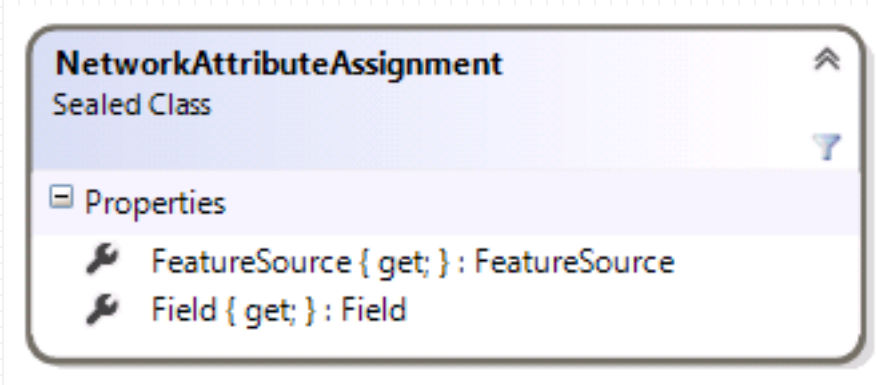
A list of how this network attribute is assigned to feature sources.

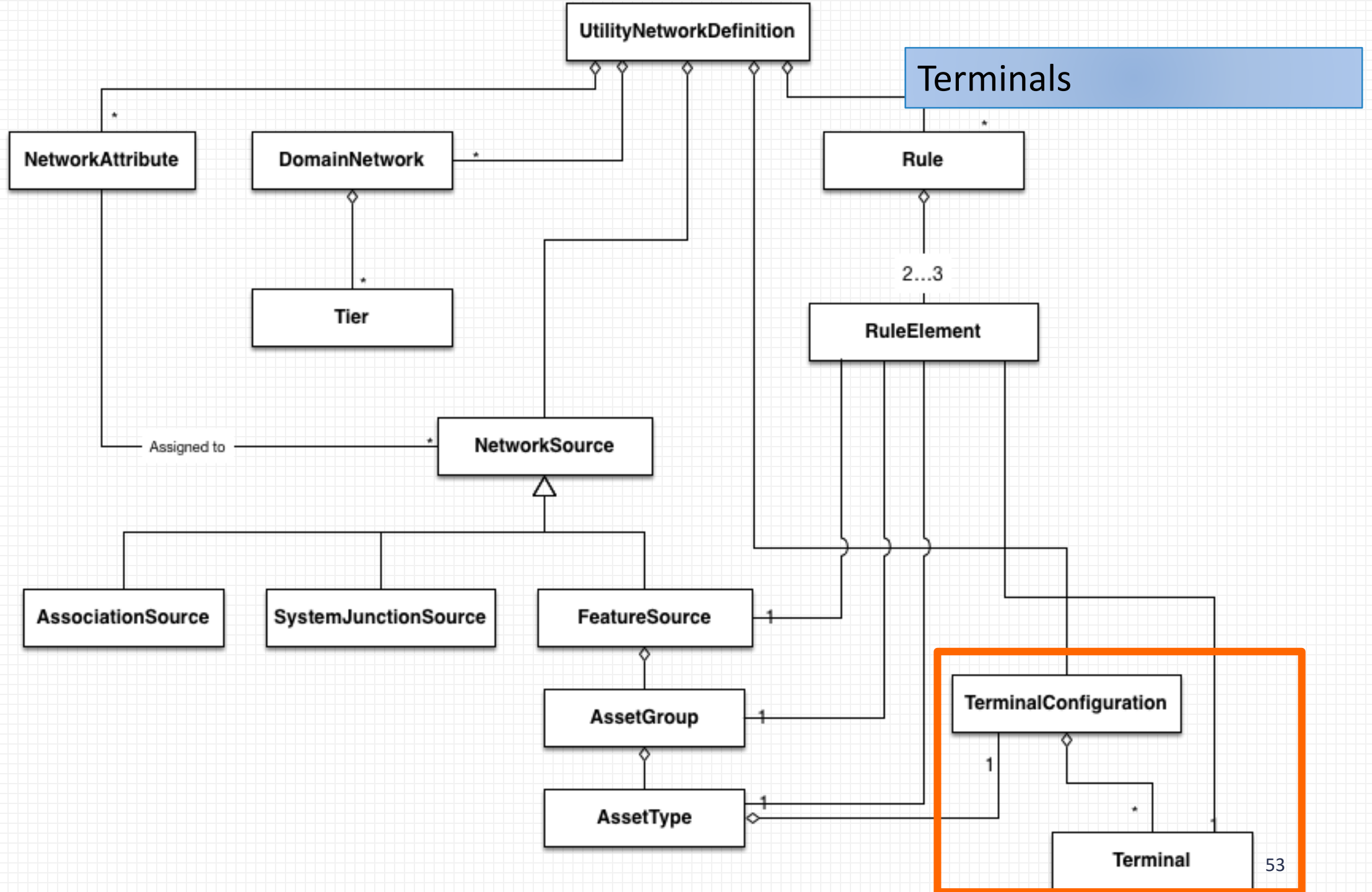
`Name : string`

The name of the network attribute

Network Attribute Assignment

- The `NetworkAttributeAssignment` class describes how a `NetworkAttribute` is assigned to a specific `FeatureSource` (feature class).
- If the same `NetworkAttribute` is assigned to multiple feature classes, `NetworkAttribute.AssignedNetworkSources` will return one object for each of these assignments.
- `FeatureSource : FeatureSource`
- `Field : Field`





Terminal Configurations

- This class will be redesigned in a future alpha release
- Terminal configurations represent a configuration of terminals associated with 0 or more subtypes.
- Terminal configurations can be obtained through two different methods:
 - `UtilityNetworkDefinition.GetTerminalConfigurations() : IReadOnlyList<TerminalConfiguration>`
 - `JunctionSource.GetTerminalConfiguration(Subtype subtype) : TerminalConfiguration`

`ConnectedTerminalCount : int`

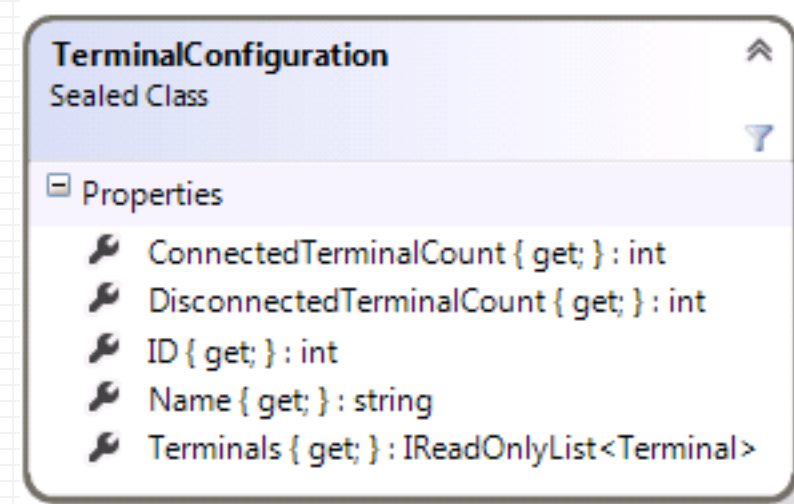
`DisconnectedTerminalCount : int`

`ID : int`

`Name : string`

- A user-readable string that describes the terminal configuration

`Terminals : IReadOnlyList<Terminal>`



Terminals

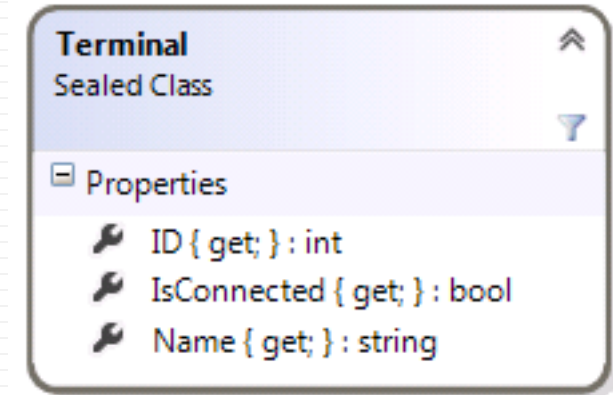
- This class will be redesigned in a future alpha release
- Terminals represent a single terminal on a junction feature

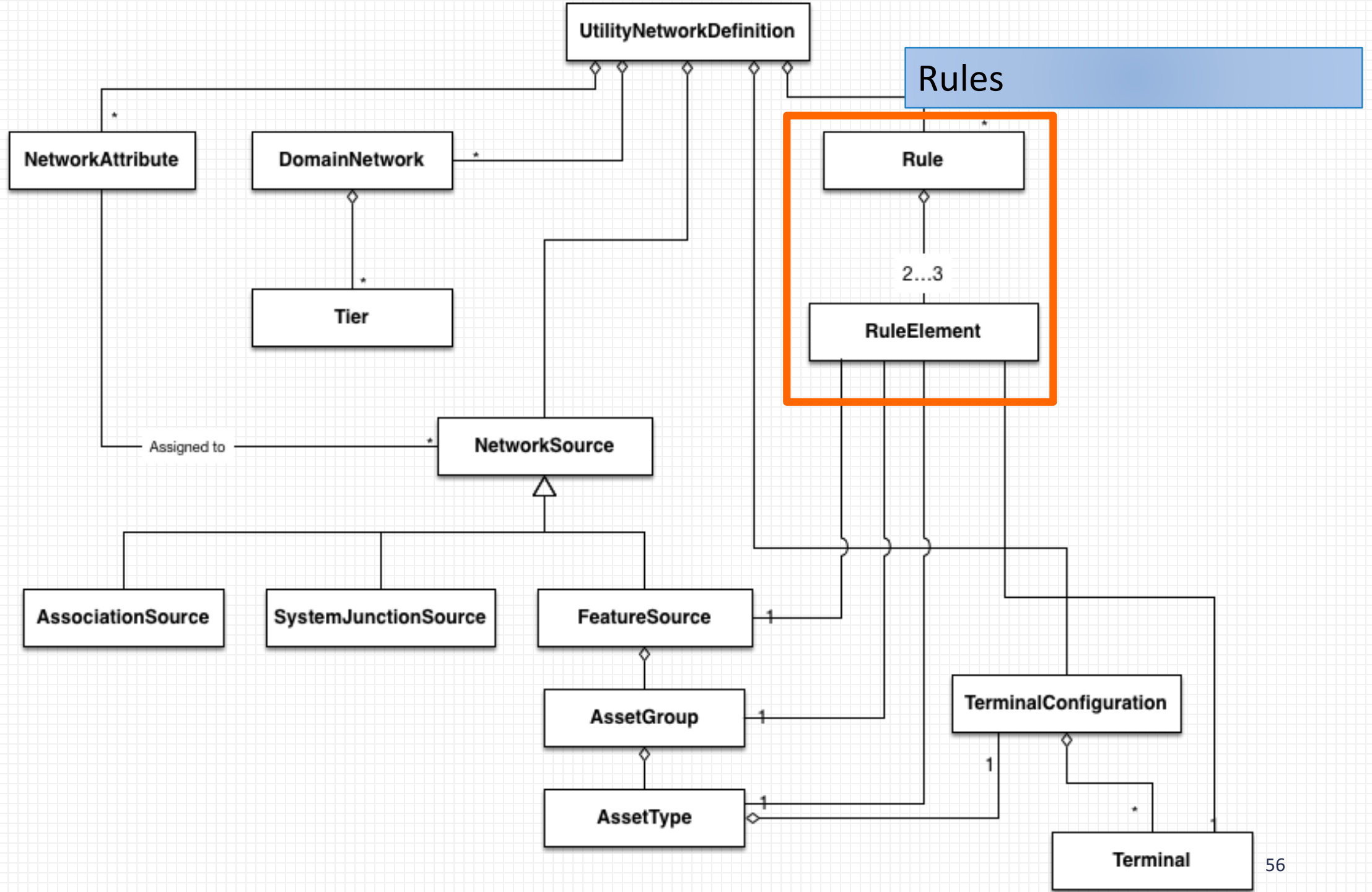
ID : int

IsConnected : bool

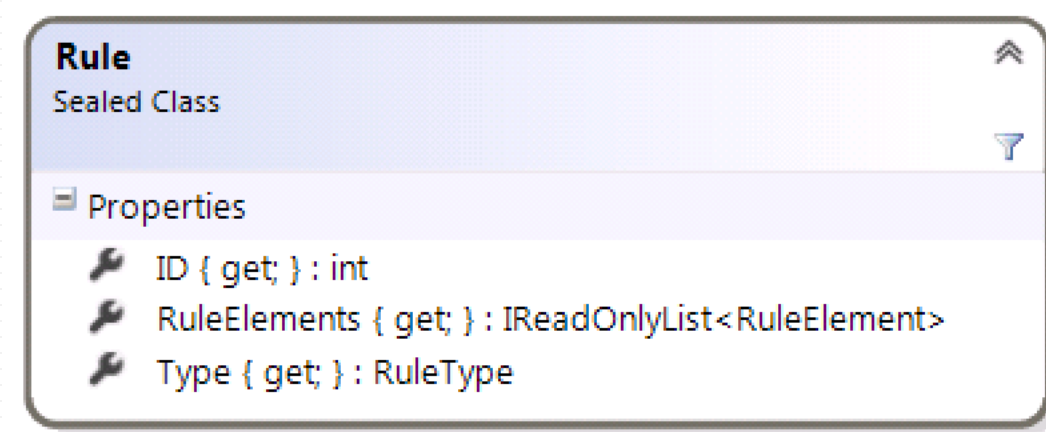
Name : string

- A user-readable string





Rule



- Describes a rule in the utility network. These define how features in a utility network can be associated with each other through connectivity, containment, and attachment
- Rule objects are obtained by calling
`UtilityNetworkDefinition.GetRules() : IReadOnlyList<Rule>`

`Type : RuleType`

- Returns either `JunctionJunctionConnectivity`, `JunctionEdgeConnectivity`, `Containment`, `Attachment`, or `EdgeJunctionEdgeConnectivity` telling us what kind of rule it is

`ID : int`

- Returns a numeric ID that can be used to identify the rule

`RuleElements : IReadOnlyList<RuleElement>`

- The elements that make up the rule (see next slide)

RuleElements

- This class stores information about the elements of a rule.

NetworkSource : NetworkSource

AssetGroup : AssetGroup

AssetType : AssetType

If null, this means that all AssetTypes of this AssetGroup participate in the rule

Terminal : Terminal

- The meaning of these parameters depend on the rule type, as illustrated below:

RuleType	NetworkSource1	AssetGroup1	AssetType1	Terminal1	NetworkSource2	AssetGroup2	AssetType2	Terminal2	NetworkSource3	AssetGroup3	AssetType3	Terminal3
JunctionJunctionConnectivity	Required	Required	Optional	Optional	Required	Required	Optional	Optional	N/A	N/A	N/A	N/A
JunctionEdgeConnectivity	Required	Required	Optional	Optional	Required	Required	Optional	N/A	N/A	N/A	N/A	N/A
Containment	Required	Required	Optional	N/A	Required	Required	Optional	N/A	N/A	N/A	N/A	N/A
Attachment	Required	Required	Optional	N/A	Required	Required	Optional	N/A	N/A	N/A	N/A	N/A
EdgeJunctionEdgeConnectivity	Required	Required	Optional	Optional	Required	Required	Optional	Optional	Required	Required	Optional	N/A
Note:												
"Optional" means wild card. "N/A" means non-applicable.												
In managed API, the value for AssetType (type Tuple<int, string> and and Terminal (type Terminal) is null for both "optional" and "N/A". The meaning of null is context-dependent.												

