Utility Network C# SDK

Beta 1

May 30, 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

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`
  - Network diagram items are included within `ArcGIS.Core.Internal.Data.UtilityNetwork.NetworkDiagrams`

- This will be changed to `ArcGIS.Core.Data.UtilityNetwork` before shipping the first release

Architectural Topics

- DML-only (Data Manipulation Language)
- Threading
- Garbage Collection
- Services-only

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 performed using Python
  - This is in alignment with the rest of the Geodatabase API
  - Python can 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

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 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

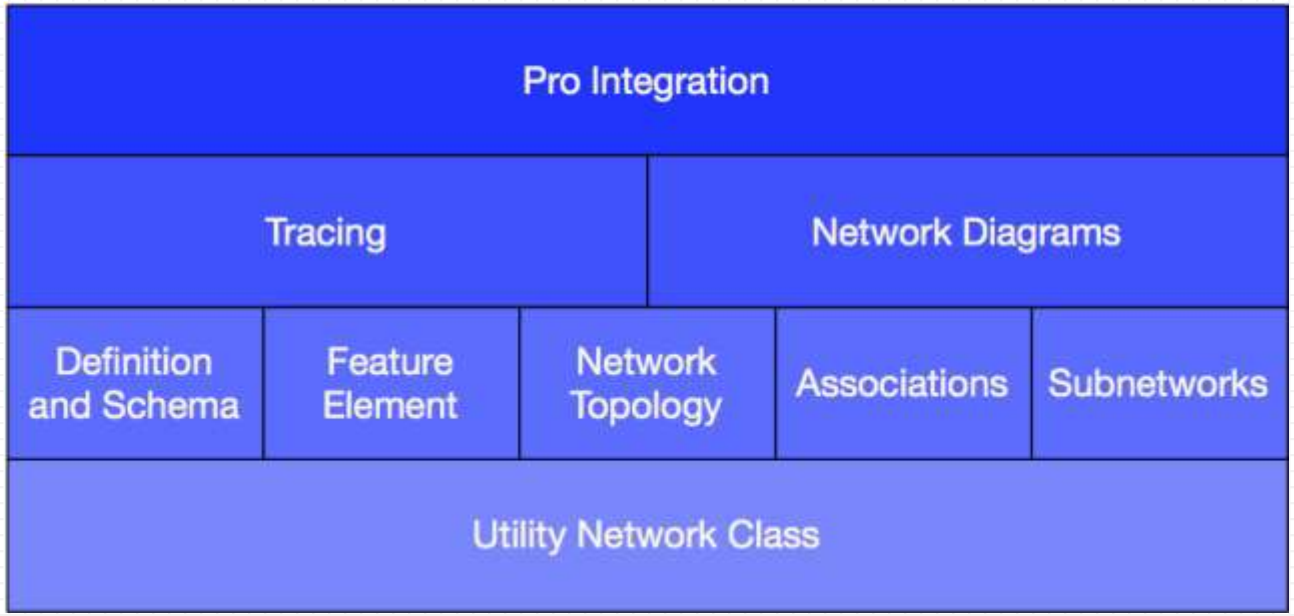
POST only

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.

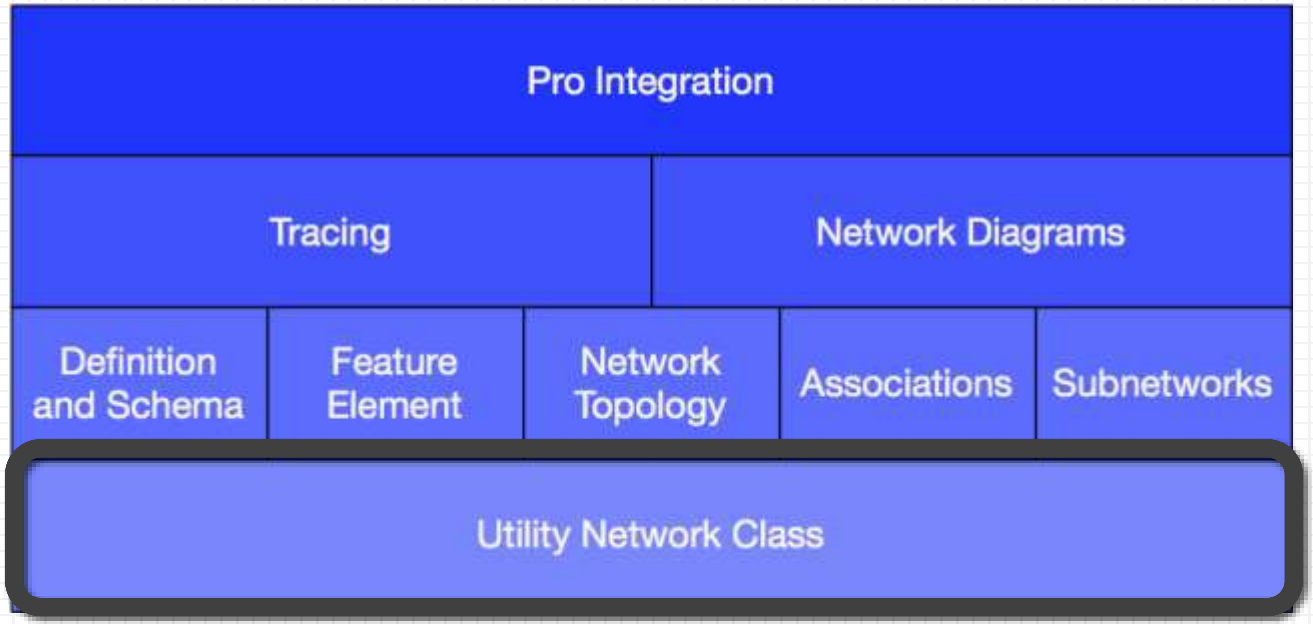
3. Organization of the Utility Network API

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



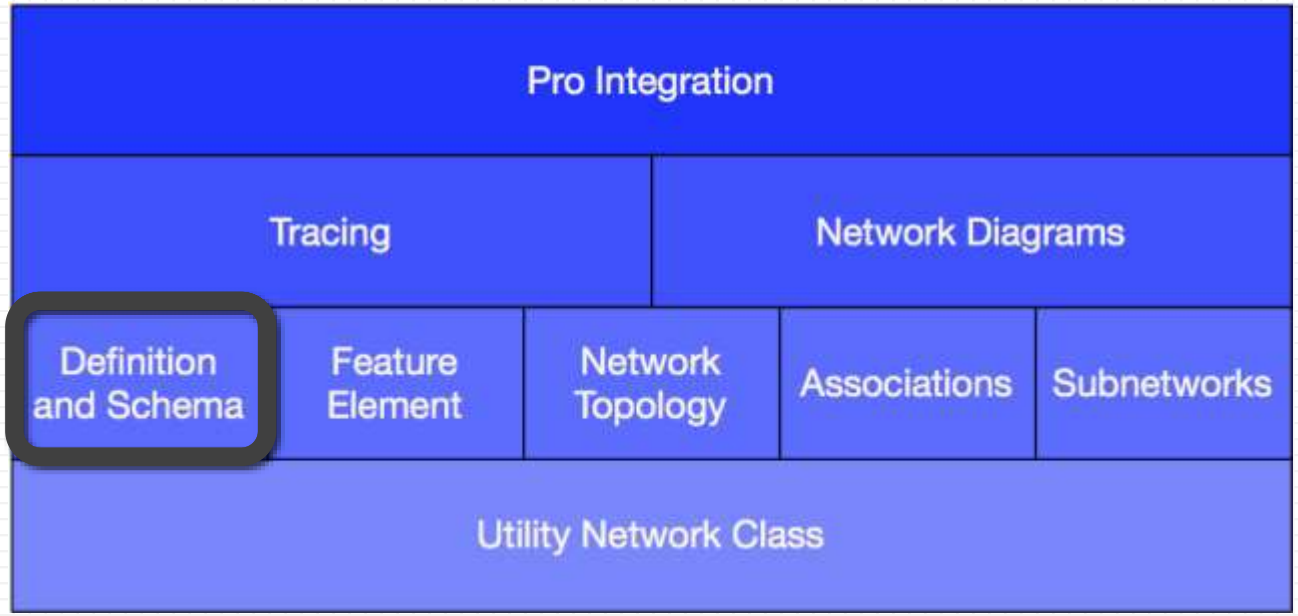
3. Organization of the Utility Network API

- **Utility Network Class** covers the root object that provides access to the utility network API



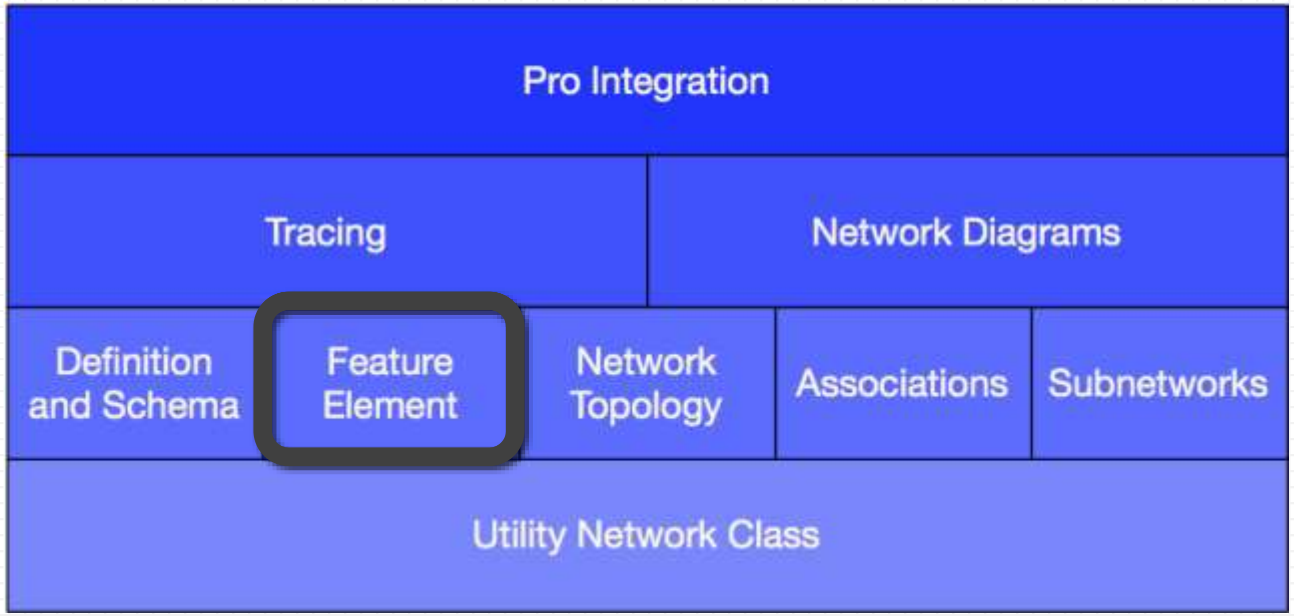
3. Organization of the Utility Network API

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



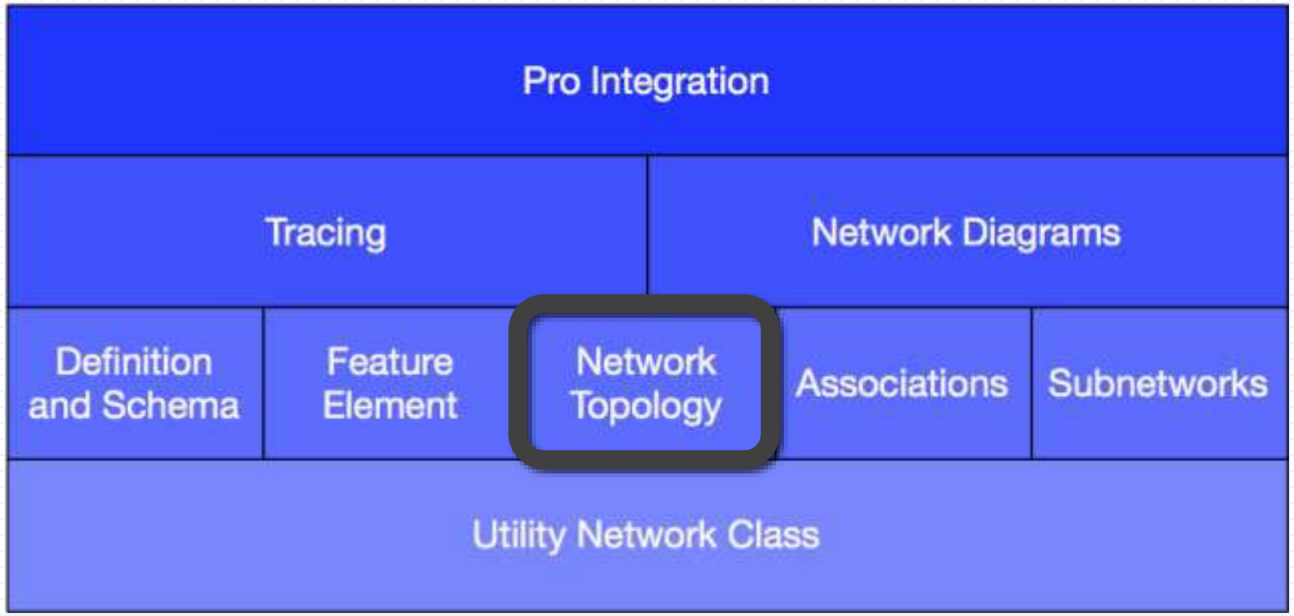
3. Organization of the Utility Network API