RuleElement

Sealed Class

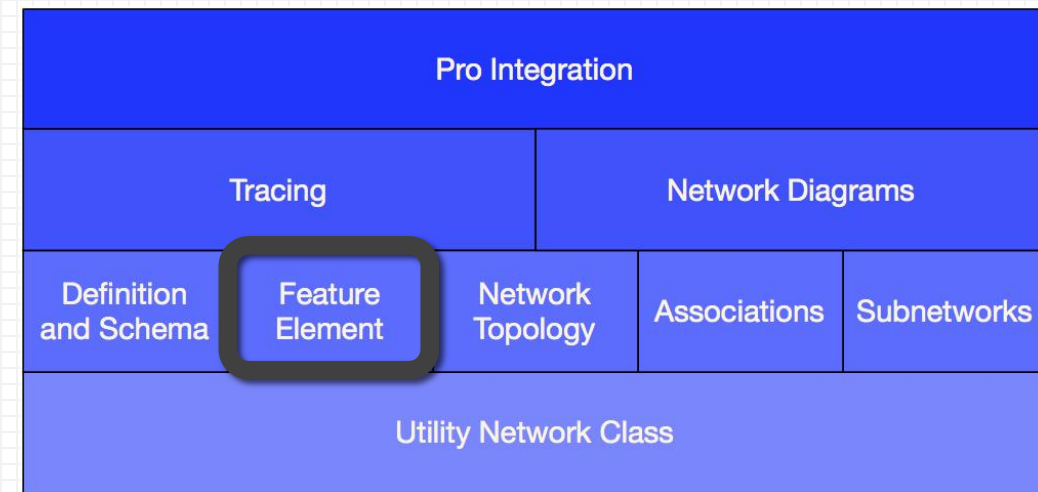
Properties

AssetGroup { get; } : AssetGroup

AssetType { get; } : AssetType

NetworkSource { get; } : NetworkSource

Terminal { get; } : Terminal



Feature Element

Feature Elements

- The FeatureElement class represents a Feature inside a utility network, *plus* a terminal (if applicable)
- These value objects are used across the utility network SDK
- Some examples:
 - Feature Elements are used to create and delete associations
 - Feature Elements specify starting points and barriers for use with tracing
 - Feature Elements are returned as results from traces

Creating Feature Elements

- FeatureElements are created using factory methods on UtilityNetwork

```
CreateFeatureElement(AssetType assetType, Guid globalID, int terminalID) :  
    FeatureElement
```

```
CreateFeatureElement(AssetType assetType, Guid globalID) : FeatureElement
```

```
CreateFeatureElement(Row row, int terminalID) : FeatureElement
```

```
CreateFeatureElement(Row row) : FeatureElement
```

FeatureElement Details

■ Read-only Properties

NetworkSource : NetworkSource

AssetGroup : AssetGroup

AssetType : AssetType

GlobalID : Guid

Terminal : Terminal

The screenshot shows the Visual Studio IDE with the 'FeatureElement' class selected in the Solution Explorer. The 'Properties' window is open, displaying the details of the 'FeatureElement' class, which is a 'Sealed Class' and implements 'IEquatable<FeatureElement>'. The 'Properties' section lists five read-only properties: 'AssetGroup', 'AssetType', 'GlobalID', 'NetworkSource', and 'Terminal', each with a 'get;' accessor. The 'Methods' section lists five methods: 'Equals' (two overloads), 'GetHashCode', and two operator methods for '!=' and '=='.

IEquatable<FeatureElement>

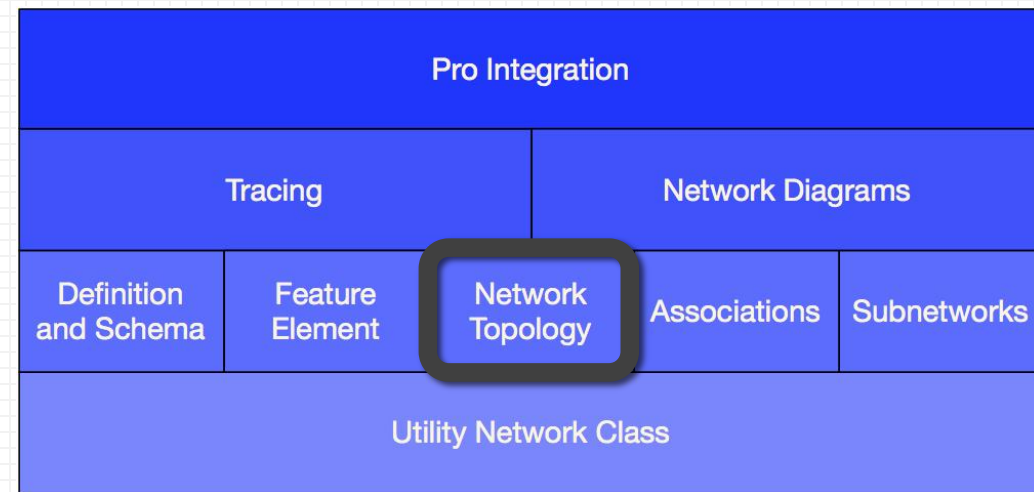
FeatureElement
Sealed Class

Properties

- AssetGroup { get; } : AssetGroup
- AssetType { get; } : AssetType
- GlobalID { get; } : Guid
- NetworkSource { get; } : NetworkSource
- Terminal { get; } : Terminal

Methods

- Equals(FeatureElement other) : bool
- Equals(object obj) : bool
- GetHashCode() : int
- operator !=(FeatureElement featureElement1, FeatureElement fe...
- operator ==(FeatureElement featureElement1, FeatureElement f...



Network Topology

Utility Network Topology

- The network topology stores connectivity, containment, and attachment information used by the utility network to facilitate fast network traversal/analytical operations
- Network topology is constructed from
 - Geometric coincidence *and...*
 - Associations *in combination with...*
 - A powerful Rules engine
- Topology is updated and validated with the `ValidateNetworkTopology` method on the `UtilityNetwork` class

UtilityNetwork: Validate Network Topology

`ValidateNetworkTopology(Geometry extent) : ValidationResult`

`ValidateNetworkTopology() : ValidationResult`

- Updates network topology within the provided extent. If the extent parameter is missing or null, the extent of the utility network is used.
- Calls to this routine must be wrapped in a transaction
 - Called from `Geodatabase.ApplyEdits()`

ValidationResult

- This class is used to return information from a call to `UtilityNetwork.ValidateTopology()`.

`UpdateLocalTime: DateTime`

- The `DateTime` when the `ValidateTopology()` call took place, converted to the time zone of the ArcGIS Pro client machine.

`HasErrors: bool`

- Returns whether there are any errors in the entire network topology.

`NumberOfDirtyAreas: int`

- Returns the number of dirty areas that were processed in the call to `ValidateTopology()`.

`IsFullUpdate: bool`

- Returns true if the `ValidateTopology` call was executed on the entire network.

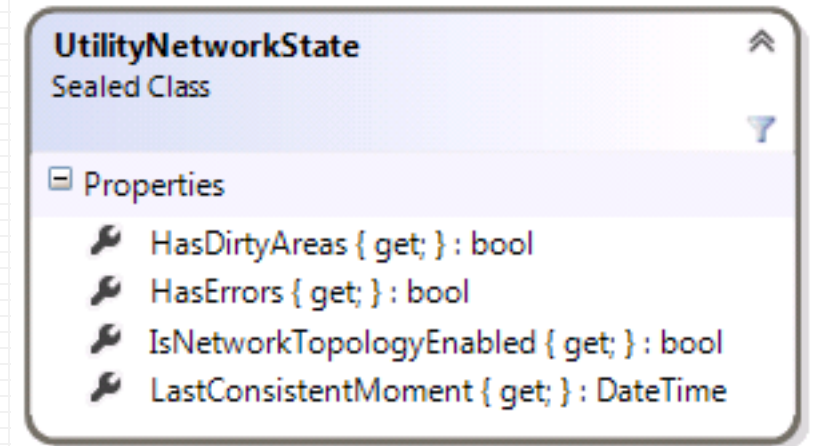
UtilityNetwork: Network Topology Queries

`GetExtent()` : `Envelope`

- Returns the extent of the feature classes within the network (same as other controller datasets in the product)

`GetState()` : `UtilityNetworkState`

- Returns the state of the network



UtilityNetworkState

- Represents the state of the utility network

HasDirtyAreas : bool

- Returns whether the network topology of the utility network contains any dirty areas.

HasErrors : bool

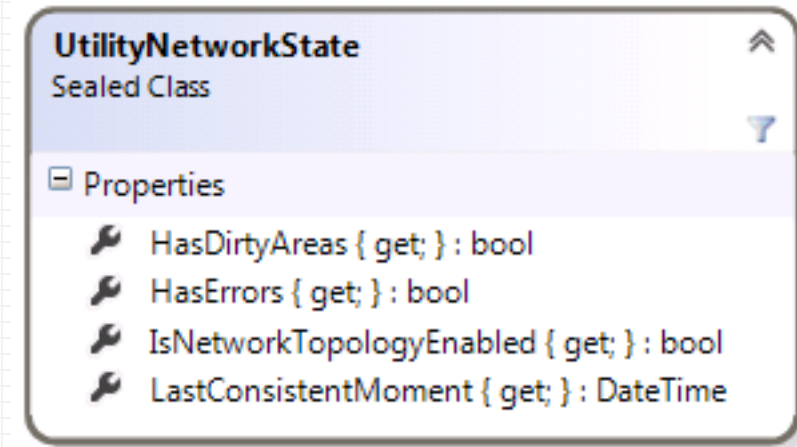
- Returns whether the network topology of the utility network has any errors.

IsNetworkTopologyEnabled : bool

- Returns whether the network topology of the utility network is enabled.

LastConsistentMoment : DateTime

- Returns the DateTime of the last time that the utility network was fully validated.

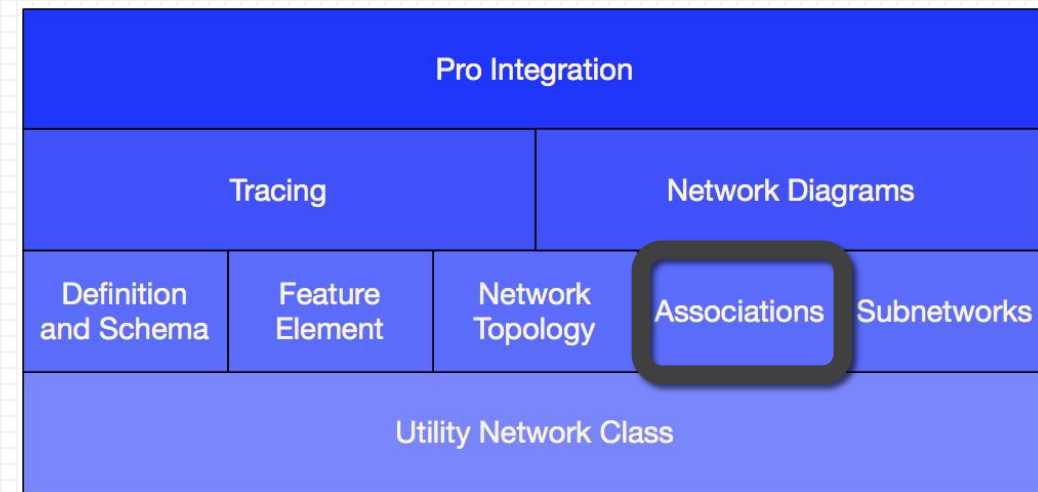


Fine-grained Topological Access is Not Provided

- Existing geometric network-based solutions provide fine-grained topological access
 - For example, the IForwardStar ArcObjects interface allows clients to step through the network, element by element
 - Tracing code can be built on top of these low-level primitives
- Fine-grained topological access is incompatible with a services-based solution
 - If each query resulted in a query to a service, performance and scalability would be unacceptable
- Instead, the Utility Network provides a powerful tracing framework that removes the need for fine-grained access to network topology

Future: Topological Neighborhoods

- A future version of the utility network SDK may provide limited query capability against the network topology
- Callers may be able to determine what features are directly connected, attached, or contained to or from an input feature
- This routine may have applicability for some special-case workflows and tools
 - E.g., Building a network inspector to aid in debugging
- It is not intended to be used repeatedly in an IForwardStar-like fashion and will perform poorly if used in this fashion



Associations

Associations

- The UtilityNetwork class contains routines that query and edit associations
- Associations are only *one* of the building blocks that are used to build network topology
- Associations on their own do not return an accurate or complete view of that network topology
 - Features that are connected by geometric coincidence are not returned by association queries
 - Association queries can return associations that have not yet been validated and are therefore not yet included in the topological index
 - Conversely, it will not return deleted records that still exist in the topological index
- However, Associations are the correct mechanism to use when building an editing tool, as they show the current edited state of the database

Creating and Deleting Associations

- The UtilityNetwork class provides a number of routines to create an edit associations
- These routines create dirty areas; they do not update network topology
- Calls to these routines must be wrapped in a transaction

```
AddConnectivityAssociation(FeatureElement fromElement, FeatureElement toElement) : void
```

```
AddContainmentAssociation(FeatureElement containerElement, FeatureElement contentElement) : void
```

```
AddContainmentAssociation(FeatureElement containerElement, FeatureElement contentElement, bool  
isVisibleContent) : void
```

```
AddStructuralAttachmentAssociation(FeatureElement structureElement, FeatureElement  
attachmentElement) : void
```

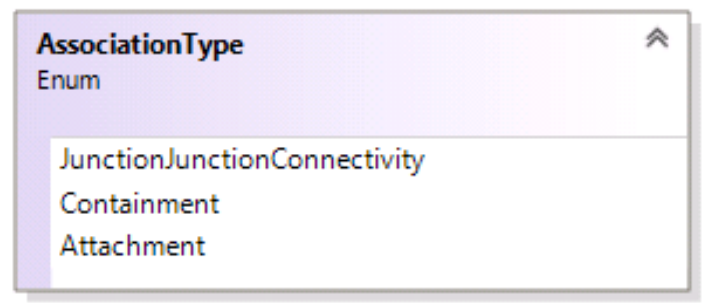
```
DeleteConnectivityAssociation(FeatureElement fromElement, FeatureElement toElement) : void
```

```
DeleteContainmentAssociation(FeatureElement containerElement, FeatureElement contentElement) :  
void
```

```
DeleteStructuralAttachmentAssociation(FeatureElement structureElement, FeatureElement  
attachmentElement) : void
```

Get Associations

- The UtilityNetwork class provides routines to query associations
- `GetAssociations(FeatureElement featureElement, AssociationType type) : IReadOnlyList<Association>`
- `GetAssociations(FeatureElement featureElement) : IReadOnlyList<Association>`
 - AssociationType is defined as follows:



- If the AssociationType parameter is missing, associations
- of **all** types are returned

The Association Class

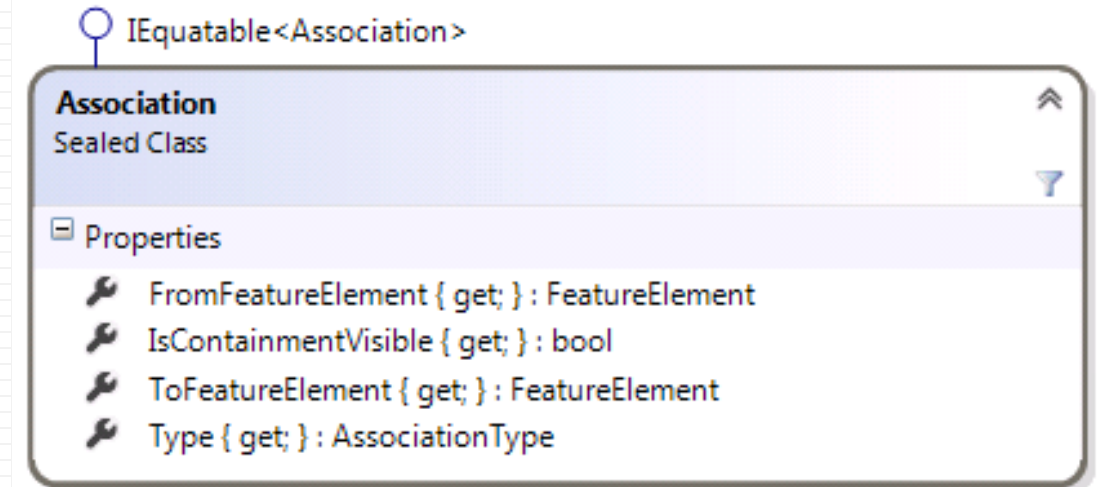
- Read-only properties on the Association class

FromFeatureElement : FeatureElement

IsContainmentVisible: bool

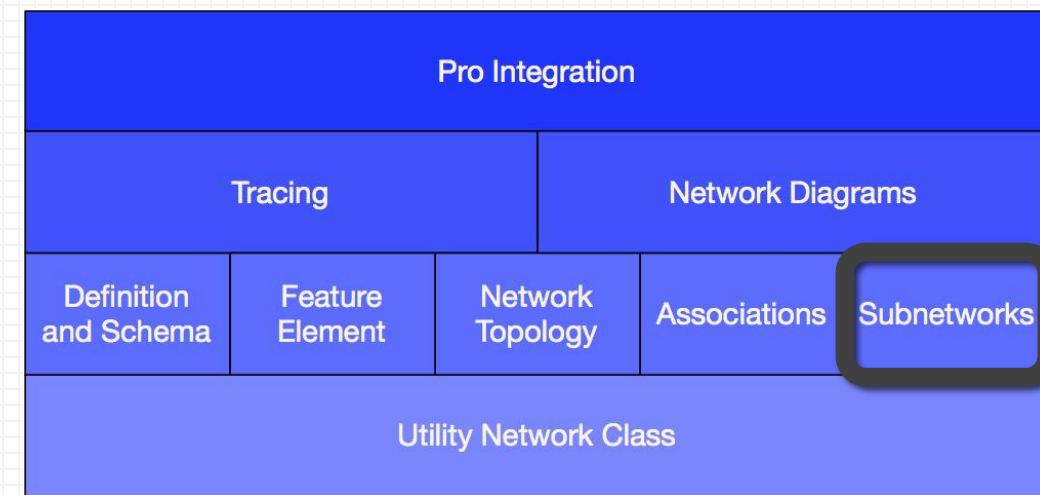
ToFeatureElement: FeatureElement

Type: AssociationType



Edge-Junction Terminal Connectivity

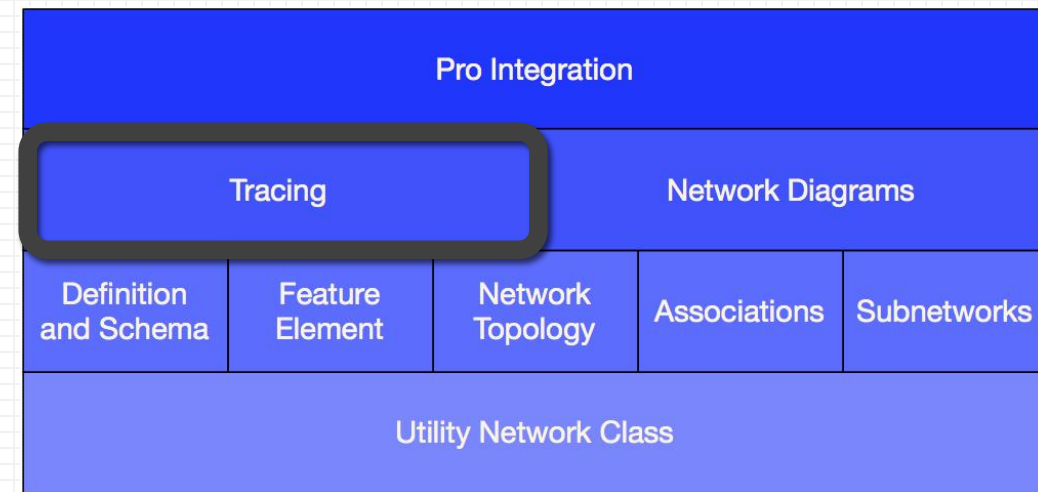
- The Utility Network also allows edges to be connected directly to junction feature Terminals
- These edge-junction terminal connections are not defined through associations
- Instead they are defined as follows
 - The endpoint of the line must be geometrically coincident to the junction feature
 - The FROMTERMINALID field on the line is used to store the terminal ID of the junction feature that intersects the first point of the line
 - The TOMTERMINALID field on the line is used to store the terminal ID of a junction feature that intersects the last point of the line
- These fields are edited using normal geodatabase editing routines



Subnetworks

Subnetworks

- The existing SDK contains a number of Subnetworks-based routines on the Utility Network object
- These routines are not currently operable and will be replaced with a new API in a future release



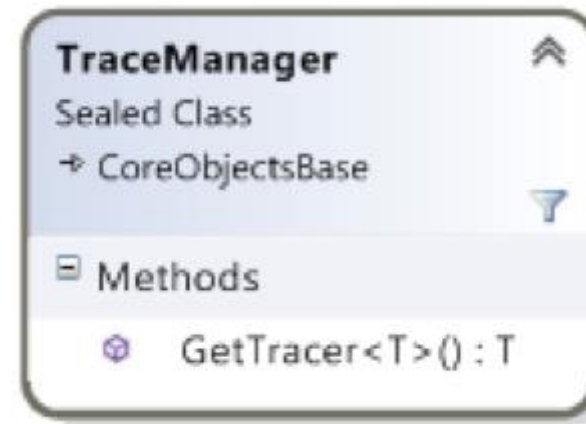
Tracing

TraceManager class

- The TraceManager class provides access to Tracer objects
- TraceManager objects are obtained through a call to `UtilityNetwork.GetTraceManager()`

`GetTracer<T>() : T`

- Returns a Tracer object of the specified type (described on next page)



Tracer Abstract Class

`Name : string`

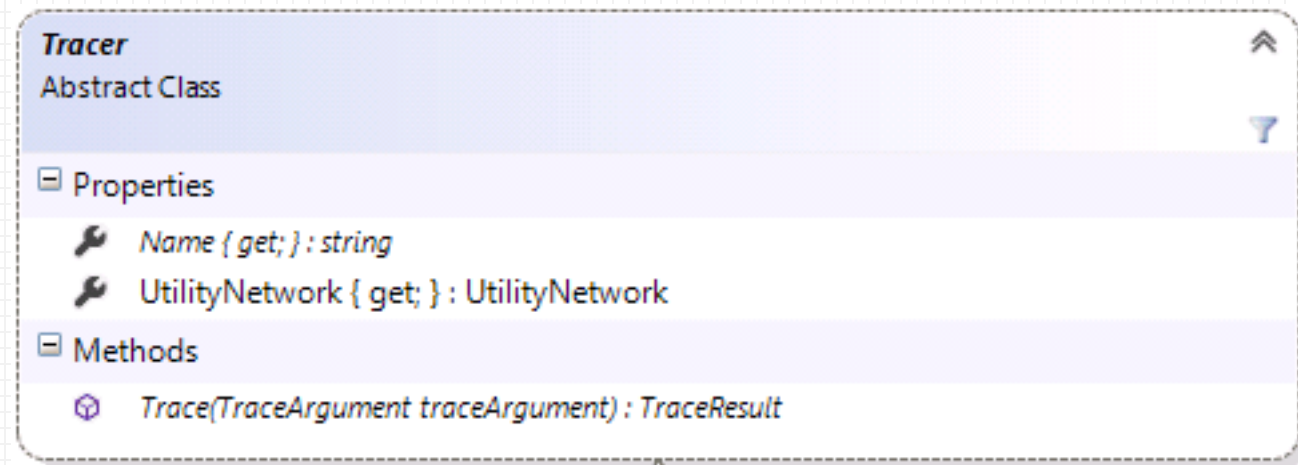
- The name of the Tracer object

`UtilityNetwork : UtilityNetwork`

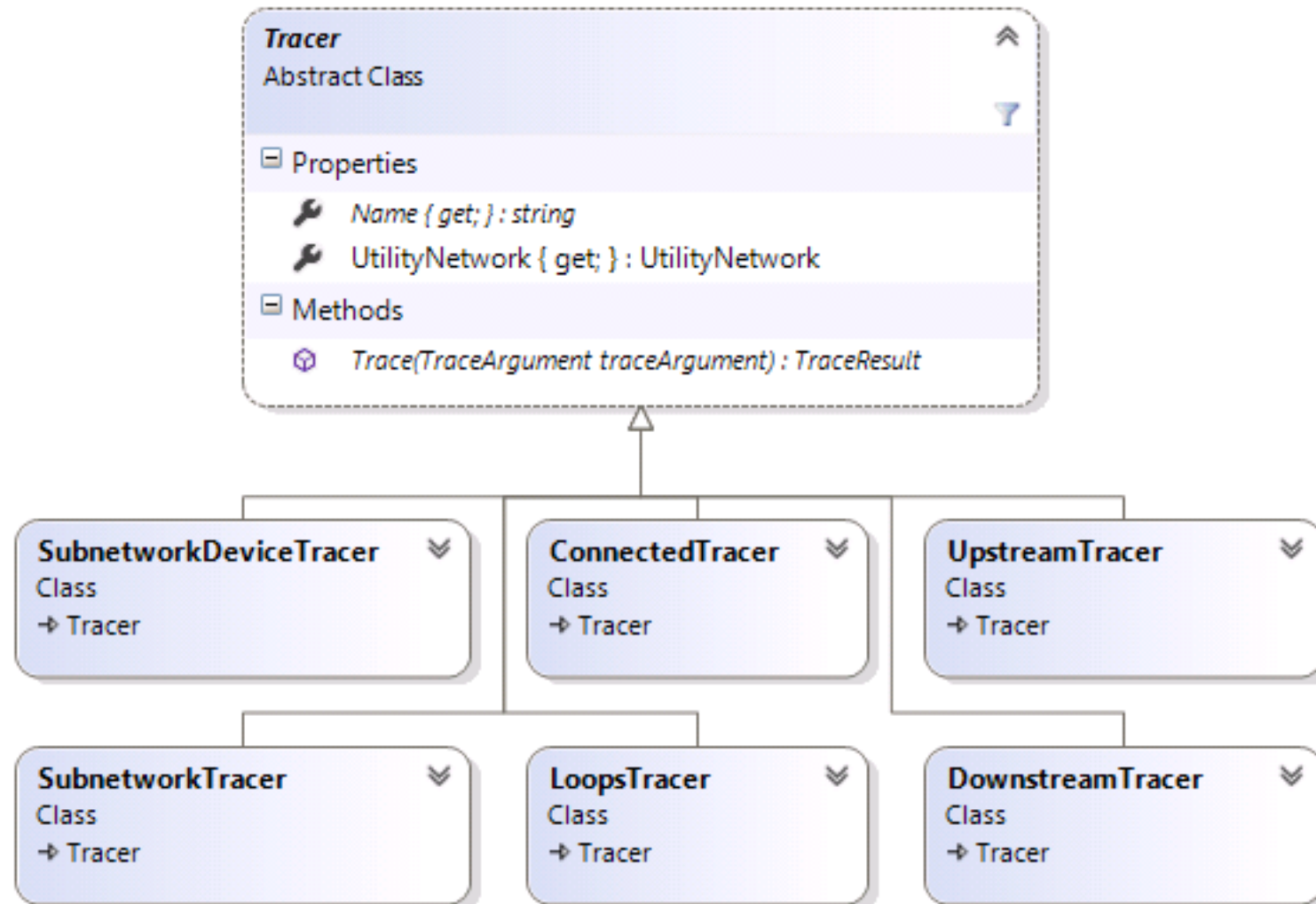
- A pointer back to the UtilityNetwork core object

`Trace(TraceArgument traceArgument) : TraceResult`

- Perform a trace



Tracer Concrete Classes



Extending Tracer

- Although we expect that most tracing functionality can be built using our configuration framework, there are other cases where clients will want a custom Tracer

1. Custom Tracers Could Wrap Esri Tracers

- For example, a partner might provide a Tracer objects which automatically include a set of NetworkAttribute filters to specify phase.
 - E.g., AcmeElectricDownstreamTracer
- Partners may wish to provide additional pre- or post-processing that the Esri configuration framework doesn't provide

2. Custom Tracers Could Be Written By Hand

- For example, a partner might make a call to a DMS or other external system to perform an analytic

3. Custom Tracers Could Be Written Using Low-Level Topology Primitives

- `UtilityNetworkTopology.GetConnectedEdgesAndJunctions`, etc.
- Would only perform well on SQLite or after we've implemented a client-side network cache

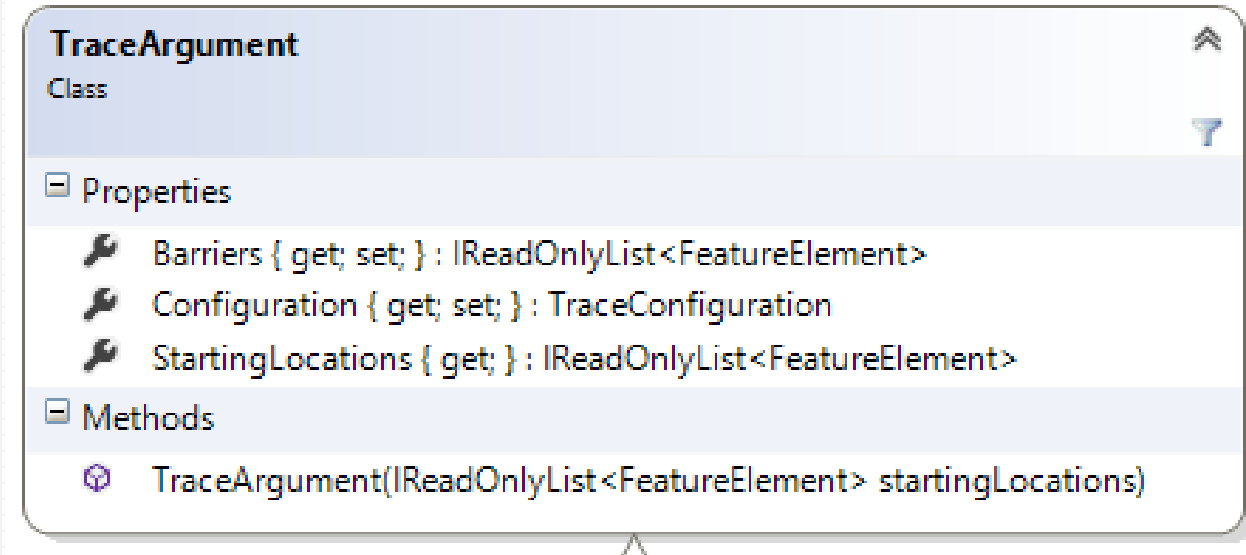
TraceArgument

- The TraceArgument class consolidates trace parameters
- Different tracers may subclass TraceArgument

Barriers : IReadOnlyList<FeatureElement>

StartingLocations: IReadOnlyList<FeatureElement>

Configuration: TraceConfiguration



The screenshot shows the definition of the `TraceArgument` class in a code editor. The class is marked as a `Class`. It has three properties: `Barriers` (type `IReadOnlyList<FeatureElement>`), `Configuration` (type `TraceConfiguration`), and `StartingLocations` (type `IReadOnlyList<FeatureElement>`). It also has one method: `TraceArgument(IReadOnlyList<FeatureElement> startingLocations)`.