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



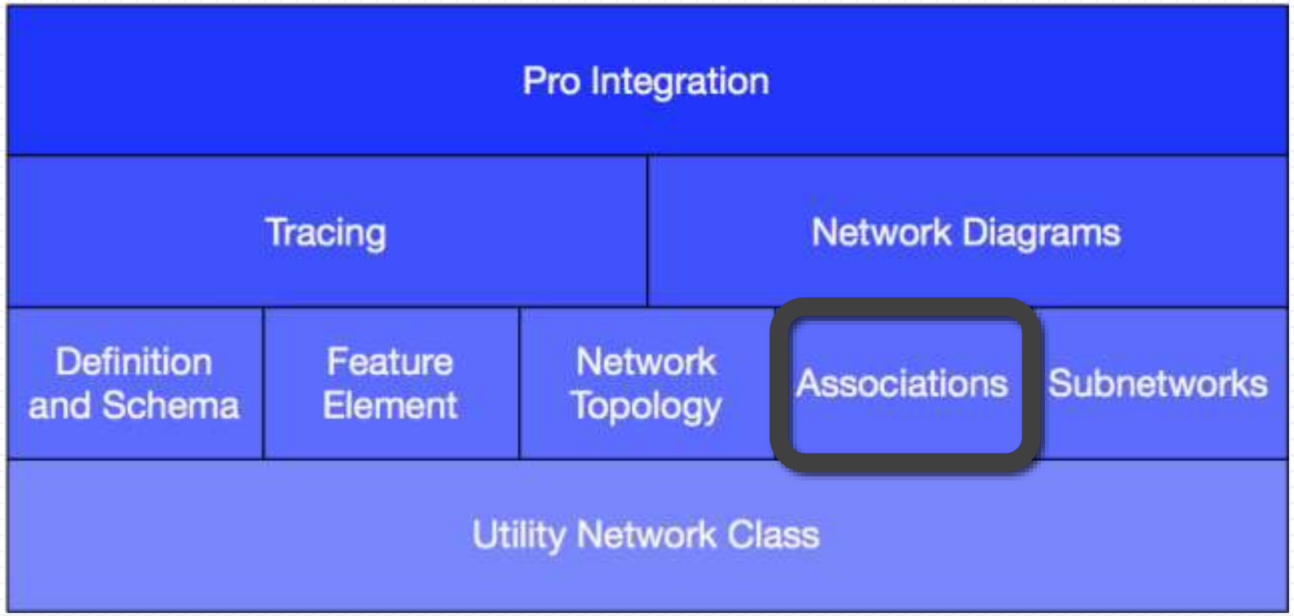
3. Organization of the Utility Network API

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



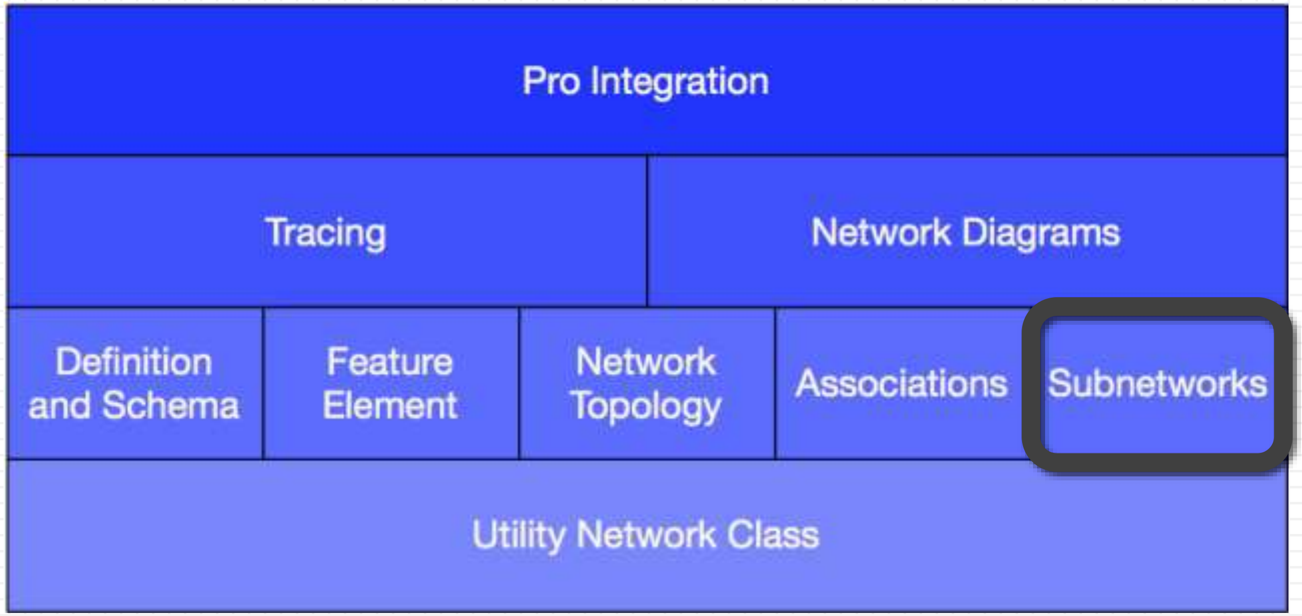
3. Organization of the Utility Network API

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



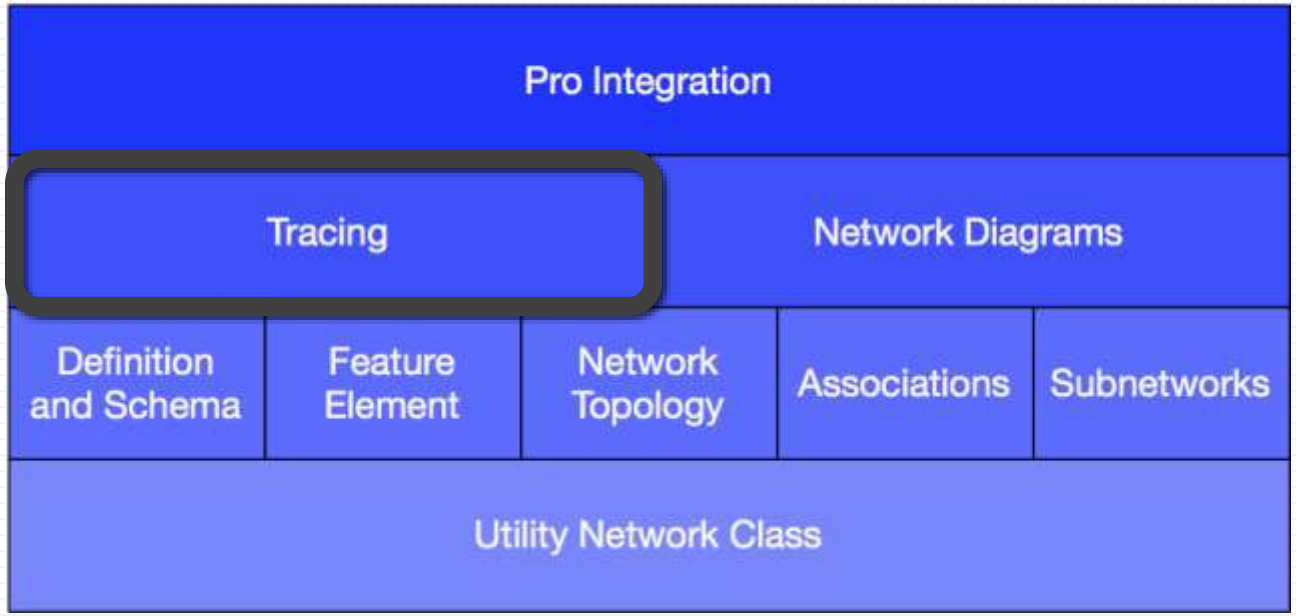
3. Organization of the Utility Network API

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



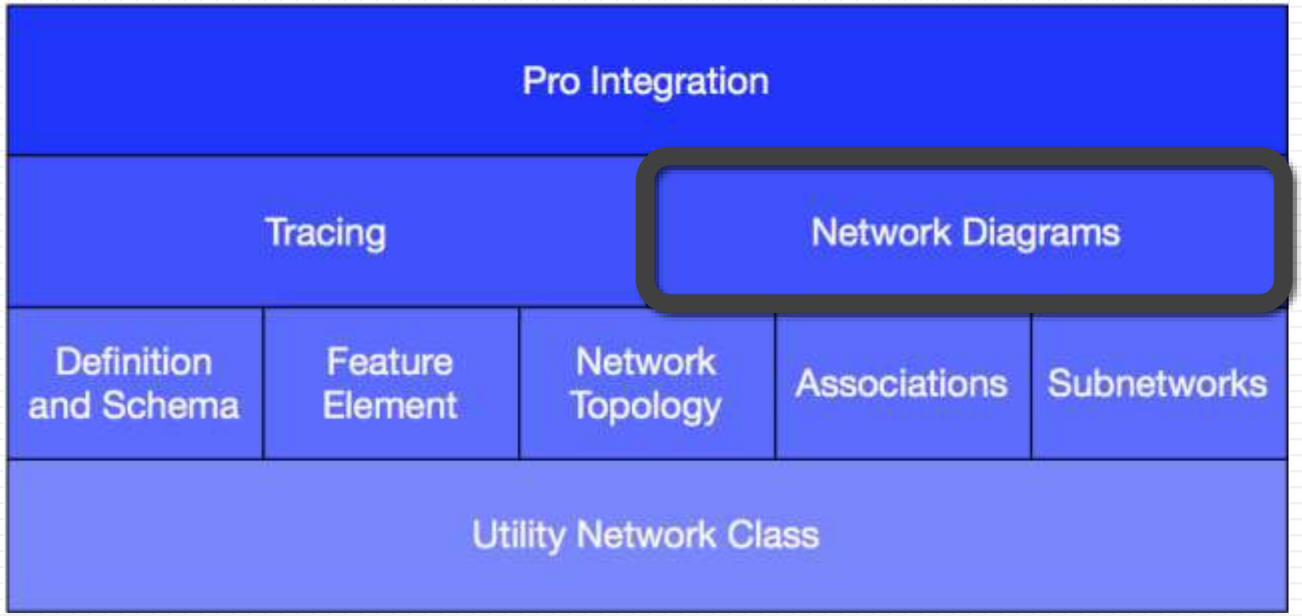
3. Organization of the Utility Network API

- **Tracing** provides tracing functionality



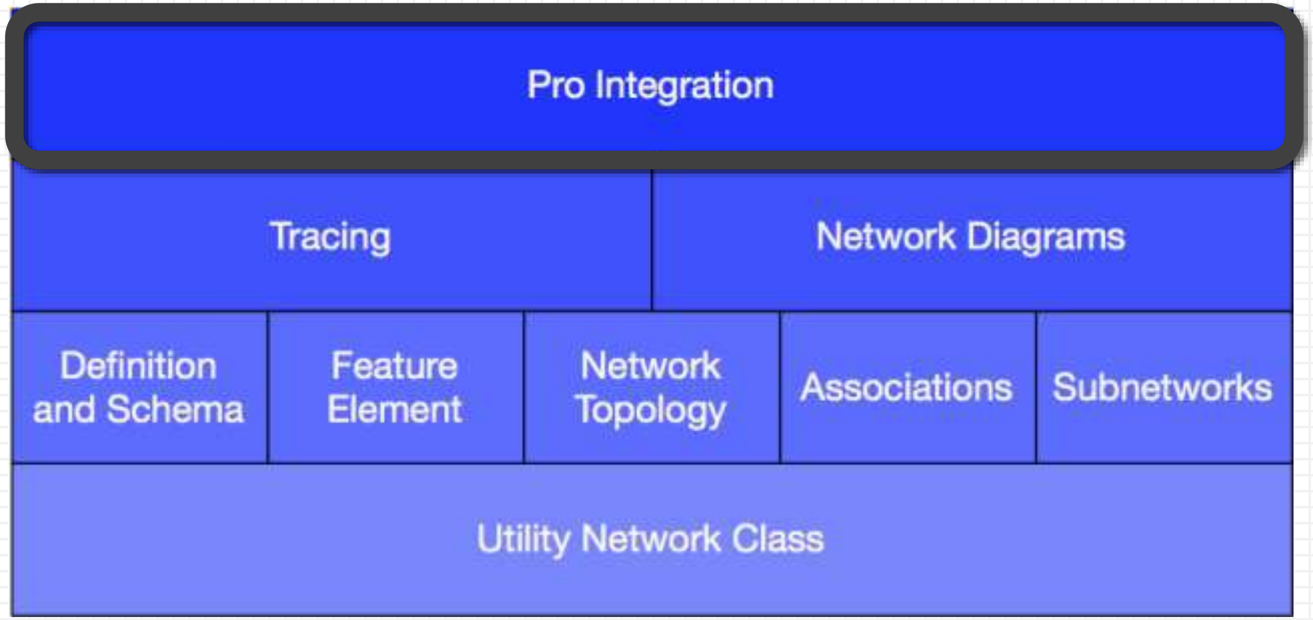
3. Organization of the Utility Network API