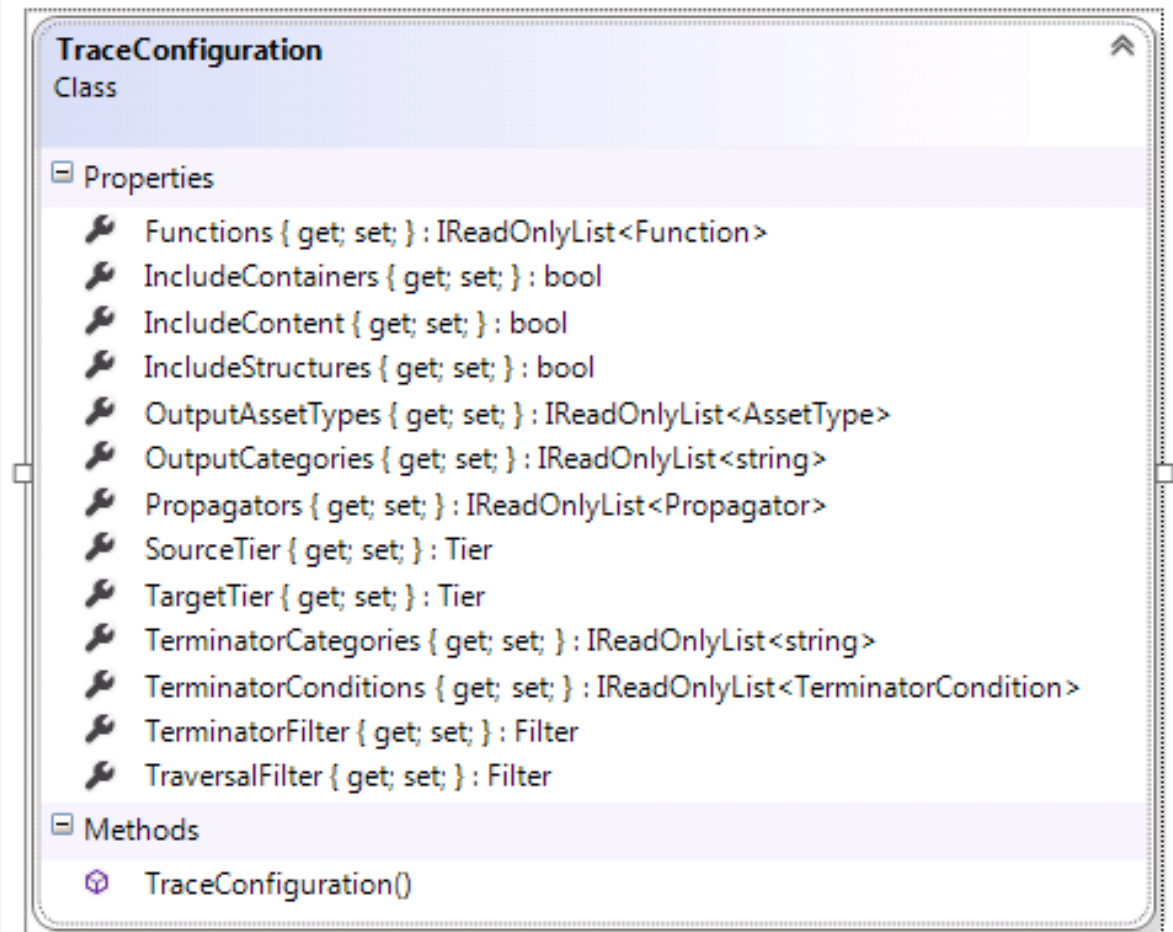
```
TraceArgument
Class

Properties
  Barriers { get; set; } : IReadOnlyList<FeatureElement>
  Configuration { get; set; } : TraceConfiguration
  StartingLocations { get; } : IReadOnlyList<FeatureElement>

Methods
  TraceArgument(IReadOnlyList<FeatureElement> startingLocations)
```

The Trace Configuration object

- Encapsulates additional optional parameters for a trace
- The properties can be categorized as follows:
 1. Basics
 2. Traversal Filters
 3. Functions
 4. Terminators
 5. Output Filters
 6. Propagators



1 TraceConfiguration - Basics

`IncludeContainers: bool`

- Default is false.

`IncludeContent: bool`

- Default is false.

`IncludeStructures: bool`

- Default is false.

- `IncludeContainers` and `IncludeStructures` are transitive
 - i.e., if they are both true, if a container contains a result element, and that container is attached to a structure, the structure is returned even if the result element is not directly attached to the structure
 - **Note: This is not yet implemented in the current alpha software.**
- `IncludeContainers` is recursive
 - i.e., if true, if a result element is inside a nested container, both containers are returned

1 Trace Configuration - Basics

SourceTier : Tier

- This property is optional. If not null, the Trace routines will perform an additional check to validate that the starting points and barriers belong to this tier.

TargetTier : Tier

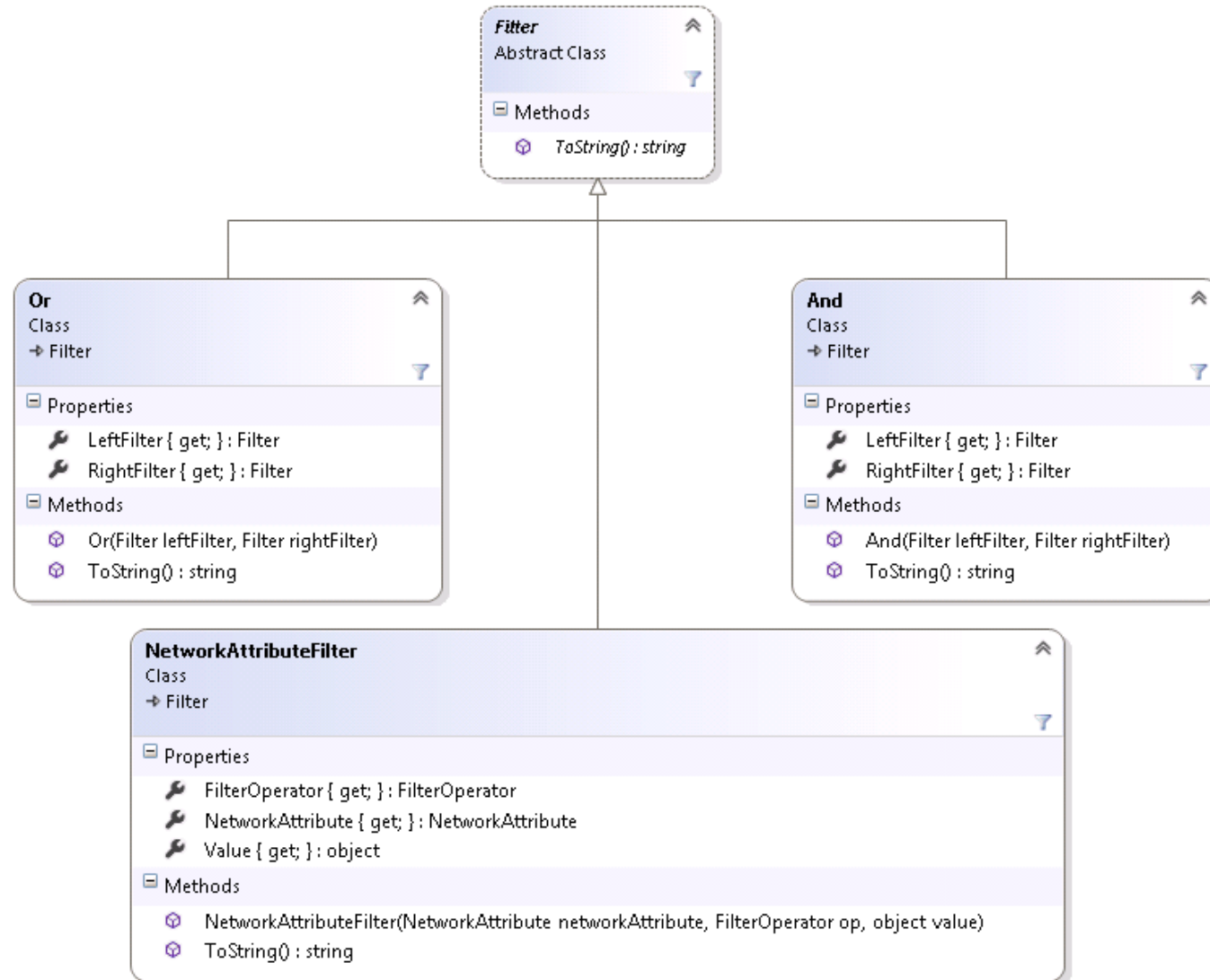
- This property is optional. If null, upstream and downstream traces will stop in the current tier (i.e., that devices that delineate the tier boundary). If a **TargetTier** is specified, the trace will continue upstream or downstream into the specified tier.

2 TraceConfiguration – Traversal Filters

- As the Tracer navigates through the network, filters can be applied to allow traversal
- If the visited network element meets the criteria of the filter, traversal continues
- If the visited network element does not meet the criteria, that element is NOT included in the results
- A null traversal filter always permits traversal
- Traversal filters are based on comparisons of network attributes
- Traversal filters can be combined with boolean And and Or operations to form more complex filters

TraversalFilter : Filter

Traversal Filter Class Hierarchy



NetworkAttributeFilter

■ `NetworkAttributeFilter(NetworkAttribute networkAttribute, FilterOperator op, object Value)`

- Creates a network attribute filter. The filter allows continued traversal depending on the result of the expression

`FilterOperator : FilterOperator`

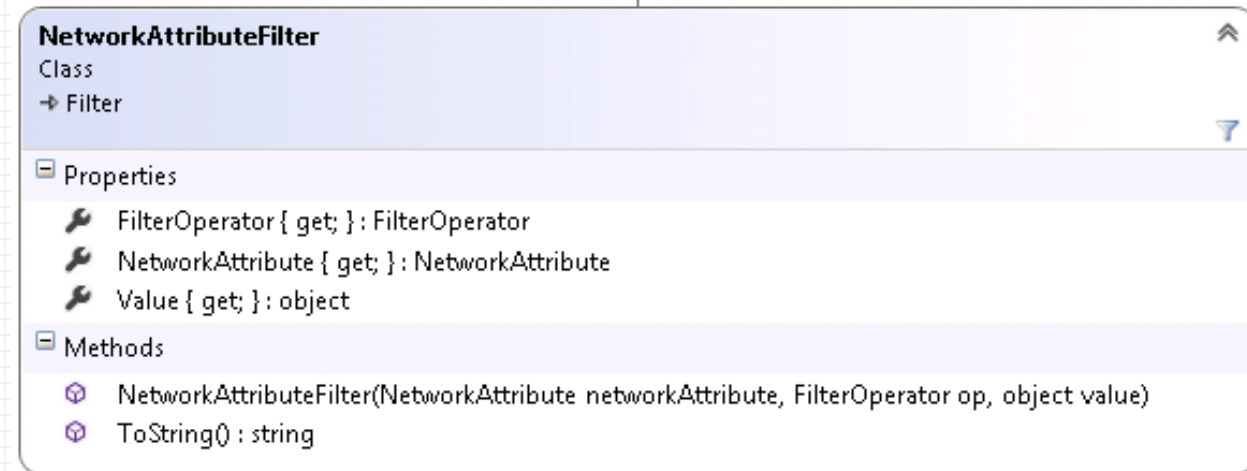
- Returns the boolean operator used to defined the filter. Possible values are Equal, NotEqual, BitwiseAnd, GreaterThan, GreaterThanEqual, LessThan, LessthanEqual, and BitwiseAny.

`NetworkAttribute : NetworkAttribute`

- The network attribute to test

`Value : object`

- The value to test against



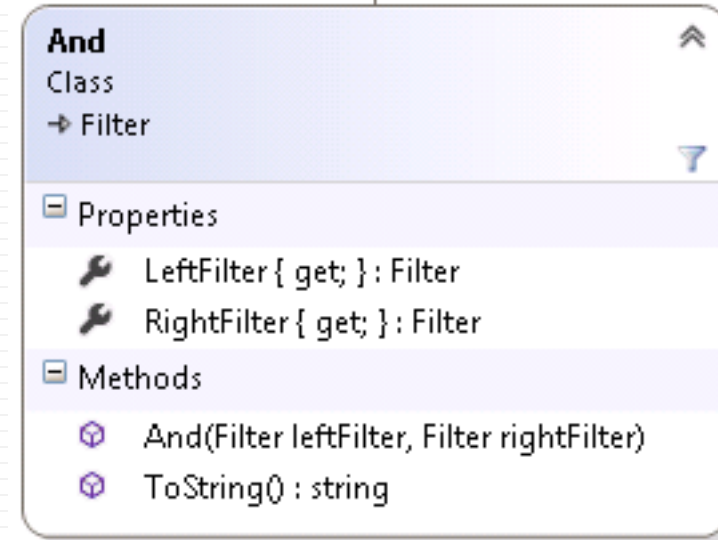
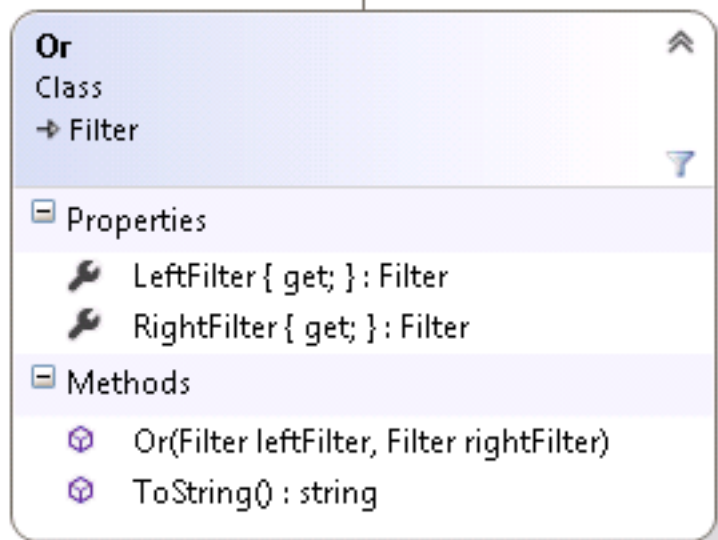
The screenshot displays the `NetworkAttributeFilter` class in a code editor. The class is identified as a `Class` and inherits from `Filter`. It features three properties: `FilterOperator` (type `FilterOperator`), `NetworkAttribute` (type `NetworkAttribute`), and `Value` (type `object`). The methods section includes a constructor `NetworkAttributeFilter(NetworkAttribute networkAttribute, FilterOperator op, object value)` and a `ToString() : string` method.

Combination Filters

- The **Or** and **And** classes are provided to allow chaining of logical expressions

Or(Filter leftOperand, Filter rightOperand)

And(Filter leftOperand, Filter rightOperand)



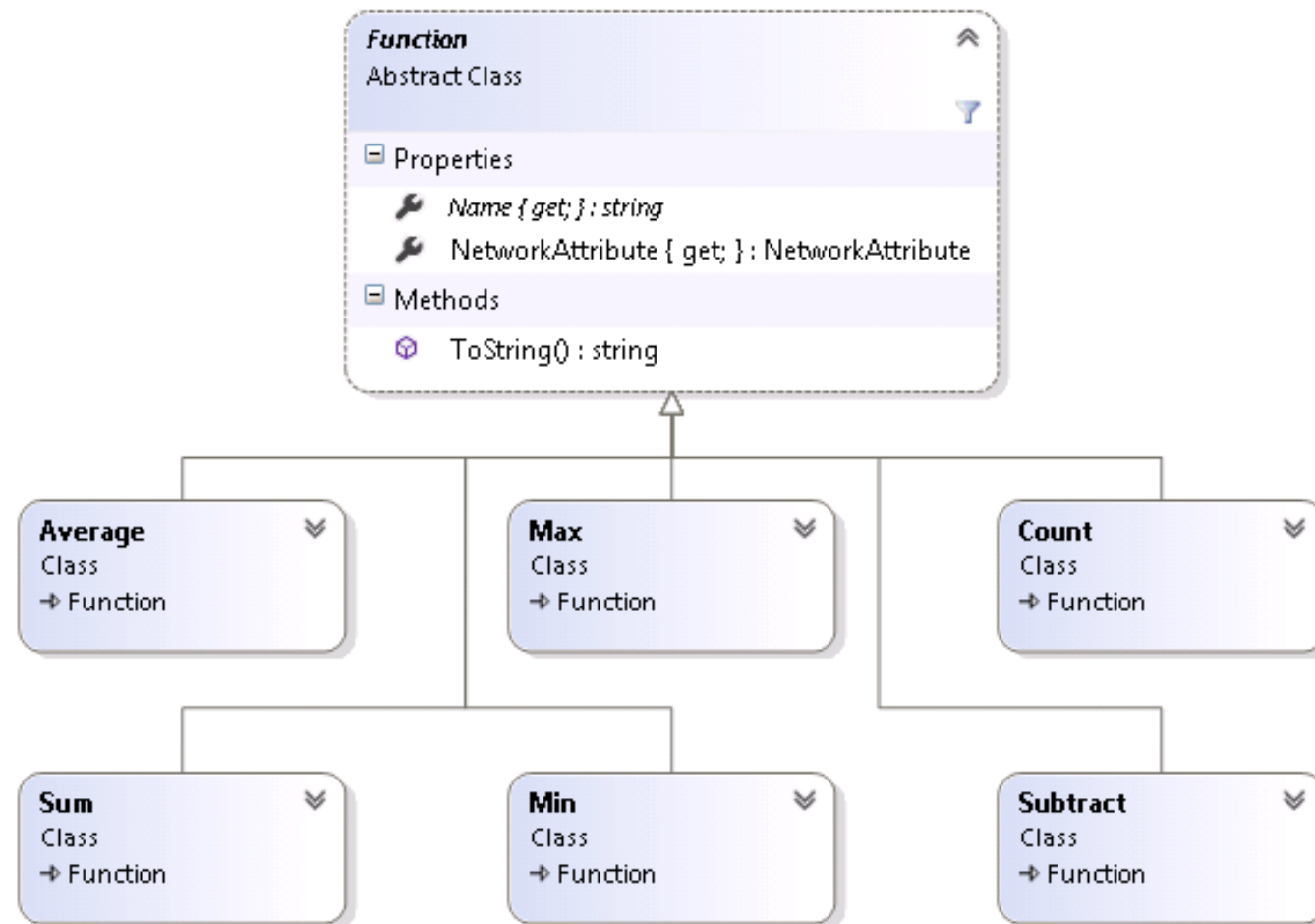
3 Trace Configuration - Functions

- The caller can specify a collection of Functions for a trace
- These functions are evaluated at each *applicable* feature element
 - The meaning of *applicable* varies per trace
 - For an upstream trace, the functions are evaluated for each upstream element
 - For a subnetwork trace, the functions are evaluated for each element in the subnetwork
 - For a subnetwork sources trace, the functions are evaluated for each subnetwork source element
 - Etc.
- At the conclusion of the trace, function results can be obtained globally **and for each applicable element**
 - **Function elements for each feature are not yet implemented**
 - Described later in this document

Functions : IReadOnlyList<Function>

Function Class Hierarchy

- Eventually, custom functions may be supported



Meaning of Trace Functions

Function	Meaning
Sum	Sums up the value of the network attribute on each applicable element
Subtract	Takes the network attribute value from the starting point as the base number, and then subtracts the value of the network attribute on each applicable element
Average	Averages the value of the network attribute on each applicable element
Count	Counts the number of applicable elements
Min	The minimum value of the network attribute on each applicable element
Max	The maximum value of the network attribute on each applicable element

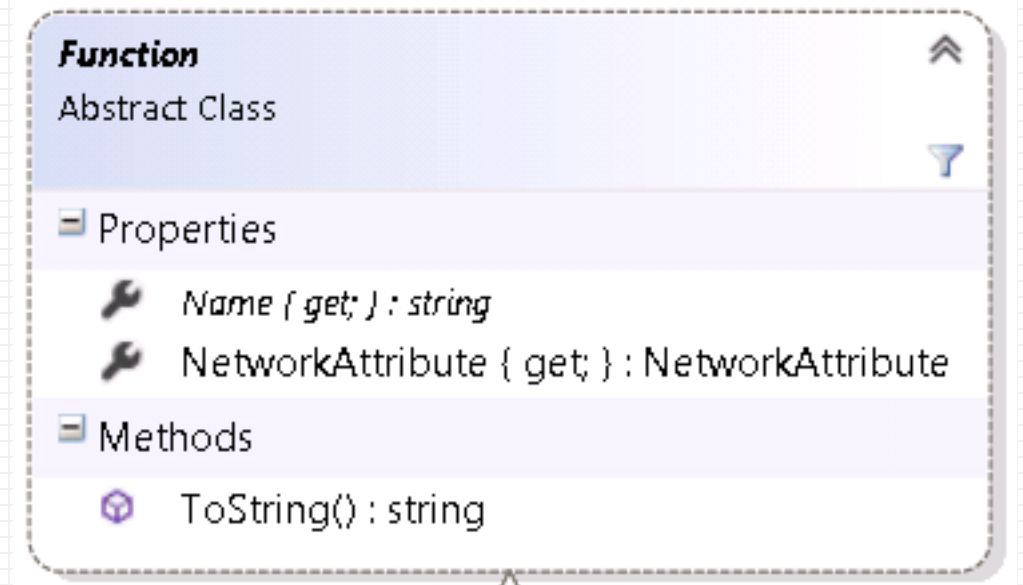
Function class

`NetworkAttribute : NetworkAttribute`

The `NetworkAttribute` used as an input to the function.

`Name: string`

The name of the function.



4 Trace Configuration - Terminators

- Terminators provide a way to terminate further traversal when visiting a network element
- Terminators take three different forms
 - Terminator Categories
 - Terminator Conditions
 - Terminator Filters
- If ANY category or condition or filter is satisfied, further traversal is terminated
 - That particular network element IS included in the trace results

TerminatorConditions: `ICollection<TerminatorCondition>`

- Traversal stops if the TerminatorCondition evaluates to true
- TerminatorConditions are based on comparing a function result to a specified value
- TerminatorConditions are always combined using Or. I.e., if ANY TerminatorCondition returns true, traversal is terminated

Terminator Categories

`TerminatorCategories : IReadOnlyList<string>`

- Traversal stops if the network element includes any Category in this list

TerminatorCondition

TerminatorCondition	
Class	
Properties	
FilterOperator { get; } : FilterOperator	
Function { get; } : Function	
Value { get; } : double	
Methods	
TerminatorCondition(Function function, FilterOperator op, double value)	
ToString() : string	

`TerminatorConditions: IReadOnlyList<TerminatorCondition>`

- Traversal stops if the TerminatorCondition evaluates to true
- TerminatorConditions are based on comparing a function result to a specified value
- TerminatorConditions are always combined using Or. I.e., if ANY TerminatorCondition returns true, traversal is terminated

`TerminatorCondition(Function function, FilterOperator op, double Value)`

- Remember that a Function references a NetworkAttribute and a function that is applied to it (Min, Max, Count, etc.)
- The value of the Function for the given network element is compared against the Value parameter using the provided FilterOperator.

Terminator Filter

TerminatorFilter : Filter

- TerminatorFilters use the same Filter type as a Traversal Filter
- TerminatorFilters are NOT the same as negating a Traversal Filter
- For Traversal Filters...
 - If the visited element does not meet the filter criteria, that element **is not included** in the trace results.
- For Terminator Filters...
 - If the visited element meets the filter criteria, that element **is included** in the trace results.
- In both cases, further traversal is halted.

5 Trace Configuration – Output Filtering

- The Trace Configuration object provides two different ways to filter the results of a trace
- Output Filtering takes two different forms
 - Output Categories
 - Output Asset Types
- If ANY category or asset type is satisfied, the element is included in the result set

`OutputCategories : IReadOnlyList<string>`

- The network element is included in the result set if it includes any Category in this list

`OutputAssetTypes: IReadOnlyList<AssetType>`


- The network element is included in the result set if it references a feature that is any AssetType in this list


6 Trace Configuration - Propagators


- Propagators allow a subset of NetworkAttribute values to propagate through a trace
- The canonical example is phase propagation- open devices along the network will restrict some phases from continuing along the trace


Propagators : IReadOnlyList<Propagator>


Propagator
Class


 Properties


 FilterOperator { get; } : FilterOperator


 NetworkAttribute { get; } : NetworkAttribute

 PropagatorFunction { get; } : PropagatorFunction

 Value { get; } : double

 Methods

 Propagator(NetworkAttribute networkAttribute, PropagatorFunction propagatorFunction, FilterOperator filterOperator, double value)

 ToString() : string

Propagator

- A propagator defines the propagation of a network attribute along a traversal, as well as provide a filter to stop traversal.
- Propagators are only applicable to subnetwork-based traces (subnetwork, subnetworksource, upstream, downstream)

NetworkAttribute : NetworkAttribute

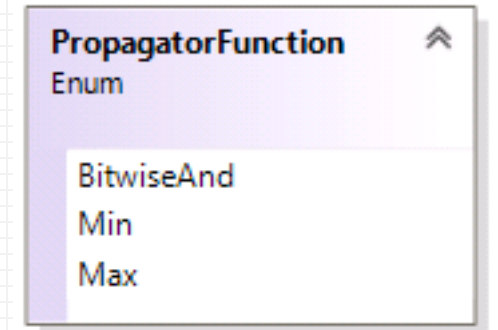
PropagatorFunction : PropagatorFunction

- Operators may be specified that control how the attribute is propagated downstream from a source
 - For attributes that correspond to numeric values, a min or max operator may be specified
 - For attributes represented as bitsets, bit-wise set operators may be specified

FilterOperator : FilterOperator

Value: double

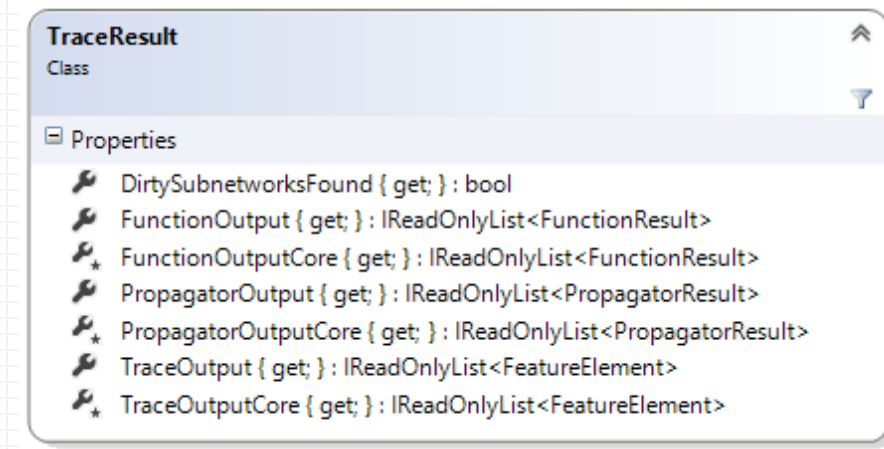
- The FilterOperator and Value provide a means to stop traversal based on a propagated network attribute.



How Propagators Work

- Propagator values are computed as a pre-process step before the main trace takes place
 - Starting at each source, the propagator uses its `PropagatorFunction` and `NetworkAttribute` to calculate a value at each feature element
 - This pre-process traversal covers the extent of a subnetwork
- During the trace itself, Propagator filters are tested at the same time as Traversal filters
 - *Traversal filters* work on *Network Attributes* and *continue traversal* if true
 - *Propagator filters* work on *propagated values* and *stop traversal* if false
- Propagated values are returned with trace results
 - `TraceResult.PropagatorOutput`
 - This property is not yet implemented

TraceResult -1



- The TraceResult class encapsulates all the information that is returned from a trace operation

TraceOutput : IReadOnlyList<NetworkElement>

- The actual network elements that are returned in the trace results

DirtySubnetworksFound : bool

- Returns whether any dirty subnetworks were encountered during the network traversal.