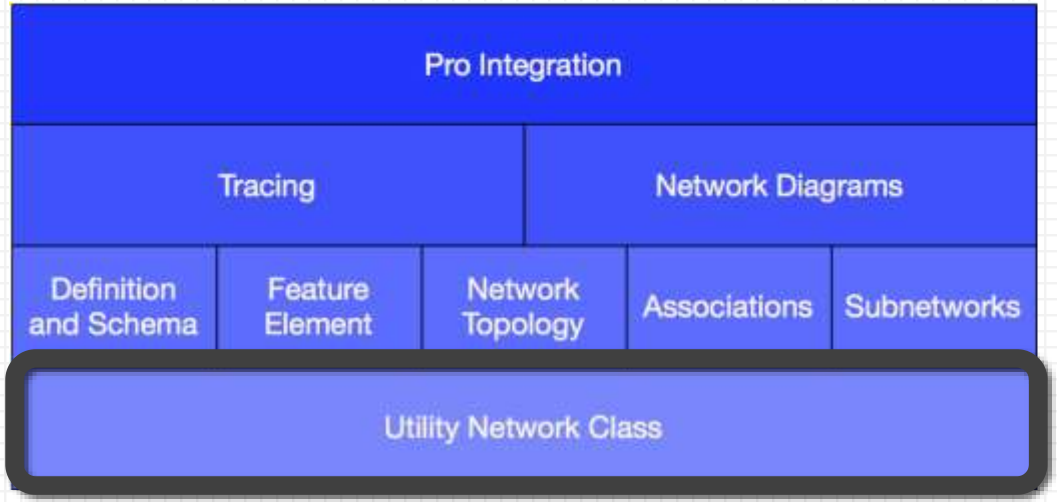
- **Network Diagrams** allows the developer to query and edit network diagrams



3. Organization of the Utility Network API

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

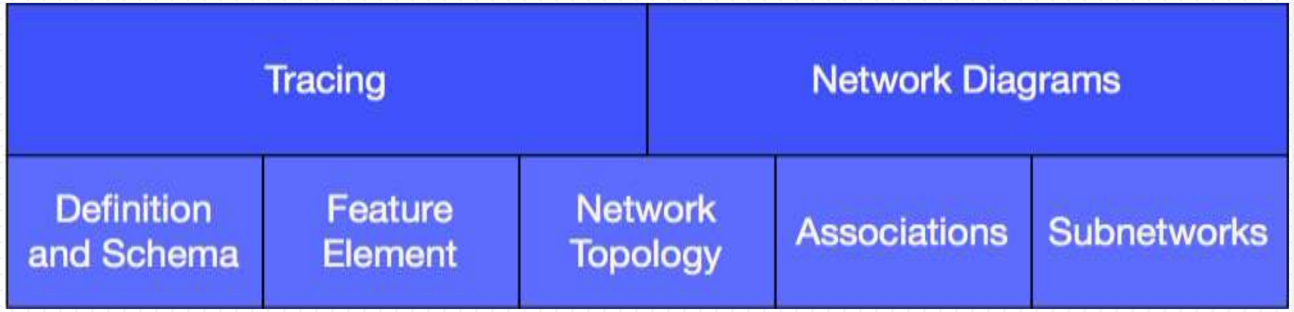




Utility Network Class

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 API



- 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`

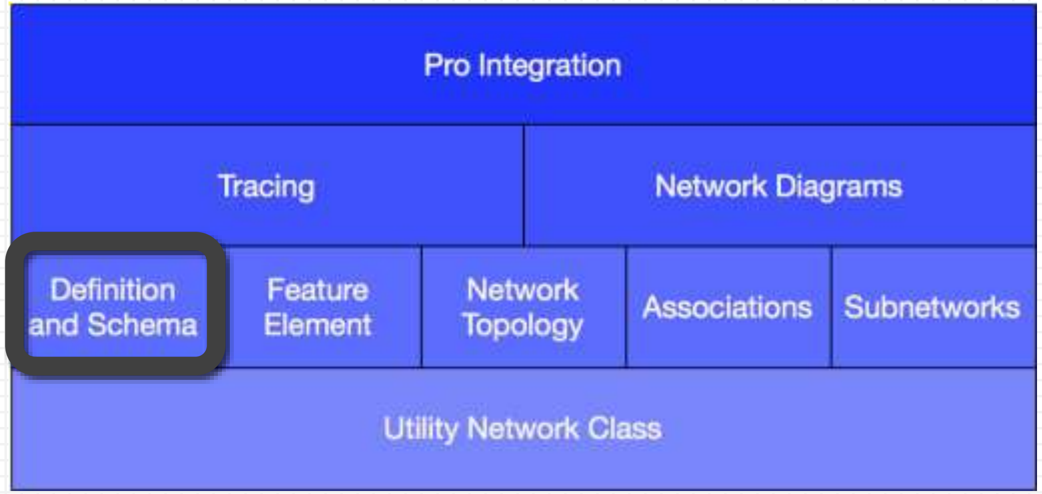
- This routine returns the extent of the feature classes within the network (same as other controller datasets)

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

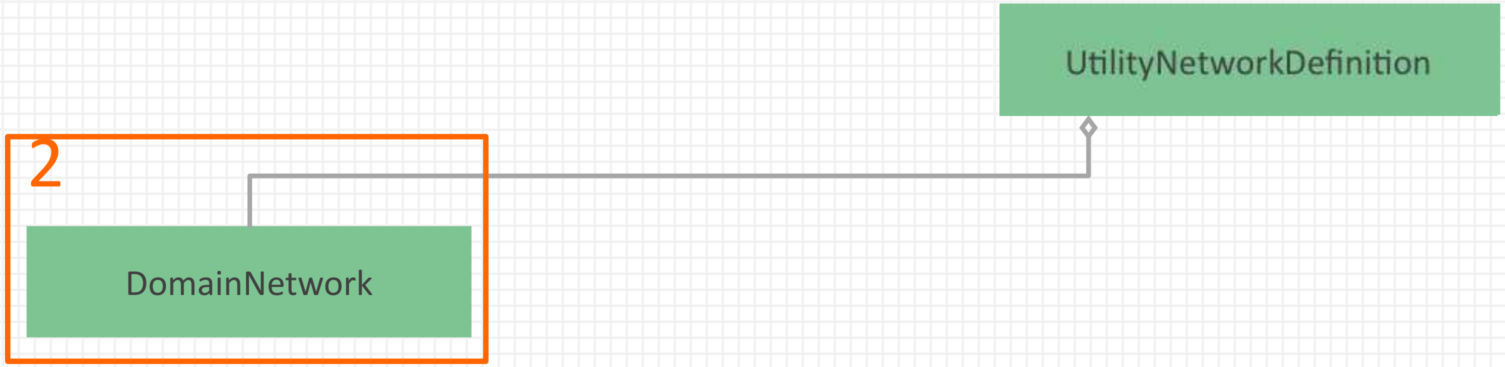
Definition and Schema

- 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 Utility network concepts section of the online help

Overview of Definition and Schema Classes

- The **utility network** itself is described by a `UtilityNetworkDefinition` class
- This class also serves as the central hub for the definition and schema classes