FunctionOutput : IReadOnlyList<FunctionResult>

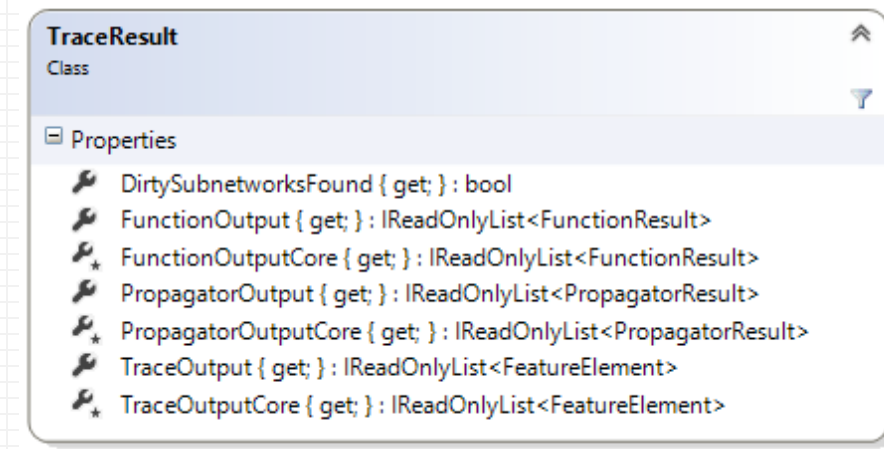
- The output from all of the Functions included with the trace

PropagatorOutput : IReadOnlyList<PropagatorResult>

- The output from the propagator functions

- This property is not yet implemented

TraceResult - 2



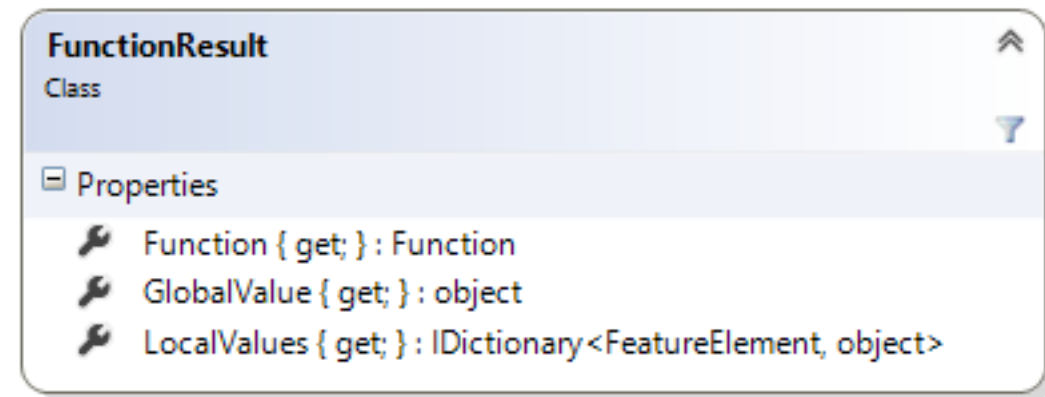
FunctionOutputCore : IReadOnlyList<FunctionResult>

PropagatorOutputCore : IReadOnlyList<PropagatorResult>

TraceOutputCore : IReadOnlyList<FeatureElement>

- These protected properties are provided for partners who wish to extend the TraceResult class.

FunctionResult



- One FunctionResult object is returned for every Function passed into the TraceConfiguration

Function : Function

- Returns the input Function

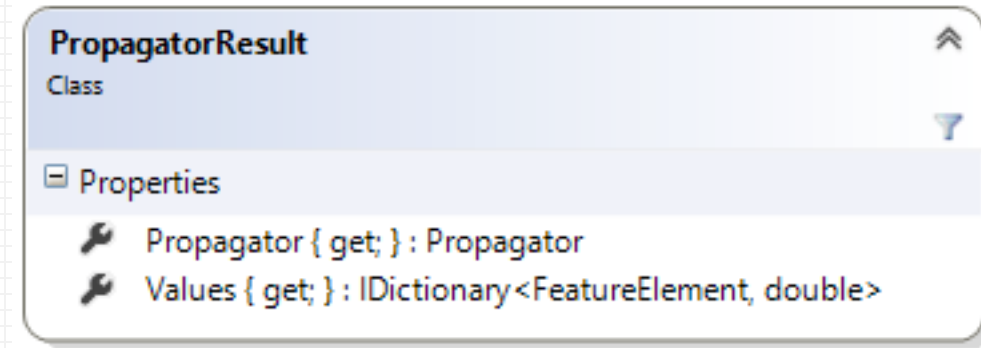
GlobalValue : object

- This routine returns the global function result- i.e., the result of applying the function to *all* of the network elements traversed.

LocalValues : IDictionary<FeatureElement, object>

- Returns the result of the function at *each* network element
- This functionality is not yet implemented

PropagatorResult



- One `PropagatorResult` object is returned for every `Propagator` passed into the `TraceConfiguration`

`Propagator` : `Propagator`

- Returns the input `Propagator`

`Values` : `IDictionary<FeatureElement, double>`

- Returns the propagated value at each feature element
- This functionality is not yet implemented

```

public void LoadAndCountPerPhaseTrace(UtilityNetwork utilityNetwork, IReadOnlyList<FeatureElement> startingPoints, int phaseValue)
{
    using (TraceManager traceManager = utilityNetwork.GetTraceManager())
    {
        DownstreamTracer downstreamTracer = traceManager.GetTracer<DownstreamTracer>();

        UtilityNetworkDefinition definition = utilityNetwork.GetDefinition();

        // Create a filter to allow traversal of devices, lines, and junctions with the specified phase
        NetworkAttribute normalPhasesNetworkAttribute = definition.GetNetworkAttribute("PhasesNormal");
        Filter phaseFilter = new NetworkAttributeFilter(normalPhasesNetworkAttribute, FilterOperator.BitwiseAnd, phaseValue);

        // Create a filter to terminate tracing at open devices
        NetworkAttribute deviceStatusAttribute = definition.GetNetworkAttribute("Device status");
        Filter deviceStatusFilter = new NetworkAttributeFilter(deviceStatusAttribute, FilterOperator.Equal, DeviceStatusOpen);

        // Create function to add up loads on service points
        NetworkAttribute loadNetworkAttribute = definition.GetNetworkAttribute("Load");
        Function sumServicePointLoadFunction = new Sum(loadNetworkAttribute);

        // Filter results based on a ServicePoint category
        IReadOnlyList<string> categoryList = new List<string>() { "ServicePoint" };

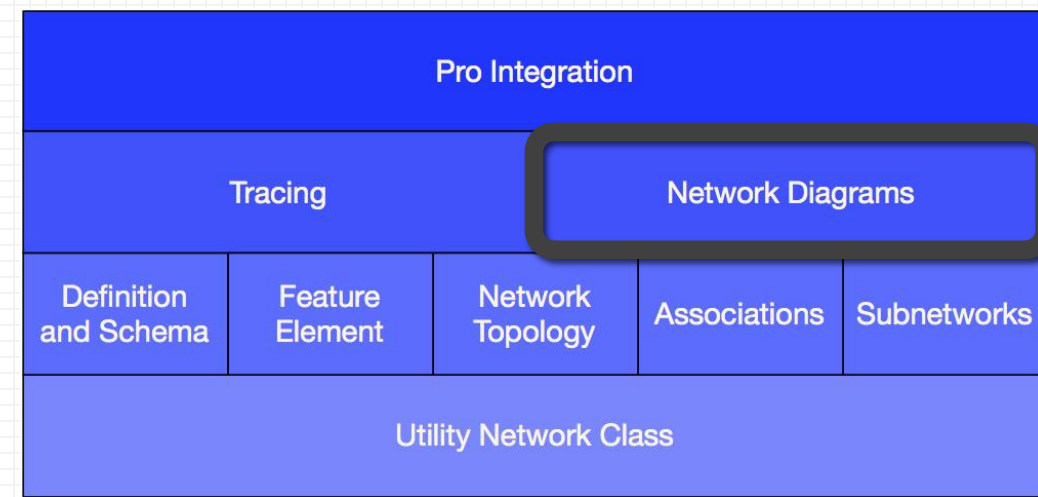
        // Create Trace Configuration object
        TraceConfiguration traceConfiguration = new TraceConfiguration();
        traceConfiguration.TraversalFilter = phaseFilter;
        traceConfiguration.TerminatorFilter = deviceStatusFilter;
        traceConfiguration.Functions = new List<Function>() { sumServicePointLoadFunction };
        traceConfiguration.OutputCategories = categoryList;

        // Execute the trace
        TraceArgument traceArgument = new TraceArgument(startingPoints);
        traceArgument.Configuration = traceConfiguration;
        TraceResult traceResults = downstreamTracer.Trace(traceArgument);

        //Output results
        int countCustomers = traceResults.TraceOutput.Count;
        int sumLoad = (int) traceResults.FunctionOutput.First().GlobalValue;

        Console.WriteLine("Number of customers assigned to phase: " + countCustomers);
        Console.WriteLine("Total load for this phase: " + sumLoad);
    }
}

```

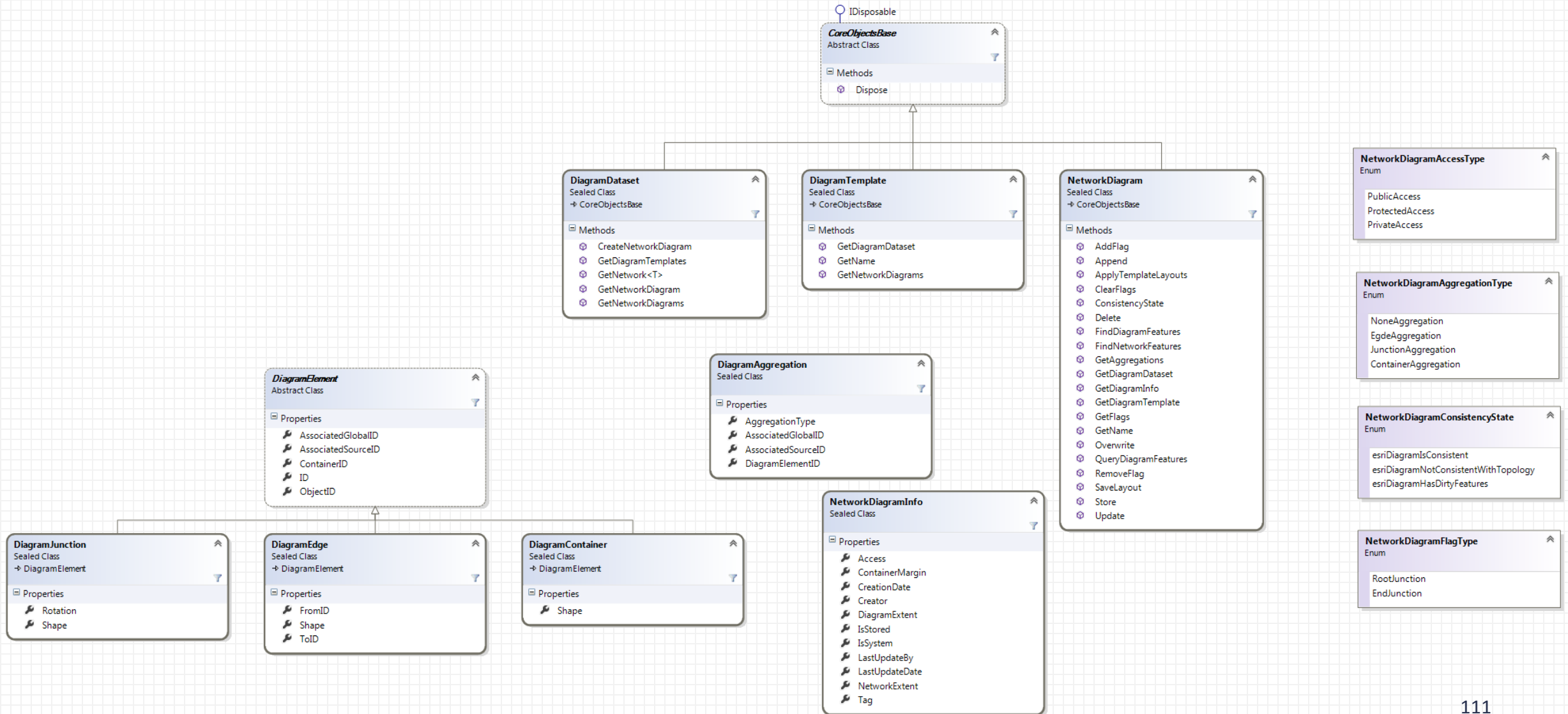


Network Diagrams

Network Diagrams SDK Introduction

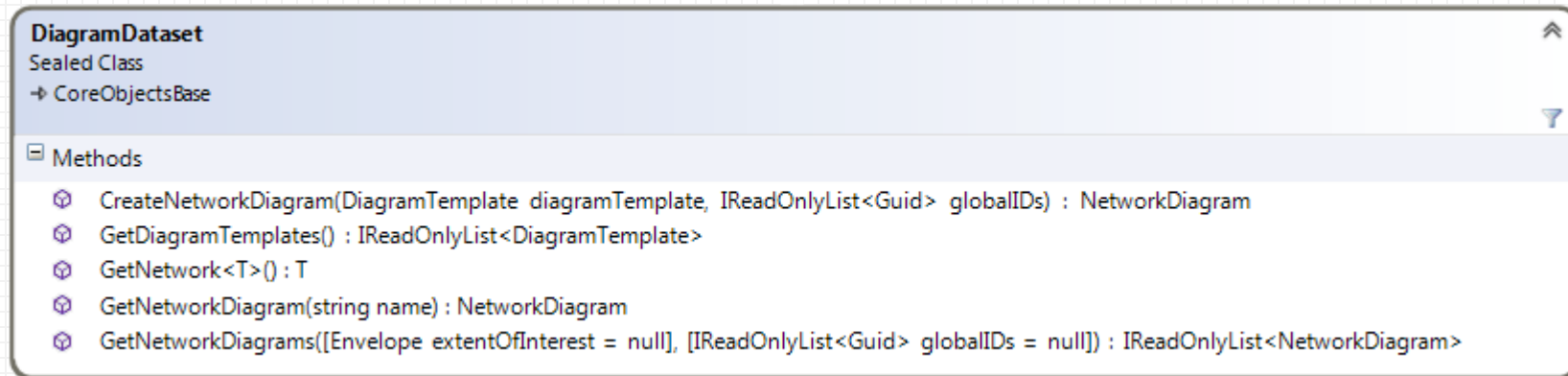
- Provides access to the Network Diagrams framework allowing developers to manage network diagrams.
- SDK functions to allow:
 - Retrieving existing diagram templates
 - Creating, storing, and deleting network diagrams
 - Retrieving stored network diagrams that cover a given extent or contained particular utility network element
 - Updating, overwriting, and appending network diagrams
 - Coding custom layout on a network diagram
- The SDK does not provide functionality to
 - Create or configure diagram templates
 - Apply Network Diagram core algorithm layouts on your network diagram
- These tasks can be accomplished with Python or by calling geoprocessing tools directly from C#
 - `Geoprocessing.ExecuteToolAsync`

Network Diagrams SDK Introduction



DiagramDataset class

- The DiagramDataset class serves as the core class in the Network Diagrams SDK
- It is obtained using `UtilityNetwork.GetDiagramDataset`
- Use the DiagramDataset class to:
 - Create network diagrams
 - Retrieve diagram templates
 - Retrieve network diagrams
 - Retrieve the related utility network



The screenshot shows the **DiagramDataset** class in a code editor. It is a **Sealed Class** that inherits from **CoreObjectsBase**. The **Methods** section lists five methods:

- `CreateNetworkDiagram(DiagramTemplate diagramTemplate, IReadOnlyList<Guid> globalIDs) : NetworkDiagram`
- `GetDiagramTemplates() : IReadOnlyList<DiagramTemplate>`
- `GetNetwork<T>() : T`
- `GetNetworkDiagram(string name) : NetworkDiagram`
- `GetNetworkDiagrams([Envelope extentOfInterest = null], [IReadOnlyList<Guid> globalIDs = null]) : IReadOnlyList<NetworkDiagram>`

DiagramDataset: Create Network Diagram

```
CreateNetworkDiagram(DiagramTemplate diagramTemplate, IReadOnlyList<Guid> globalIDs) :  
    NetworkDiagram
```

- This routine creates a temporary network diagram from a set of utility network feature GlobalIDs.
- This routine is a transactional operation; that is, calls to it must be wrapped in a transaction
 - Called from `Geodatabase.ApplyEdits()`

DiagramDataset: Retrieve Network Diagram(s)

```
GetNetworkDiagram(string name): NetworkDiagram
```

```
GetNetworkDiagrams([Envelope extentOfInterest = null], [IReadOnlyList<Guid> globalIDs = null]): IReadOnlyList<NetworkDiagram>
```

- You can retrieve a particular diagram stored in the diagram dataset using its name.
- You can also get the list of all the diagrams stored in the diagram dataset, those that intersect a given extent or those referencing specific network feature GUIDs

DiagramDataset: Retrieve Diagram Templates

`GetDiagramTemplates() : IReadOnlyList<DiagramTemplate>`

- This routine allows you to retrieve all the diagram templates related to a utility network

DiagramTemplate Class

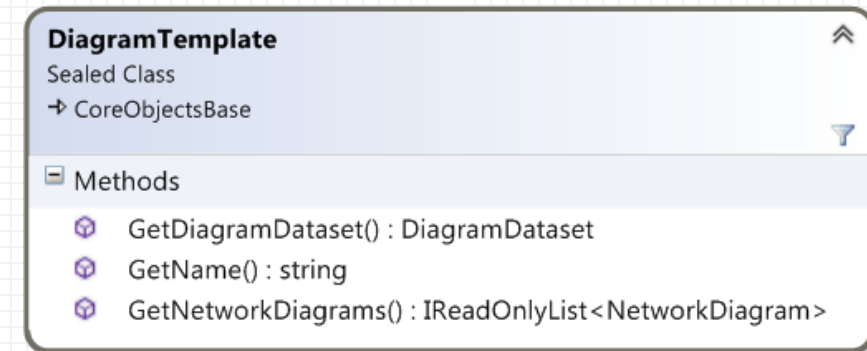
- The list of the Diagram Templates related to a utility network can be obtained from the diagram dataset using `DiagramDataset.GetDiagramTemplates`

- Methods

`GetNetworkDiagrams() : IReadOnlyList<NetworkDiagram>`

`GetDiagramDataset() : DiagramDataset`

`GetName() : string`



The image shows a software IDE window displaying the definition of the **DiagramTemplate** class. The class is identified as a "Sealed Class" and inherits from `CoreObjectsBase`. A "Methods" section is expanded, showing three methods: `GetDiagramDataset() : DiagramDataset`, `GetName() : string`, and `GetNetworkDiagrams() : IReadOnlyList<NetworkDiagram>`. Each method is preceded by a small icon representing a method.

DiagramTemplate
Sealed Class
→ CoreObjectsBase

Methods

- GetDiagramDataset() : DiagramDataset
- GetName() : string
- GetNetworkDiagrams() : IReadOnlyList<NetworkDiagram>

NetworkDiagram Class

- This class represents a diagram generated from a portion of the utility network
- New network diagrams can be created using a factory method on the diagram dataset
 - `DiagramDataset.CreateNetworkDiagram(DiagramTemplate diagramTemplate, IReadOnlyList<Guid> globalIDs) : NetworkDiagram`
- Network diagrams that already exist in the diagram dataset can be retrieved
 - Directly from the diagram dataset
 - `DiagramDataset.GetNetworkDiagrams([Envelope extentOfInterest = null], [IReadOnlyList<Guid> globalIDs = null]): IReadOnlyList<NetworkDiagram>`
 - `DiagramDataset.GetNetworkDiagram(string name): NetworkDiagram`
 - From the diagram template they are based on
 - `DiagramTemplate.GetNetworkDiagrams() : IReadOnlyList<NetworkDiagram>`

NetworkDiagram Class

NetworkDiagram

Sealed Class

→ CoreObjectsBase

Methods

- ⊞ AddFlag(NetworkDiagramFlagType flagType, int diagramElementID) : void
- ⊞ Append(IReadOnlyList<Guid> globalIDs) : void
- ⊞ ApplyTemplateLayouts() : void
- ⊞ ClearFlags(NetworkDiagramFlagType flagType) : void
- ⊞ ConsistencyState() : NetworkDiagramConsistencyState
- ⊞ Delete() : void
- ⊞ FindDiagramFeatures(IReadOnlyList<Guid> globalIDs, bool includeAggregations) : IReadOnlyList<Guid>
- ⊞ FindNetworkFeatures(IReadOnlyList<Guid> globalIDs, bool includeAggregations) : IReadOnlyList<Guid>
- ⊞ GetAggregations() : IReadOnlyList<DiagramAggregation>
- ⊞ GetDiagramDataset() : DiagramDataset
- ⊞ GetDiagramInfo() : NetworkDiagramInfo
- ⊞ GetDiagramTemplate() : DiagramTemplate
- ⊞ GetFlags(NetworkDiagramFlagType flagType) : IReadOnlyList<DiagramFlag>
- ⊞ GetName() : string
- ⊞ Overwrite(IReadOnlyList<Guid> globalIDs) : void
- ⊞ QueryDiagramFeatures(Envelope extentOfInterest, bool addContents, bool queryJunctions, bool queryEdges, bool queryContainers, out IReadOnlyList<DiagramFeature> features) : void
- ⊞ RemoveFlag(NetworkDiagramFlagType flagType, int diagramElementID) : void
- ⊞ SaveLayout(IEnumerable<DiagramJunction> junctions, IEnumerable<DiagramEdge> edges, IEnumerable<DiagramContainer> containers) : void
- ⊞ Store(string name, NetworkDiagramAccessType accessType, string tag) : void
- ⊞ Update() : void

NetworkDiagram: Retrieve diagram features

```
QueryDiagramFeatures(Envelope extentOfInterest, bool addContents, bool queryEdges, bool queryJunctions, bool queryContainers, out IReadOnlyList<DiagramJunction> junctions, out IReadOnlyList<DiagramEdge> edges, out IReadOnlyList<DiagramContainer> containers): void
```

```
FindDiagramFeatures(IReadOnlyList<Guid> globalIDs, bool includeAggregations): IReadOnlyList<Guid>
```

```
GetAggregations(): IReadOnlyList<DiagramAggregation>
```

■ These routines retrieve diagram features

- QueryDiagramFeatures retrieves the set of edges, junctions and/or containers represented in the diagram which extent intersects a given envelope.

Notes:

- #1: Specifying a null envelope allows you to retrieve all the edges, junctions and/or containers of the diagram.
- #2: Running with addContents = false (default) returns diagram features which strictly intersect the specified envelope, while running with addContents = true enlarges the “searching” envelope to include the extent of any container that are partially within the initial envelope.
- FindDiagramFeatures returns a list of GUIDs corresponding to all features in the diagram that are associated with a set of utility network feature GUIDs, those features being clearly represented in the diagram (includeAggregations=true | false) and/or aggregated (includeAggregations= true); that is, collapsed or reduced
- GetAggregations returns details on all the diagram features that are aggregated in the diagram; that is, reduced or collapsed in the diagram

NetworkDiagram: Retrieve related utility network features

```
FindNetworkFeatures(IReadOnlyList<Guid> globalIDs, bool includeAggregations):  
    IReadOnlyList<Guid>
```

- FindNetworkFeatures returns a list of GUIDs corresponding to all utility network features that are associated with a set of diagram feature GUIDs whether those diagram features are aggregated or not (includeAggregations= true | false)

NetworkDiagram: Lay out the diagram content

```
SaveLayout(IEnumerable<DiagramJunction> junctions, IEnumerable<DiagramEdge> edges,  
IEnumerable<DiagramContainer> containers) : void
```

```
ApplyTemplateLayouts() : void
```

- Use the SaveLayout to save any custom layout you may have coded by editing the diagram junctions, edges and containers geometry field (Shape)
- The ApplyTemplateLayouts routine applies the list of the layout algorithms currently configured on the template the input network diagram layer is based on.

NetworkDiagram: Manage flags on network diagrams

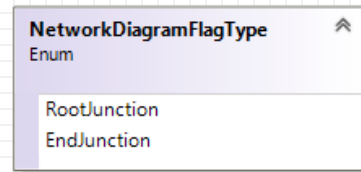
`GetFlags(NetworkDiagramFlagType flagType) : IReadOnlyList<DiagramFlag>`

`AddFlag(NetworkDiagramFlagType flagType, int diagramElementID) : void`

`RemoveFlag(NetworkDiagramFlagType flagType, int diagramElementID) : void`

`ClearFlags(NetworkDiagramFlagType flagType): void`

- The GetFlags routine retrieves all of the root or end diagram flags (depending on the specified flagType).
- The AddFlag and RemoveFlag routines flags or un-flags a particular diagram element. The ClearFlags clear all the root or end flags set up on the diagram (depending on the specified flagType)
- The AddFlag, RemoveFlag and ClearFlags routines are transactional operations; that is, their calls must be wrapped in a transaction
 - Called from `Geodatabase.ApplyEdits()`



NOTE: Root and End Junctions are specific junctions recognized by certain algorithm layouts that process the resulting diagram layout accordingly

DiagramFlag

ElementID : int

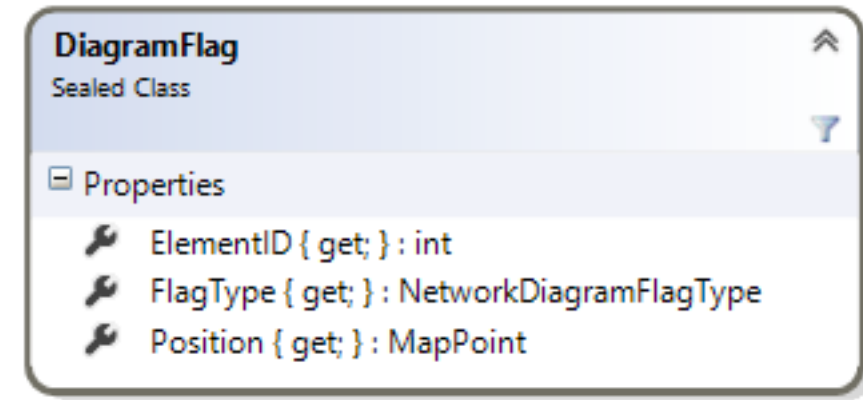
- The diagram element ID for the flag

FlagType : NetworkDiagramFlagType

- Type of flag

Position: MapPoint

- Location on the diagram



NetworkDiagram: Update diagram content

`Update(): void`

- Updating a diagram synchronizes a diagram based on the latest network topology- it incorporates any changes that were made to the features since the last update.
- The subnetwork 'system' diagrams cannot be updated using this routine. These diagrams are updated by calling `UtilityNetwork.UpdateSubnetwork`
- The Update routine is a transactional operation; that is, its calls must be wrapped in a transaction
 - Called from `Geodatabase.ApplyEdits()`

NetworkDiagram: Append features to a diagram

`Append(IReadOnlyList<Guid> globalIDs) : void`

- Appending features to a network diagram allows you to add any network features you missed at its generation or that didn't exist at the time the diagram was generated.
- No features can be appended to any subnetwork 'system' diagrams using this routine. These diagrams are managed by calling `UtilityNetwork.UpdateSubnetwork()`.
- These routines are transactional operations; that is, their calls must be wrapped in a transaction
 - Called from `Geodatabase.ApplyEdits()`

NetworkDiagram: Overwrite diagram content

`Overwrite(IReadOnlyList<Guid> globalIDs) : void`

- Overwriting a network diagram is a complete rebuilding of the network diagram content from the provided list of features.
- The subnetwork 'system' diagrams cannot be overwritten using this routine. These diagrams are managed by calling `UtilityNetwork.UpdateSubnetwork()`.
- These routines are transactional operations; that is, their calls must be wrapped in a transaction
 - Called from `Geodatabase.ApplyEdits()`

NetworkDiagram: Accessing diagram properties

`GetName() : string`

`GetDiagramDataset() : DiagramDataset`

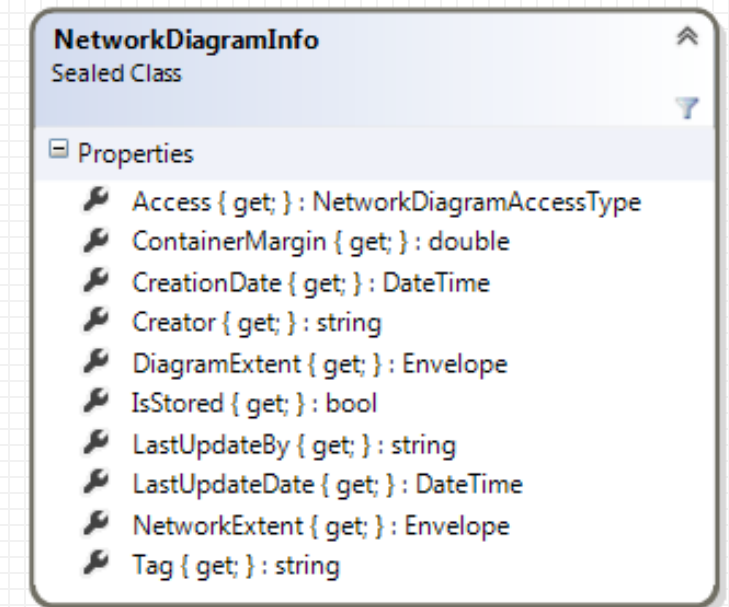
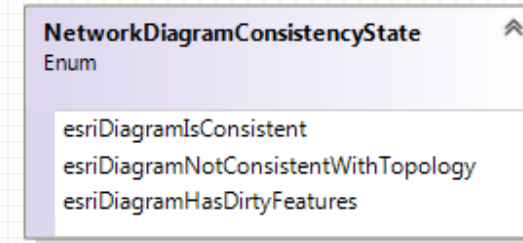
`GetDiagramTemplate() : DiagramTemplate`

`ConsistencyState(): NetworkDiagramConsistencyState`

`GetDiagramInfo(): NetworkDiagramInfo`

■ Those routines allow access to diagram properties

- The diagram name,
- Its diagram dataset,
- The template it is based on,
- The diagram consistency state,
- All info related to the network diagram, [NetworkDiagramInfo](#)—for example, whether it is stored or not, its access level, its tags, when and by whom it has been created/last updated, its extent, etc.



NetworkDiagram: Delete and store

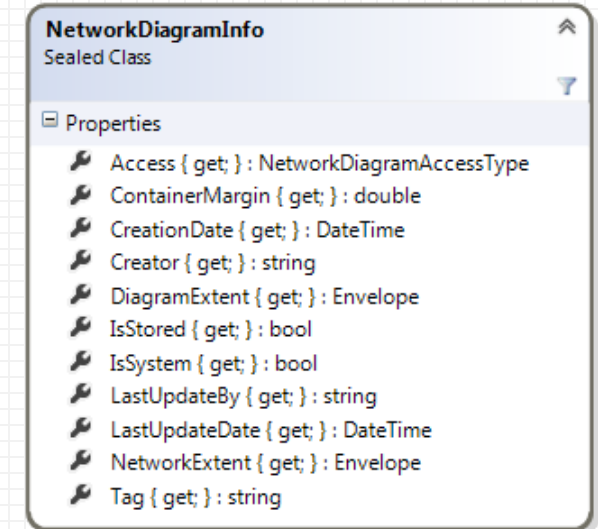
`Store(string name, NetworkDiagramAccessType accessType, string tag) : void`

`Delete() : void`

- The Store routine persists a temporary network diagram in the database. The diagram name, access level, and tags which are specified at the storing can be altered afterwards using the Set Diagram Properties GP tool.
- The Delete routine cannot process subnetwork 'system' diagrams.
- These routines are transactional operations; that is, their calls must be wrapped in a transaction
 - Called from `Geodatabase.ApplyEdits()`

NetworkDiagramInfo class

- All the info related to a network diagram can be obtained using `NetworkDiagram.GetDiagramInfo(): NetworkDiagramInfo`
- The NetworkDiagramInfo class provides a set of read-only properties which allow you to know:
 - the access level of the network diagram, `Access`
 - the internal margin of its container polygon, `ContainerMargin`
 - whether it is stored or not, `IsStored`
 - whether it is a subnetwork 'system' diagram or not, `IsSystem`
 - its tags, `Tag`
 - when and by whom it has been created/last updated, `CreationDate`, `Creator`, `LastUpdateDate`, `LastUpdateBy`
 - its extents, `DiagramExtent`, `NetworkExtent`.