Overview of Definition and Schema Classes

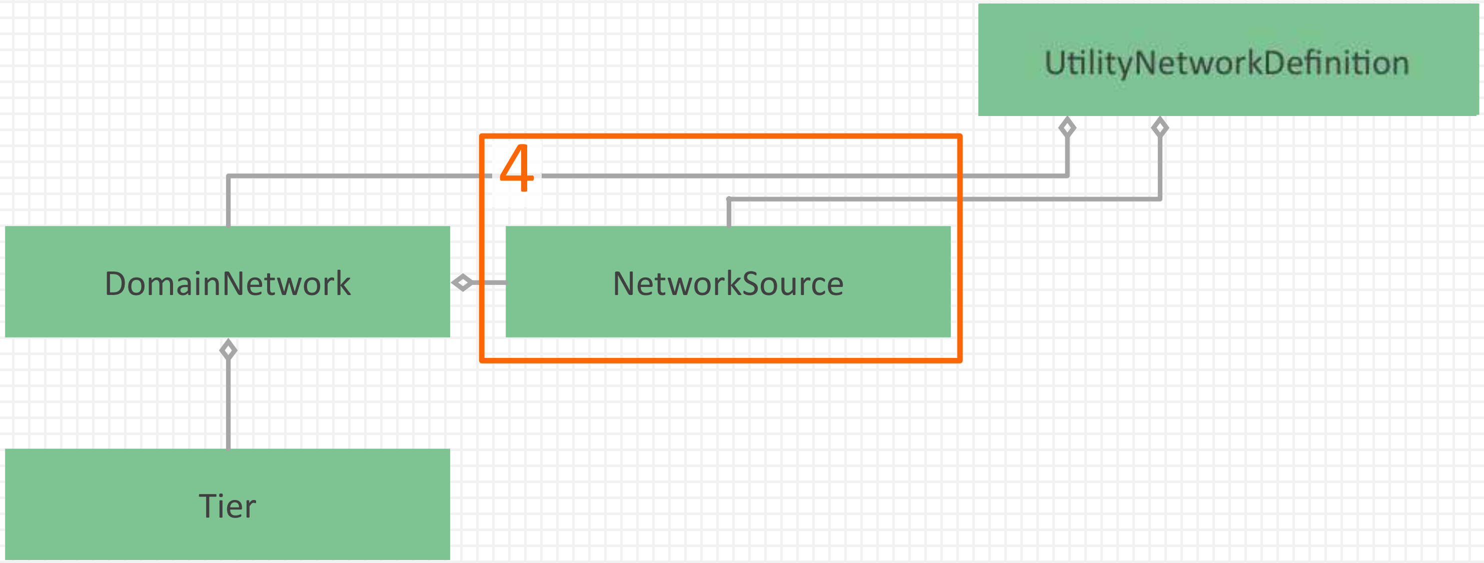


- Every utility network consists of one or more **domain networks**

Overview of Definition and Schema Classes

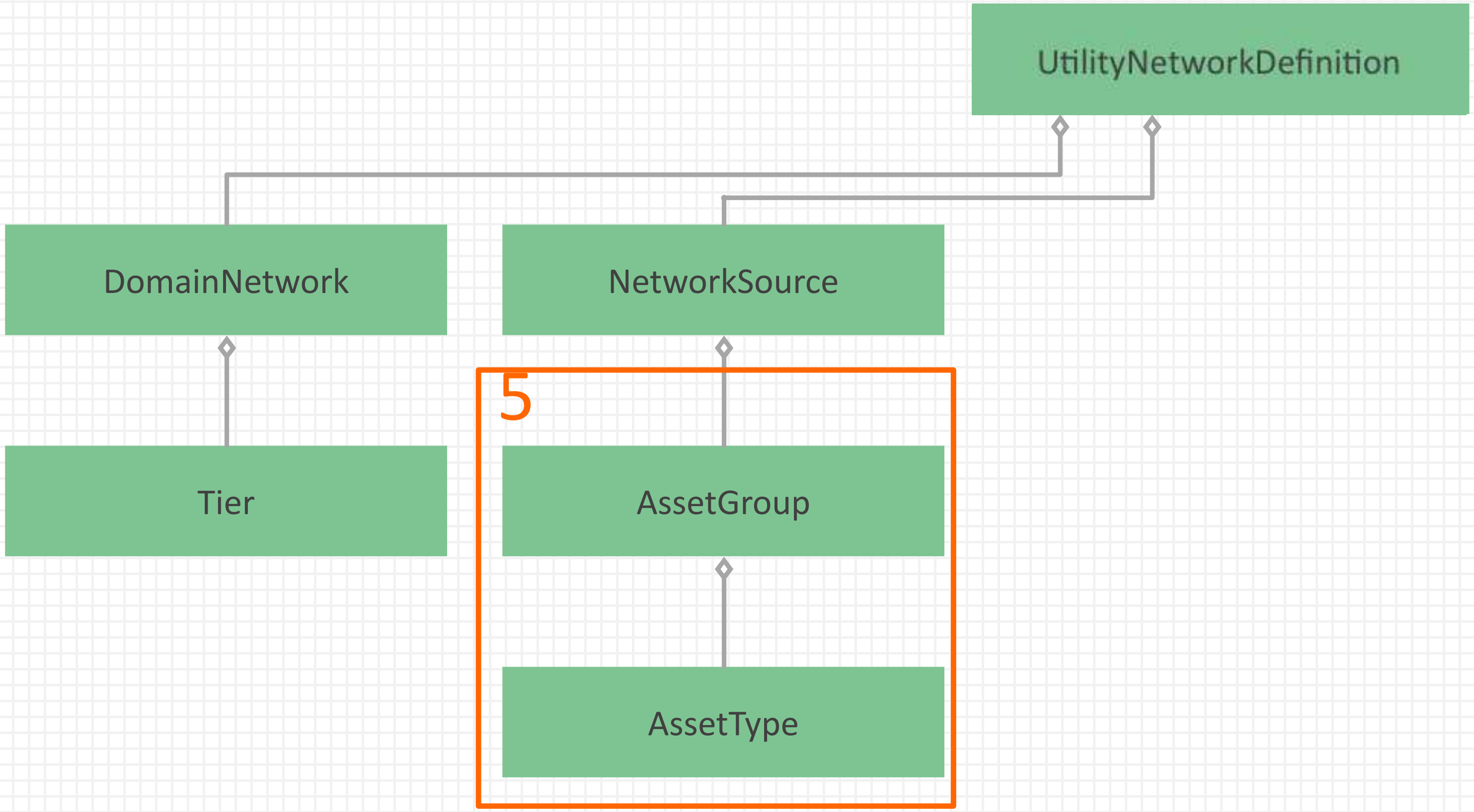
- Domain networks contain **tiers**

Overview of Definition and Schema Classes



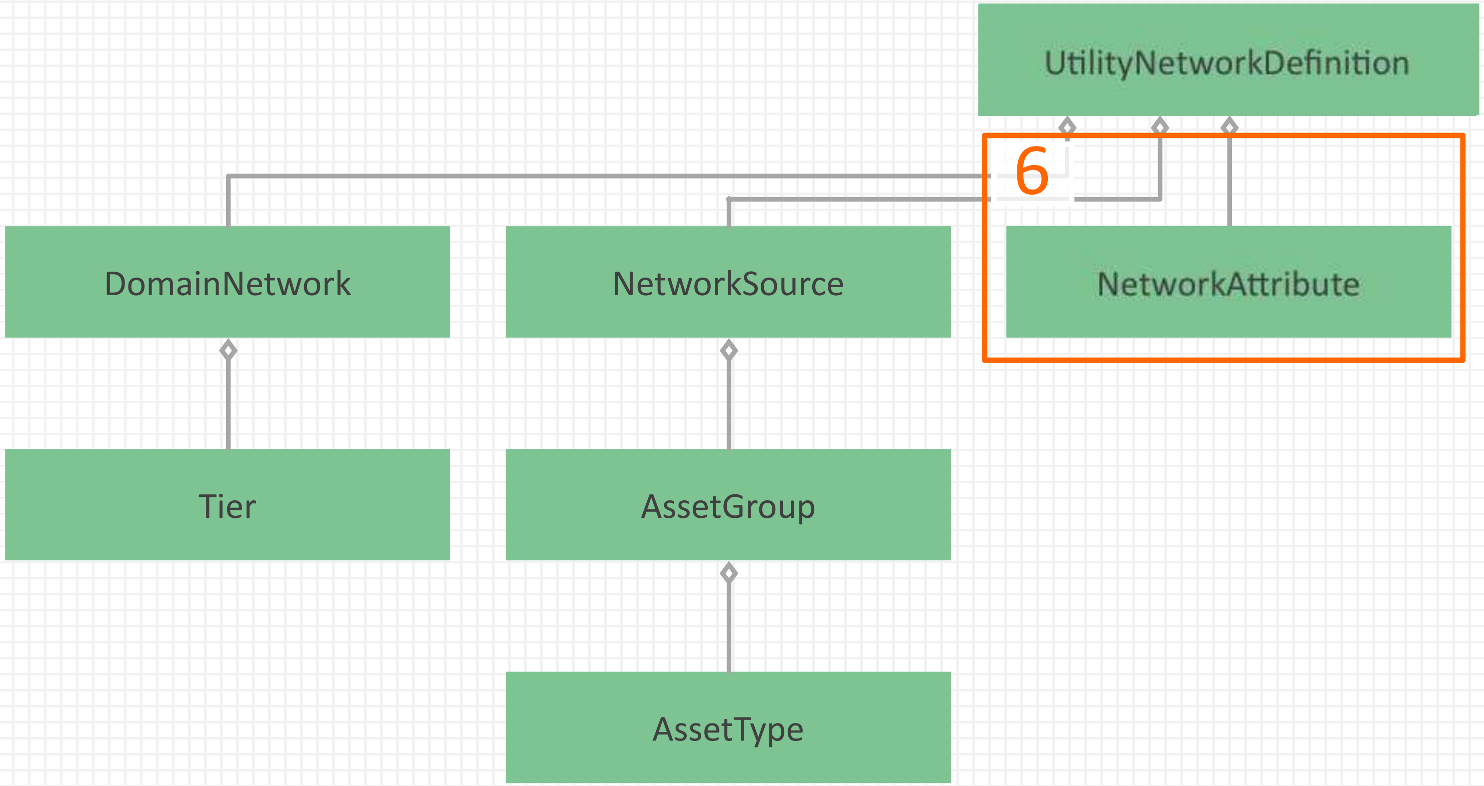
- Every utility network also contains a set of `NetworkSource` classes
- These classes describe the sources of data for the utility network (usually feature classes)

Overview of Definition and Schema Classes



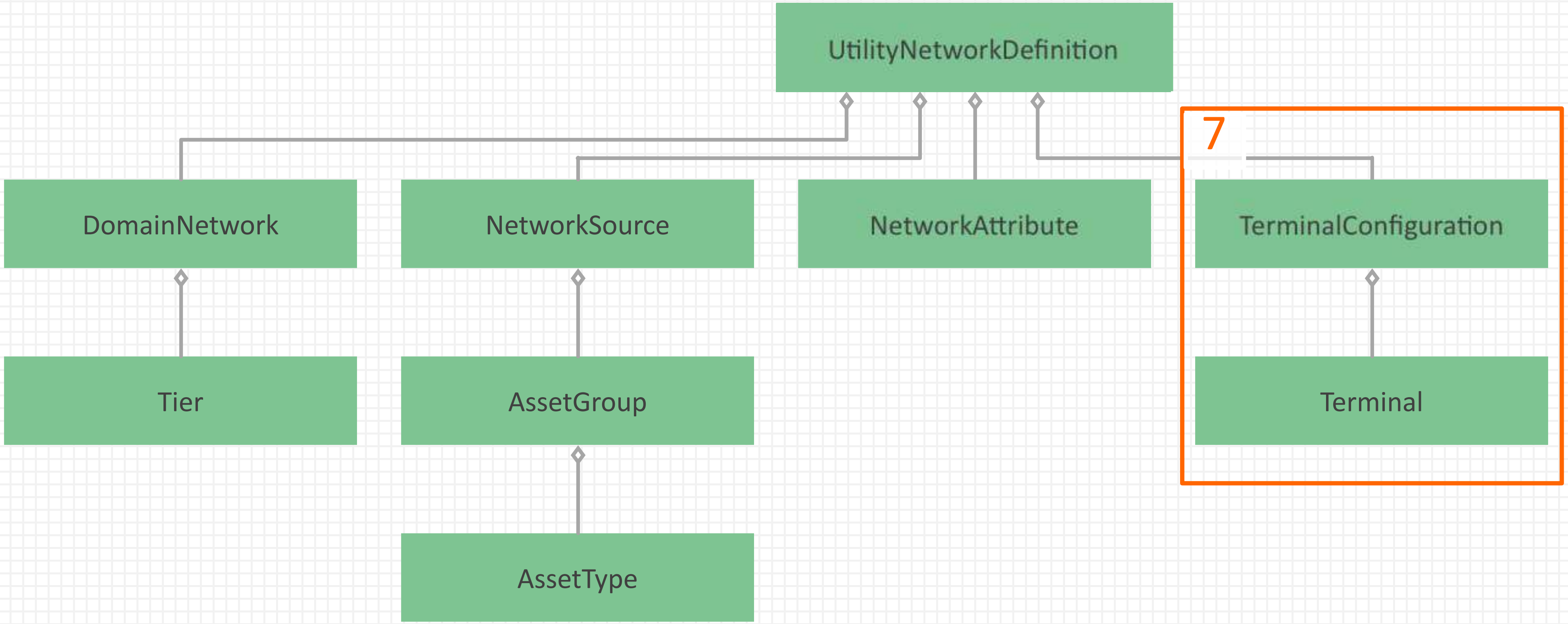
- Every feature class that serves as a network source contains **asset groups**
- Asset groups contain **asset types**

Overview of Definition and Schema Classes



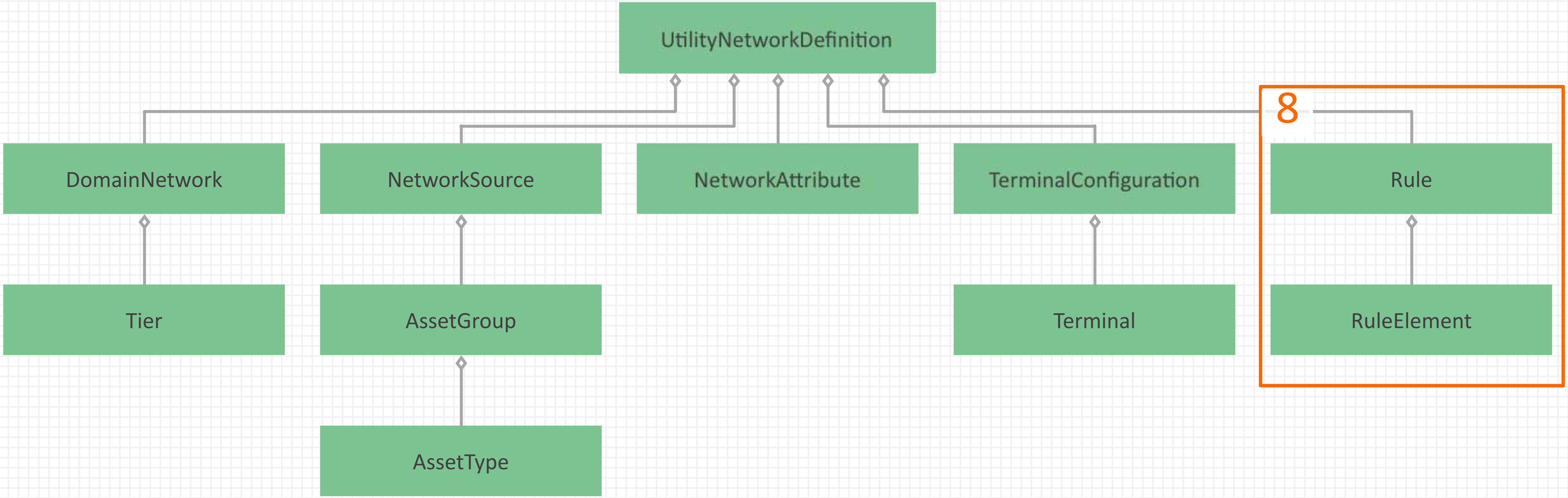
- Utility networks also contain a set of **network attributes**

Overview of Definition and Schema Classes



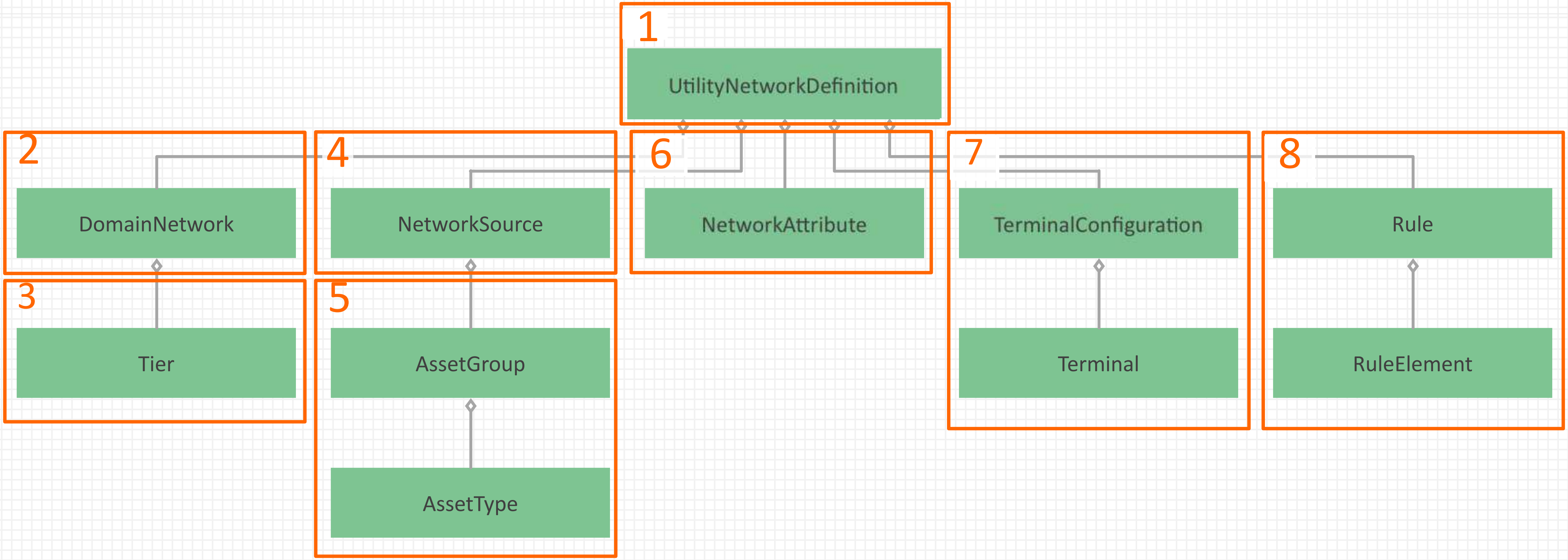
- Terminal configurations of a utility network are described by the `TerminalConfiguration` and `Terminal` classes