NOTE: The DiagramExtent is the extent of the diagram itself.

The NetworkExtent is the extent formed by the set of utility network features that participated in the diagram building whether those features are represented in the diagram or not.

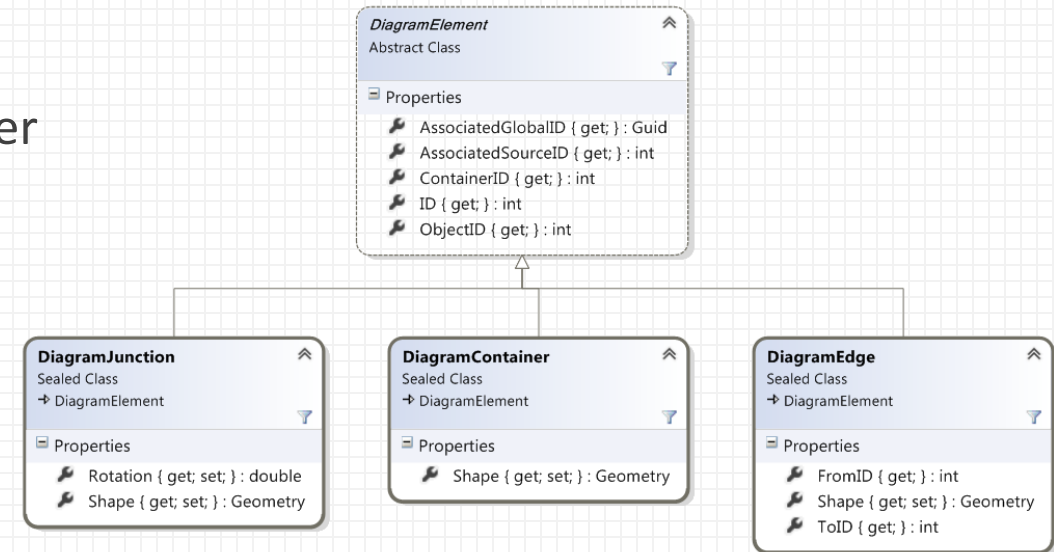
DiagramJunction, DiagramEdge and DiagramContainer classes

- A network diagram consists in a set of diagram elements which are either junctions, edges, or containers.

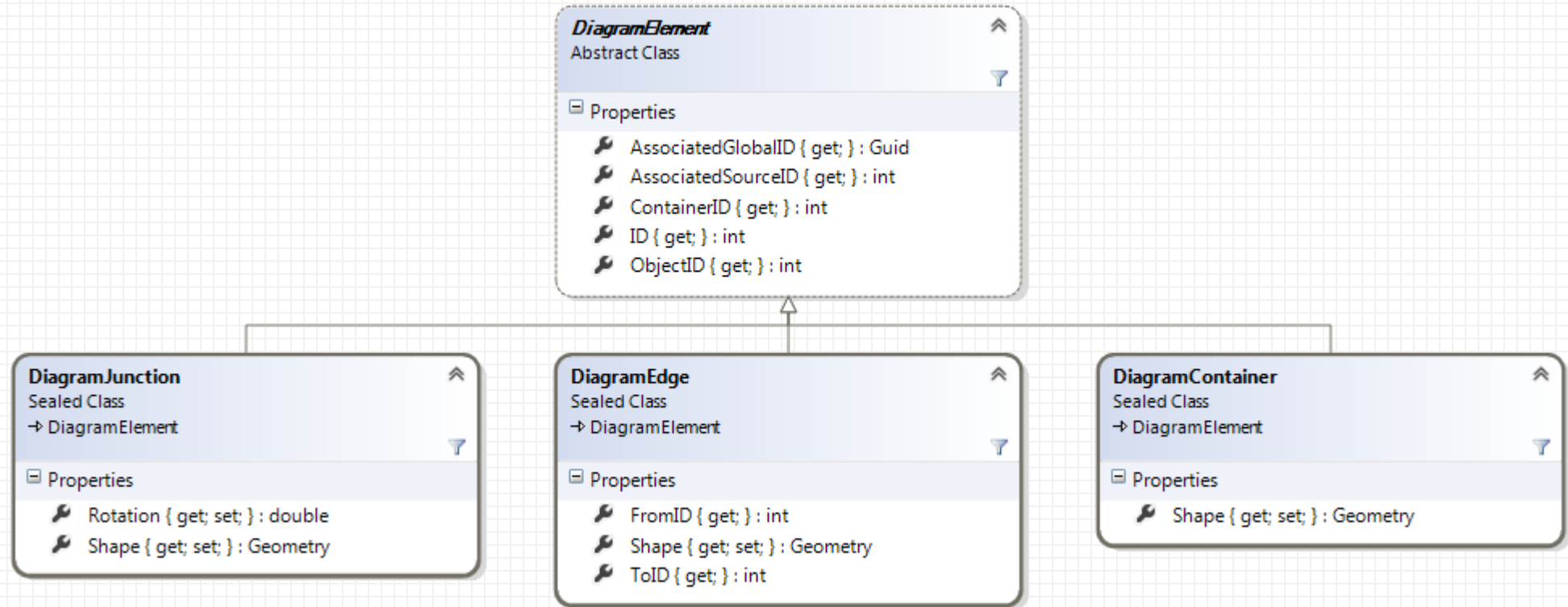
- The three DiagramJunction, DiagramEdge and DiagramContainer classes derive from the same class, DiagramElement, which is a value class that stores information common to all diagram elements:

AssociatedGlobalID : Guid
AssociatedSourceID() : int
ContainerID() : int
ID(): int

Those read-only properties retrieve the utility network feature/utility network element the diagram element is associated with



DiagramJunction, DiagramEdge and DiagramContainer classes

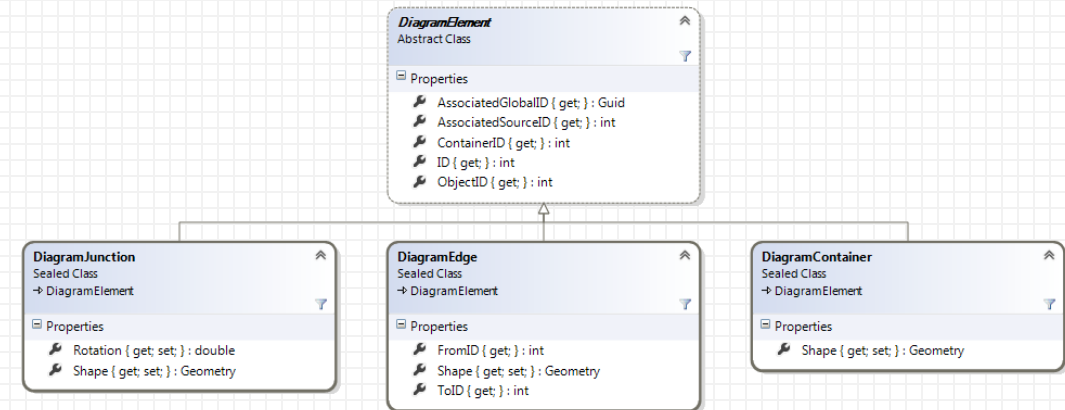


DiagramJunction, DiagramEdge and DiagramContainer classes

- The set of DiagramJunction, DiagramEdge and DiagramContainer that compose a diagram are obtained calling `NetworkDiagram.QueryDiagramFeatures`

- The DiagramJunction, DiagramEdge and DiagramContainer are classes that provide access to element geometry, `Shape`, which are read-write properties

- The DiagramEdge class also provide topology information, `FromID` and `ToID`, which are read-only properties.



DiagramAggregation class

- The DiagramAggregation class is a value class that stores aggregation information related to an utility network feature which is not represented in the diagram since it is aggregated under another diagram element; that is, it is reduced or collapsed in the diagram:

`AssociatedGlobalID : GUID`

`AssociatedSourceID() : int`

`DiagramElementID() : int`

Those read-only properties retrieve the aggregated utility network feature and the ID of the diagram element under which it has been reduced or collapsed.

- The set of DiagramAggregations related to a diagram are obtained by calling `NetworkDiagram.GetAggregations`

DiagramAggregation
Sealed Class

Properties

⚙

 AggregationType { get; } : NetworkDiagramAggregationType

⚙

 AssociatedGlobalID { get; } : Guid

⚙

 AssociatedSourceID { get; } : int

⚙

 DiagramElementID { get; } : int

NetworkDiagramAggregationType
Enum

NoneAggregation

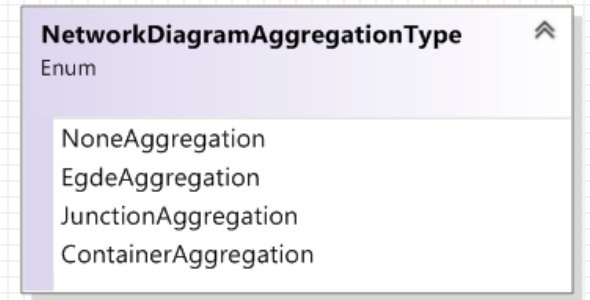
EdgeAggregation

JunctionAggregation

ContainerAggregation

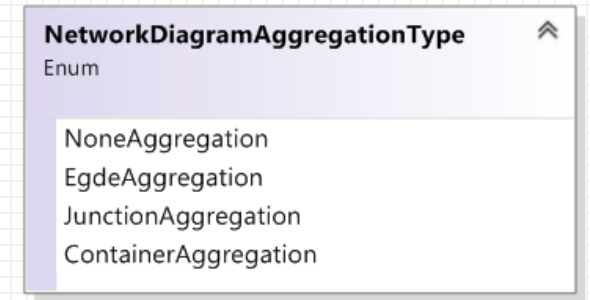
DiagramAggregation class

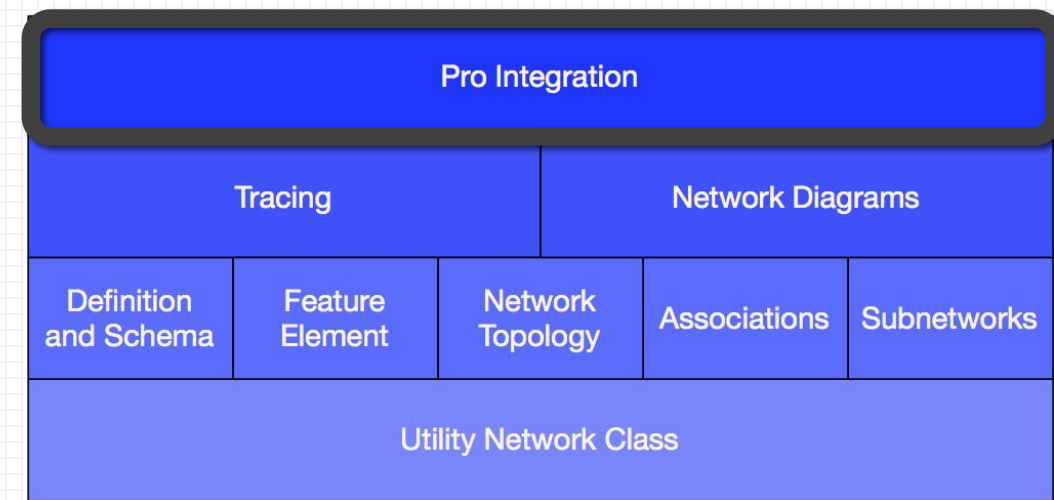
- There are 4 types of aggregations—[JunctionAggregation](#), [EdgeAggregation](#), [ContainerAggregation](#), and [NoneAggregation](#):
 - [JunctionAggregation](#)—for utility network features aggregated under a diagram junction. This diagram junction may correspond to:
 - An utility network “container” point or polygon represented as a point in the diagram; this diagram point is related to a set of utility network “content” features that have been collapsed in the network diagram.
 - A utility network junction point represented as a point in the diagram; this diagram point is related to a set of utility network features that have been reduced in the network diagram.
 - [EdgeAggregation](#)—for utility network features aggregated under a diagram edge. This diagram edge may correspond to:
 - An utility network “container” line represented as an edge in the diagram; this diagram edge is related to a set of utility network “content” features that have been collapsed in the network diagram.
 - A reduction edge diagram; this diagram edge is related to a set of utility network features that have been reduced in the network diagram.



DiagramAggregation class

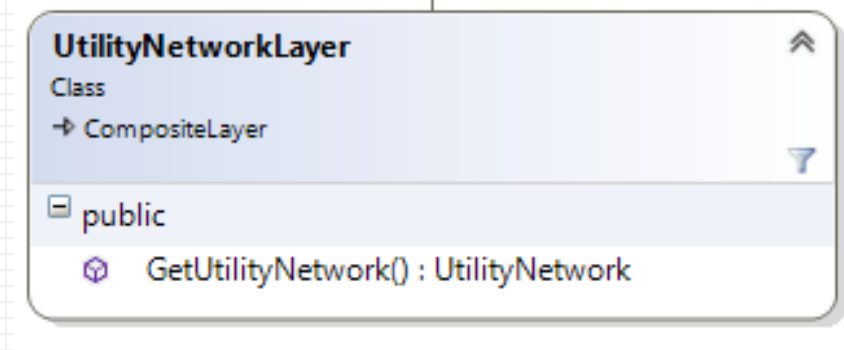
- There are 4 types of aggregations—[JunctionAggregation](#), [EdgeAggregation](#), [ContainerAggregation](#), and [NoneAggregation](#):
 - [ContainerAggregation](#)—for utility network features aggregated under a diagram polygon. This diagram polygon may correspond to:
 - An utility network “container” point or polygon represented as a polygon in the diagram; this diagram polygon is related to a set of utility network “content” features that have been collapsed in the network diagram.
 - [NoneAggregation](#)—for utility network features aggregated in the diagram under no diagram element
 - For example, unconnected system junctions are systematically reduced to nothing when running a Reduce Junction rule that processes unconnected junctions. In the same way, depending on the Reduce Junction rule settings, certain disconnected portions of the utility network may be reduced to nothing in the resulting diagram





Pro Integration

UtilityNetworkLayer



- The `UtilityNetworkLayer` class represents a utility network layer in a map
- This class resides in a mapping rather than a geodatabase namespace
 - Currently `ArcGIS.Desktop.Internal.Mapping`
 - Will change to `ArcGIS.Desktop.Mapping` for final release

`GetUtilityNetwork() : UtilityNetwork`

- Returns the `UtilityNetwork` class pointed at by this layer
- Used with ArcGIS Pro add-ins to obtain the underlying geodatabase object from the selected layer