Overview of Definition and Schema Classes



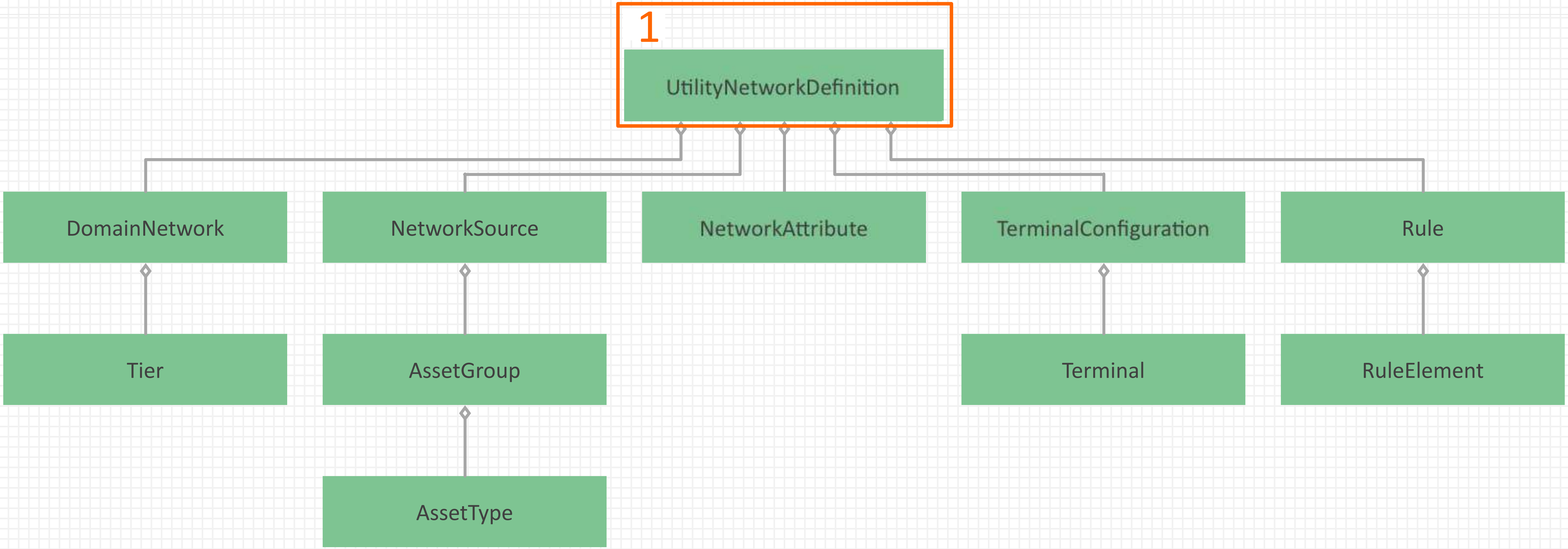
- Finally, the **rules** of a utility network are described by the `Rule` and `RuleElement` classes

Overview of Definition and Schema Classes



- We'll cover each of these parts in more detail

The UtilityNetworkDefinition Class



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

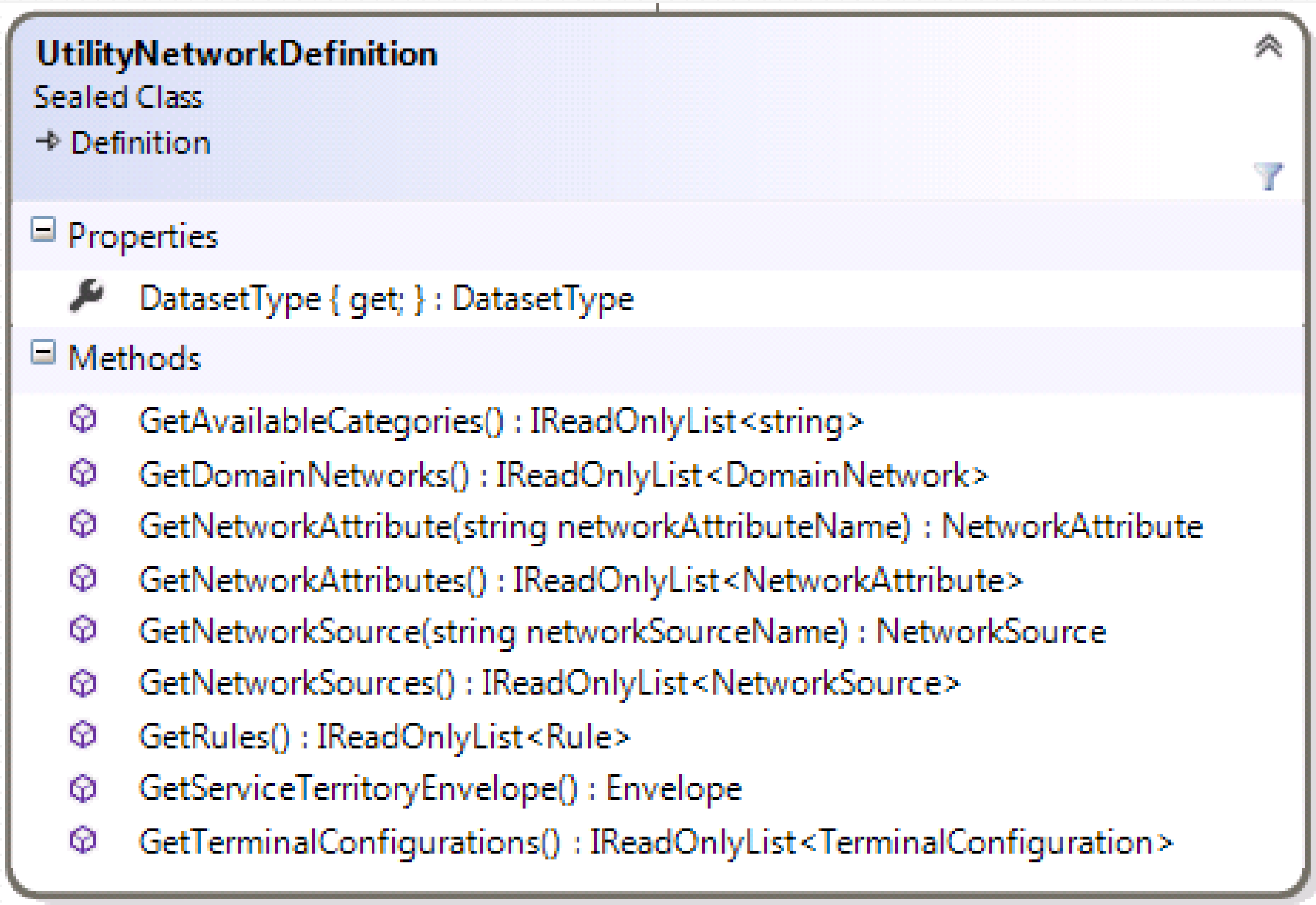
```
UtilityNetworkDefinition definition = geodatabase.GetDefinition<UtilityNetworkDefinition>("UtilityNetworkName");
```

- 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 Class Diagram



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

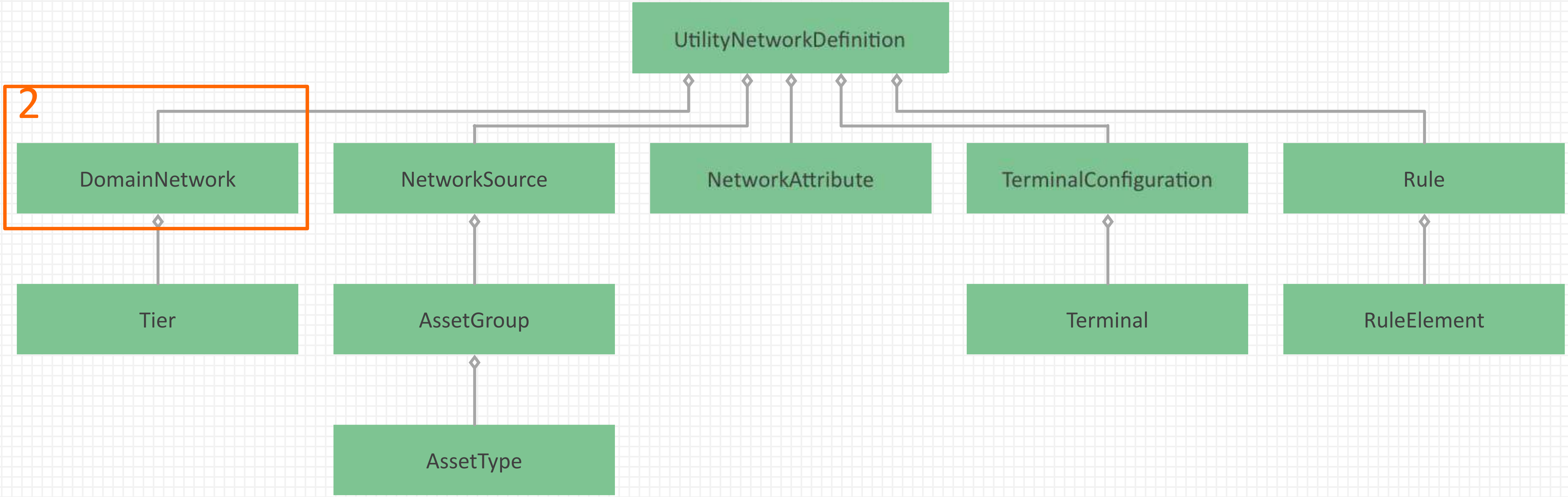
`GetServiceTerritoryEnvelope() : Envelope`

- 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() : IReadOnlyList<string>`

- Returns a list of all the categories that have been registered with the utility network
- Categories are tags that can be assigned to asset types

The DomainNetwork Class

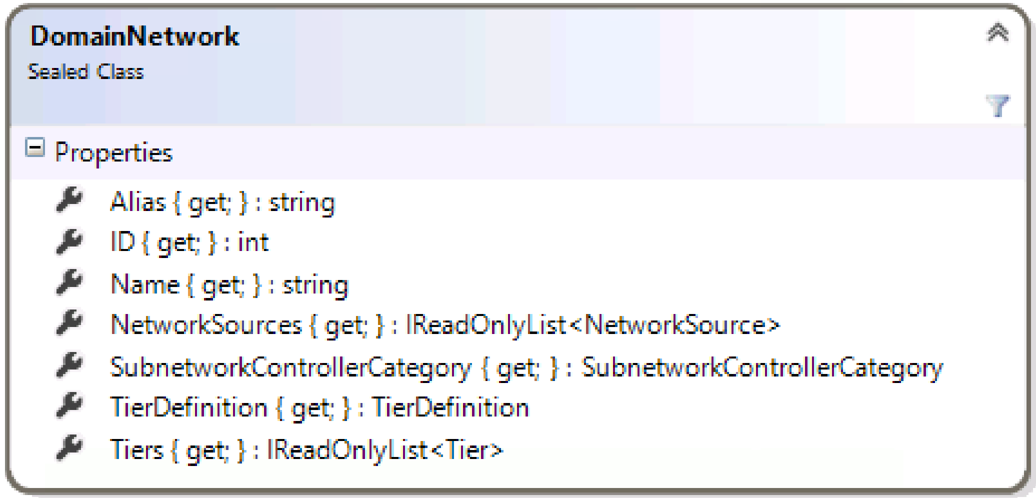


Domain Networks

- A domain network represents a type of utility service that an organization serves
  - Domain networks can represent different levels of the same utility resource such as distribution and transmission levels for gas, water, or electricity
  - Domain networks can also represent multiple types of services such as natural gas and electricity

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

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



DomainNetwork — General Routines

`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

DomainNetwork — Tier and Subnetwork Routines

`Tiers : IReadOnlyList<Tier>`

- Returns the tiers in this domain network

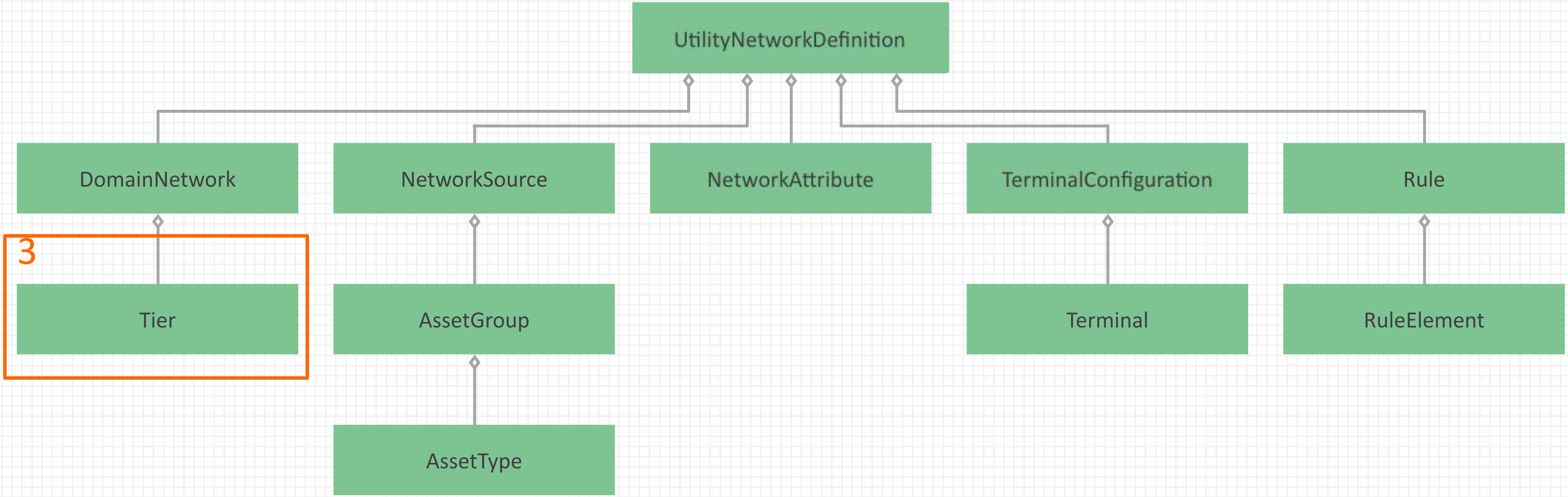
`SubnetworkControllerCategory :  SubnetworkControllerCategory`

- Returns the type of subnetwork controllers supported in this domain network
- Valid values in the `SubnetworkControllerCategory` enum are `Source` and `Sink`

`TierDefinition : TierDefinition`

- Returns the type of subnetworks supported in this domain network
- Values values in the `TierDefinition` enum are `Hierarchical` (typically used with pressure networks) and `Partitioned` (typically used with electrical networks)

The Tier Class

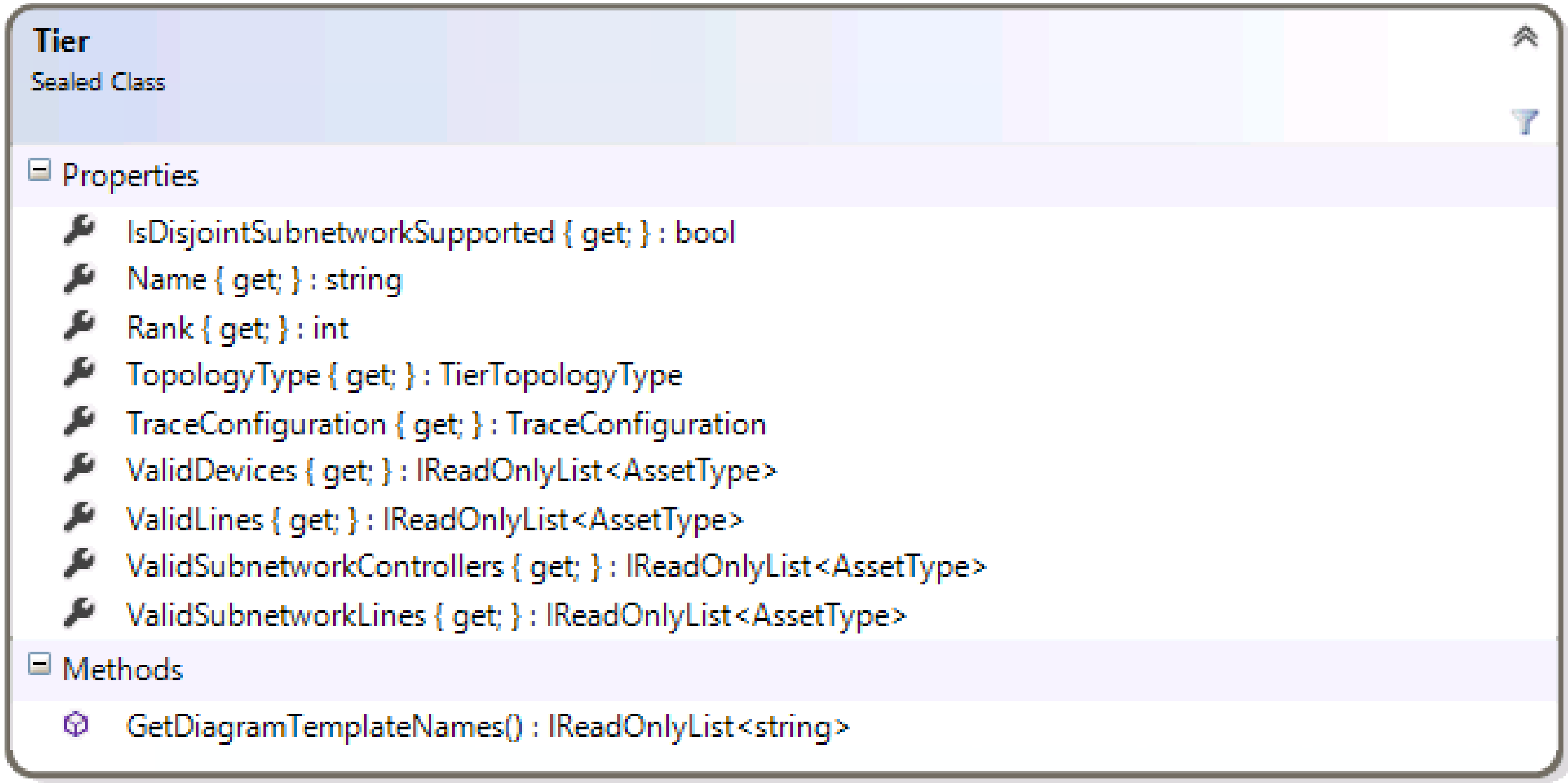


Tiers

- Tiers model the hierarchy of how the network delivers a resource such as natural gas, electricity, and water
- A tier typically represents a pressure or voltage level
  - For example, an electric distribution system can be subdivided into subtransmission, medium voltage, and low voltage levels and some types of analysis should be performed only within one of these hierarchical levels
- A tier can also represent parts of the network that can be isolated from one another such as for valve isolation zone in a pressure based system
- Tiers are useful because they allow you to constrain the valid feature types for each tier and they can also define the extent of network tracing analysis

Tiers

- The Domain Network class has a method that returns a set of tiers
  - `DomainNetwork.Tiers : IReadOnlyList<Tier>`



Code Snippet: Find the Medium Voltage Tier

```
UtilityNetworkDefinition definition = utilityNetwork.GetDefinition();
DomainNetwork domainNetwork =
  definition.GetDomainNetworks().First<DomainNetwork>(d => d.Name == "Electric Distribution");
IReadOnlyList<Tier> tiers = domainNetwork.Tiers;
Tier tier = tiers.First<Tier>(t => t.Name == "Medium Voltage");
```

Tier — General Info

`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`

Tier — Valid Asset Types

ValidDevices : IReadOnlyList<AssetType>

ValidLines : IReadOnlyList<AssetType>

- These properties return the asset types that can be included within the subnetwork

ValidSubnetworkControllers : IReadOnlyList<AssetType>

- The asset types that can serve as subnetwork controllers within this tier

ValidSubnetworkLines : IReadOnlyList<AssetType>

- The asset types that are used to build features in the SubnetLine feature class when updating the subnetwork

Tier — Subnetwork Properties

`GetDiagramTemplateNames() : IReadOnlyList<string>`

- Returns the names of the subnetwork diagram templates for this tier

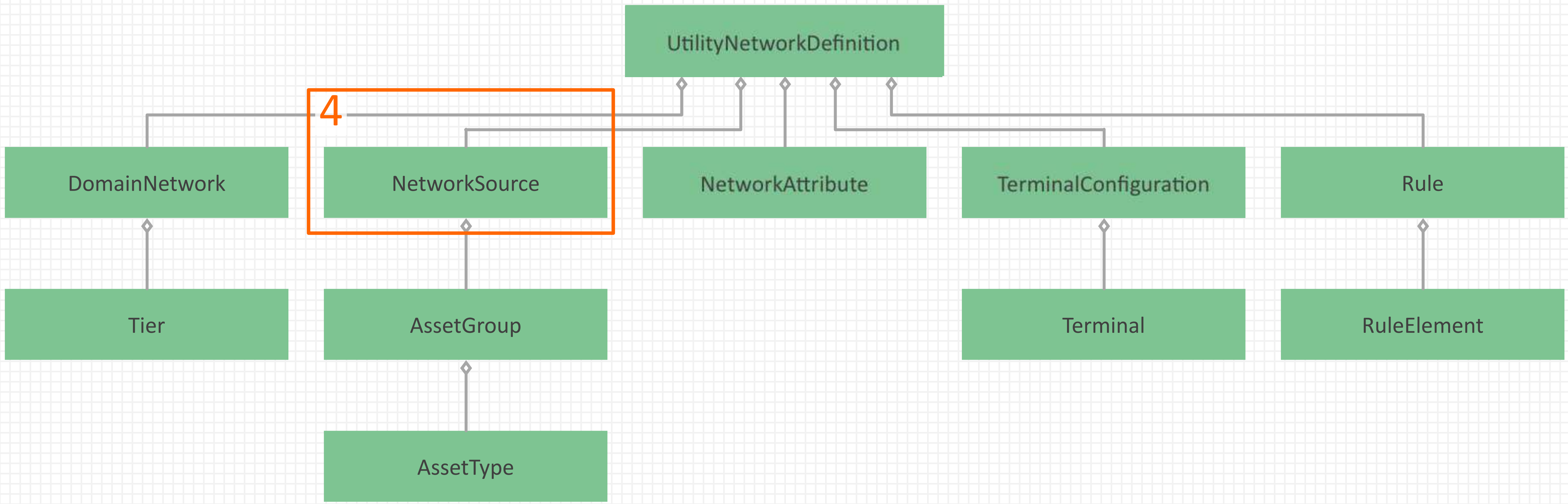
`IsDisjointSubnetworkSupported : bool`

- A disjoint subnetwork contains at least one subnetwork controller that is untraversable from its other subnetwork controllers
- If disjoint subnetworks are not supported, calling `Subnetwork.Update()` will result in an error

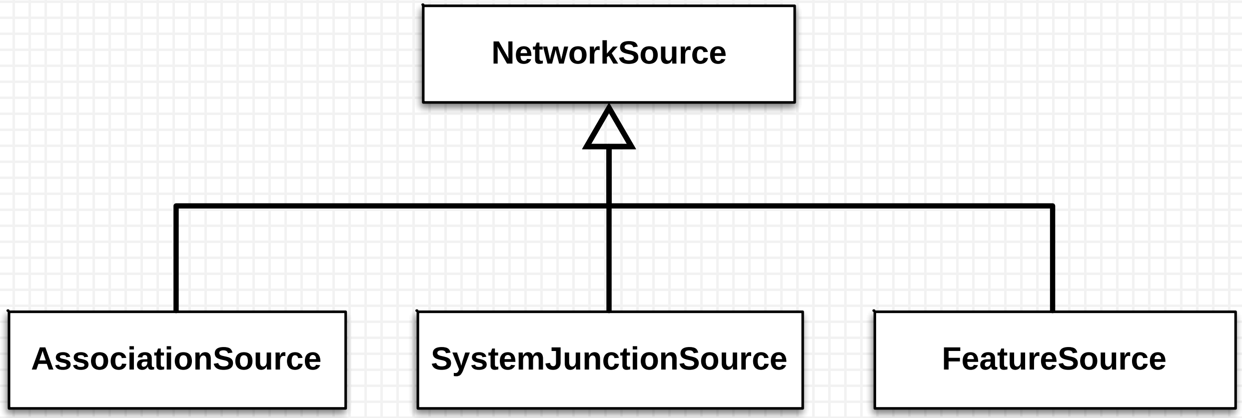
`TraceConfiguration : TraceConfiguration`

- The trace configuration that was set by the Set Subnetwork Definition geoprocessing tool
- The trace configuration is explained in detail in the Tracing section of this document

Network Sources

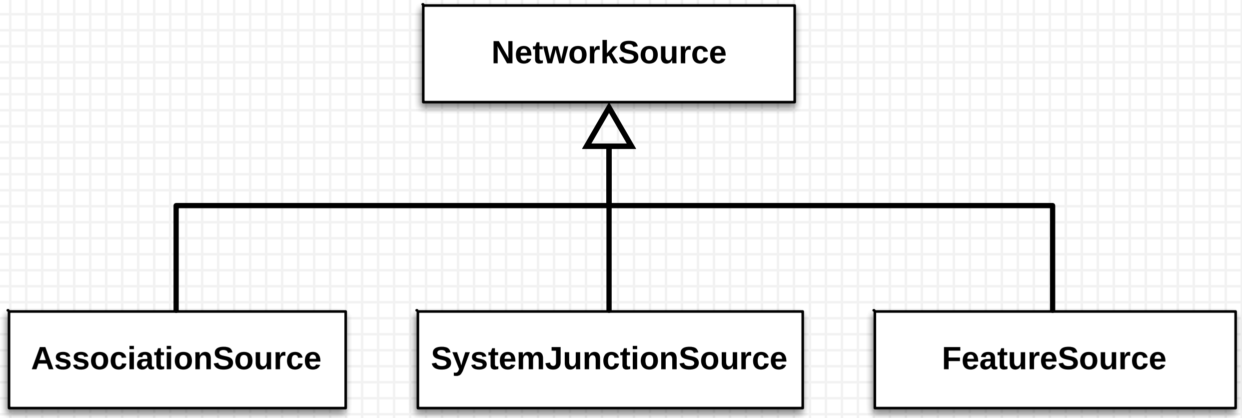


Network Sources



- One source of information in a utility network are the feature classes that make it up
  - Structure feature classes: StructureLine, StructurePoint, and StructureBoundary
  - Domain network feature classes: Device, Line, Junction, Assembly, and SubnetLine
- Another source of information is the set of associations that have been created
- The final source of information are the system junctions that are automatically generated where needed
- These sources of information are collectively known as network sources, and are represented by the `NetworkSource` class

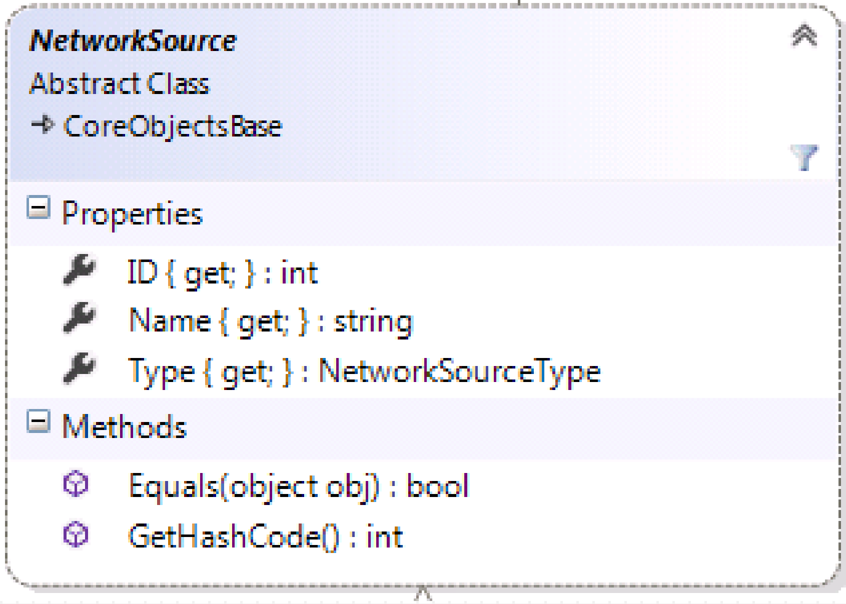
Network Sources



- 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 the 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`

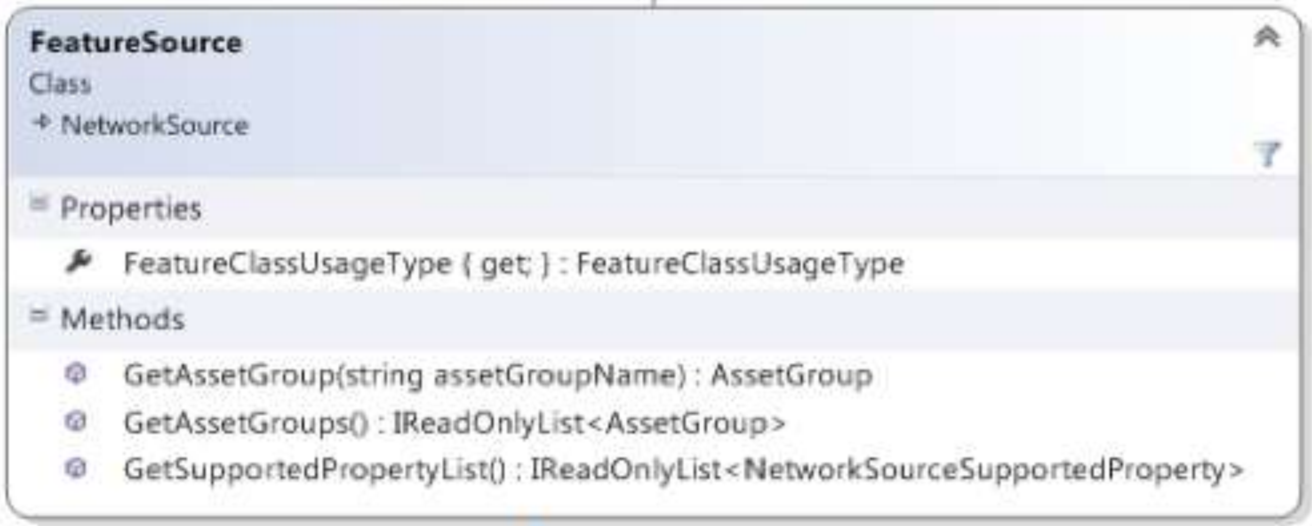
- The name of the network source. This is the name of the underlying table, not a user-readable name

`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 asset groups (subtypes) for this particular feature class

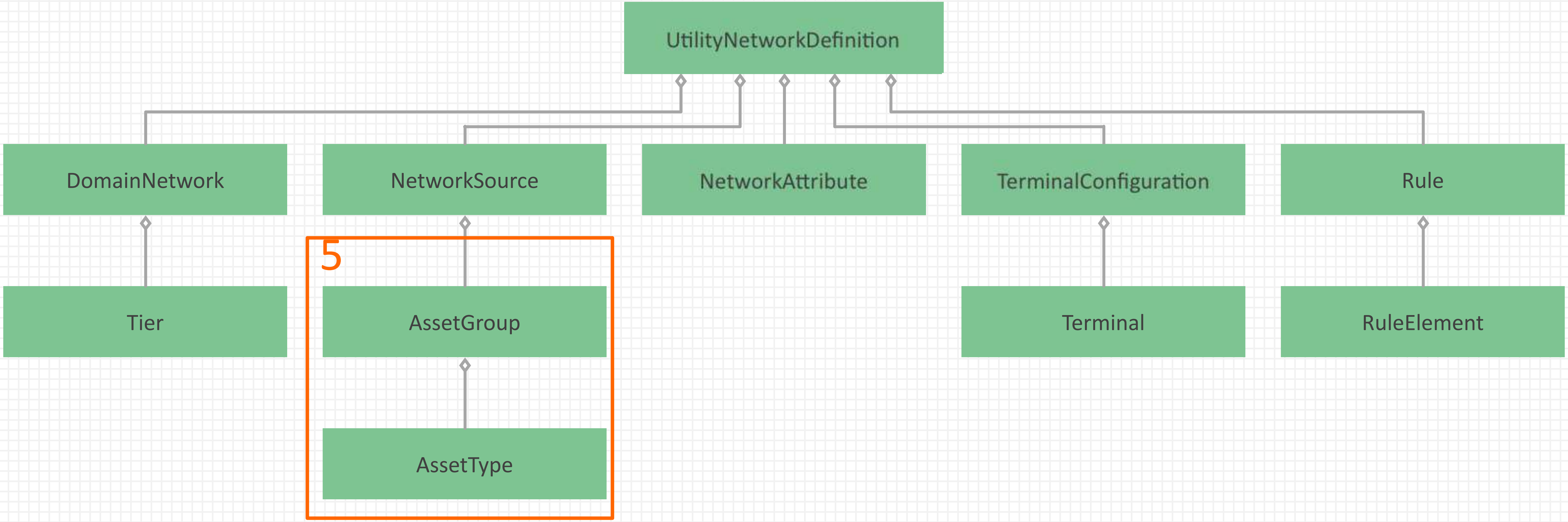
GetSupportedPropertyList() : IReadOnlyList<NetworkSourceSupportedProperty>

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

System Sources

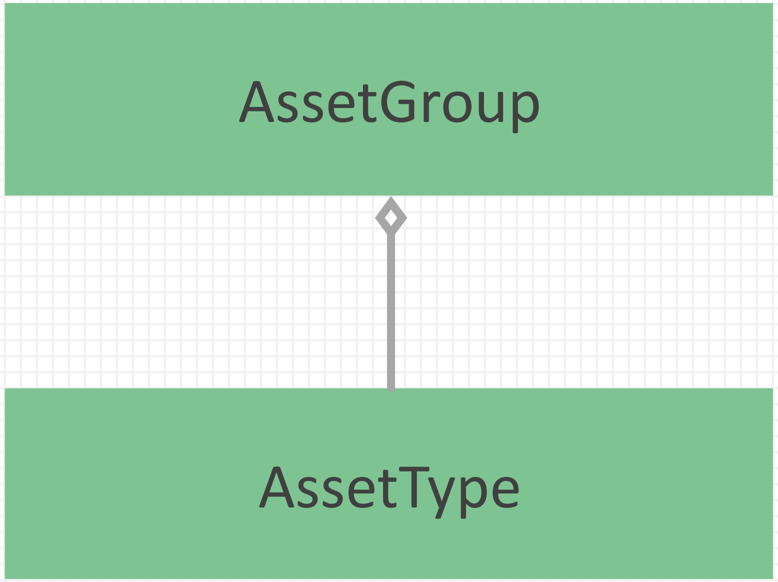
- `SystemJunctionSource`
  - Concrete class that represents system junctions (marker interface)
- `AssociationSource`
  - Concrete class that represents associations (marker interface)

Asset Groups and Asset Types



Asset Groups and Asset Types

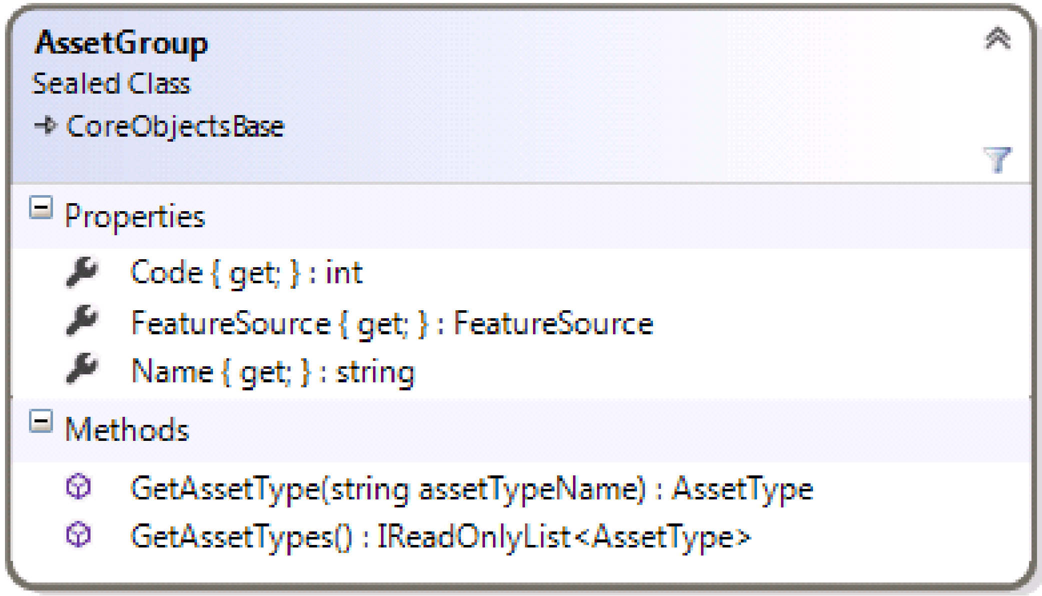
- There are two predefined fields on each feature class in a domain network and structure network that provide a classification system for all the features in a utility system
- These type attributes allow you to define feature types with great specificity while limiting the number of feature classes, important for high performance of the utility network
- Most configuration in the utility network is set at the asset type level



Asset Group

- The asset group attribute represents the primary classification of utility features
- The ASSETGROUP field is the subtype field of all feature classes in the structure network and domain networks (except for the SubnetLine class)
- Examples
  - Examples of asset group types for devices in an electric domain network could be Breaker, Capacitor, Fuse, Recloser, Switch, and Transformer
  - Examples of asset group types for lines in a gas domain network could be Connector, Distribution, GatheringWater, StationPipe, and Transmission
  - Examples of asset group types for assemblies in a water distribution domain network could be CompressorStation, PumpStation, RegulatorStation, and TownBorderStation

AssetGroup



- The `AssetGroup` class provides information about asset groups within the utility network. In the core geodatabase, they are implemented as subtypes

`Code : int`
- The subtype code

`FeatureSource : FeatureSource`
- The parent feature source

`Name : string`
- The subtype name

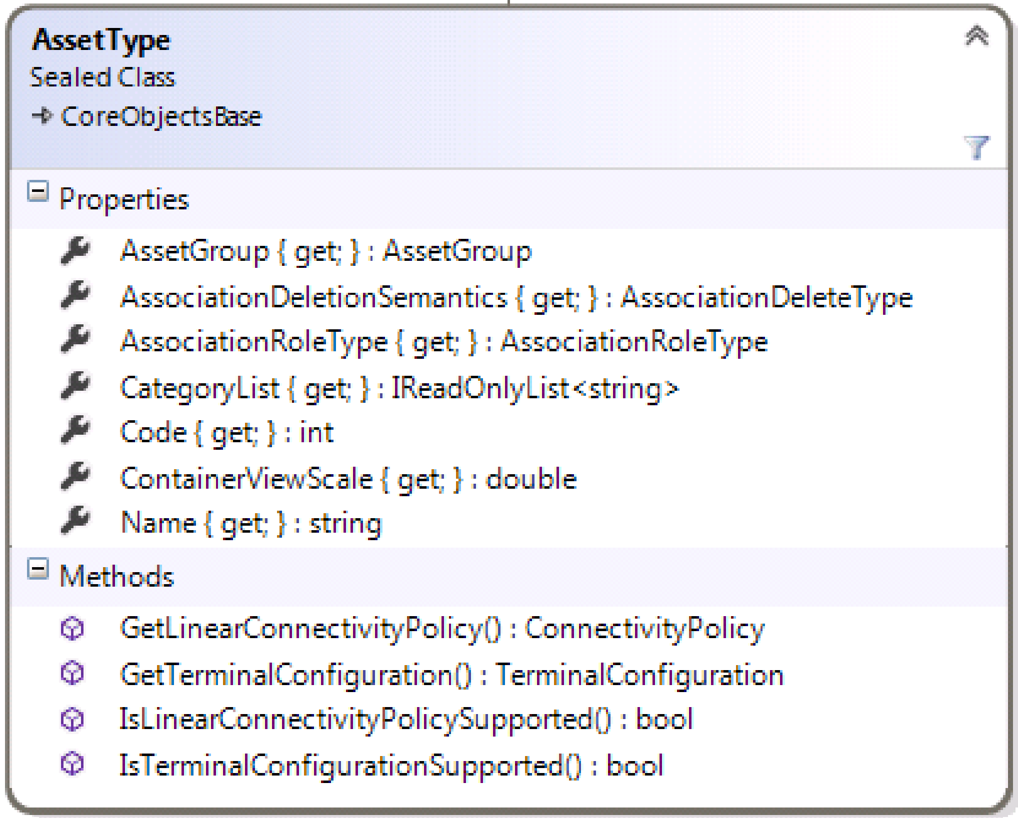
`GetAssetType(string assetTypeName) : AssetType`

`GetAssetTypes() : IReadOnlyList<AssetType>`
- Returns asset type definitions

AssetType

- The asset type attribute represents the secondary classification of asset group types
- The ASSETTYPE field is implemented as a set of attribute domains for each asset group
- Examples
    - Examples of asset type values for a transformer asset group in an electric distribution domain network could be StepTransformer, PowerTransformer, and DistributionTransformer
    - Examples of asset type values for a line feature class in a water distribution domain network could be PVCpipe, ClayPipe, and CastIronPipe

AssetType — 1



`AssetGroup : AssetGroup`

- The parent asset group

`Code : int`

- The code of the `AssetType` domain

`Name: string`

- The string value of the `AssetType` domain

`IsLinearConnectivityPolicySupported() : bool`

- Returns whether or not the `GetLinearConnectivityPolicy()` routine is applicable

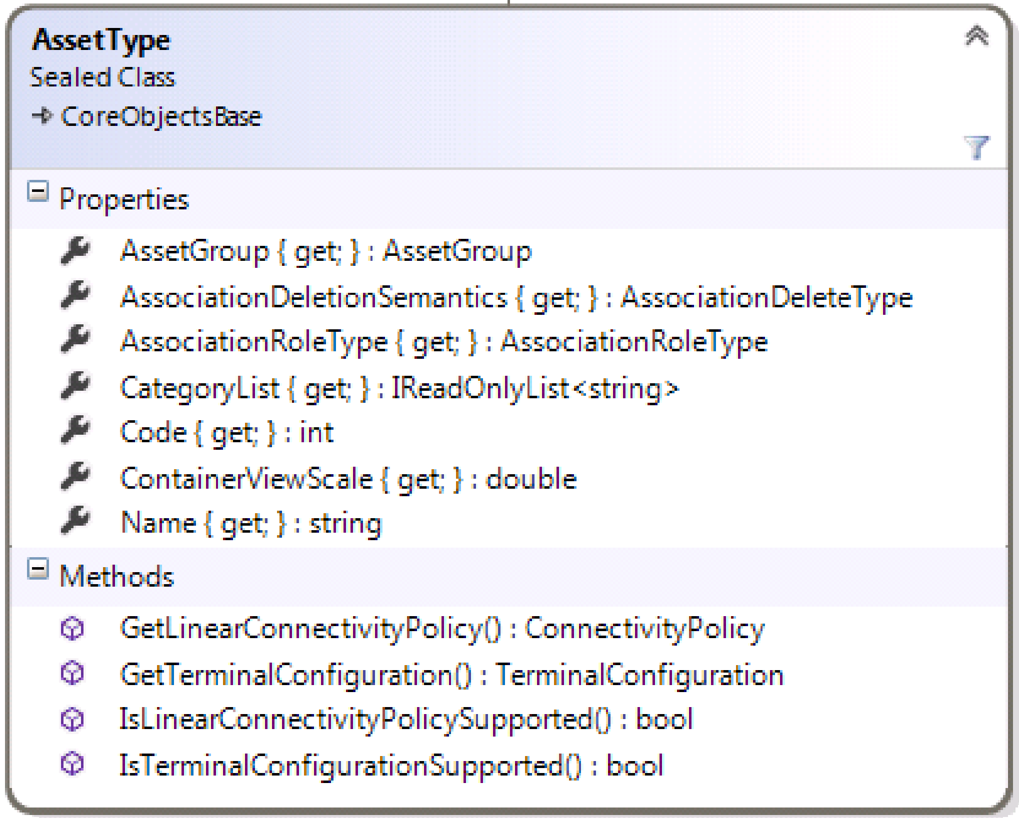
`GetLinearConnectivityPolicy() : ConnectivityPolicy`

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

`AssocationDeletionSemantics : AssociationDeleteType`

- The deletion type for this asset type (`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

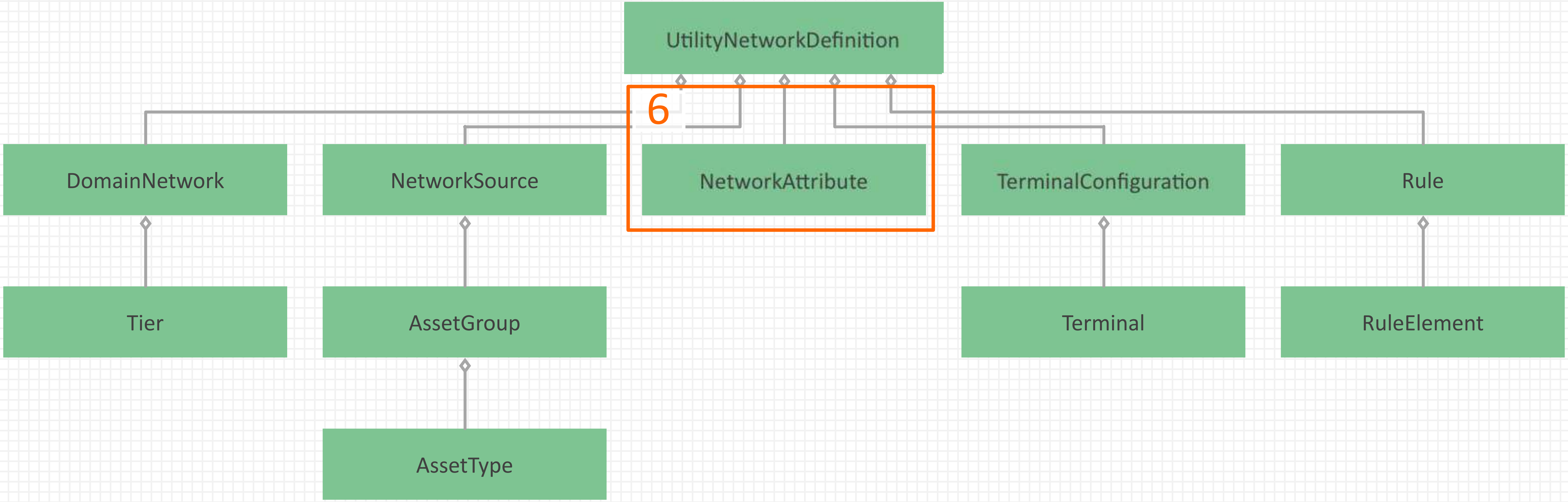
IsTerminalConfigurationSupported() : bool

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

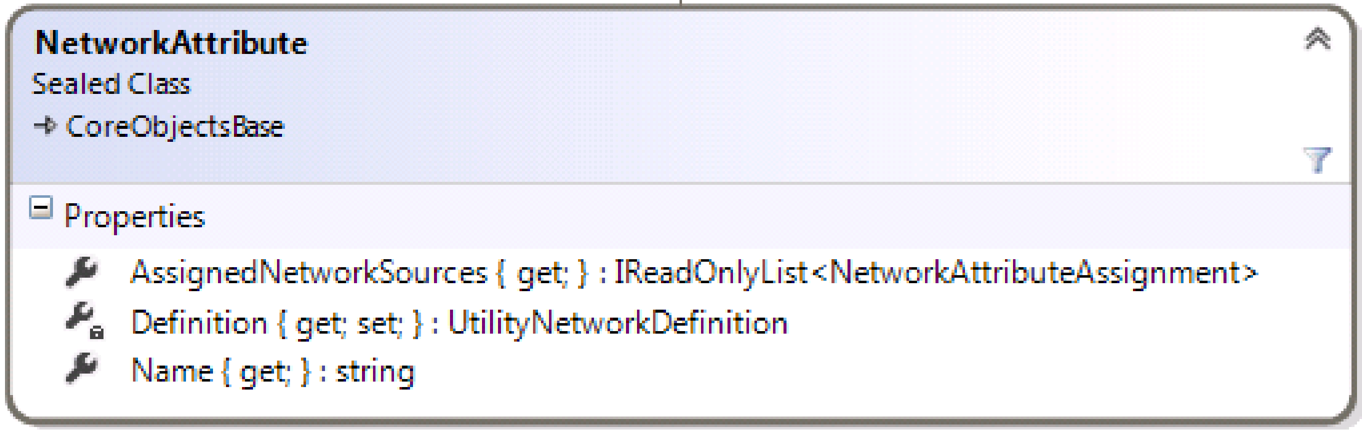
GetTerminalConfiguration() : TerminalConfiguration

- Returns the terminal configuration for this asset type

The NetworkAttribute Class



NetworkAttribute



- The NetworkAttribute class represents a network attribute (logically equivalent to a weight in the geometric network)
- Network attributes can be obtained by calling
  - UtilityNetworkDefinition.GetNetworkAttributes() : IReadOnlyList<NetworkAttribute>
  - UtilityNetworkDefinition.GetNetworkAttribute(string networkAttributeName): NetworkAttribute

AssignedNetworkSources : IReadOnlyList<NetworkSourceAssignment>

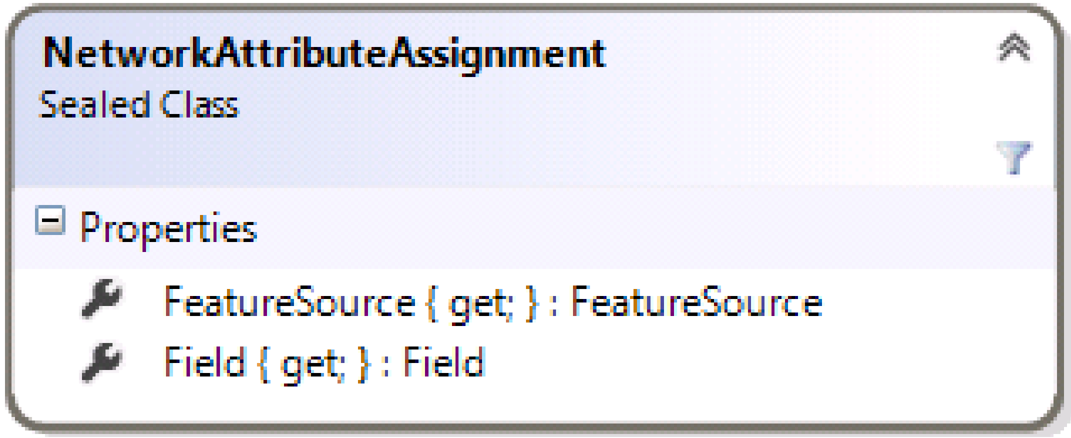
- A list describing how this network attribute is assigned to feature sources

- Name : string
  - The name of the network attribute

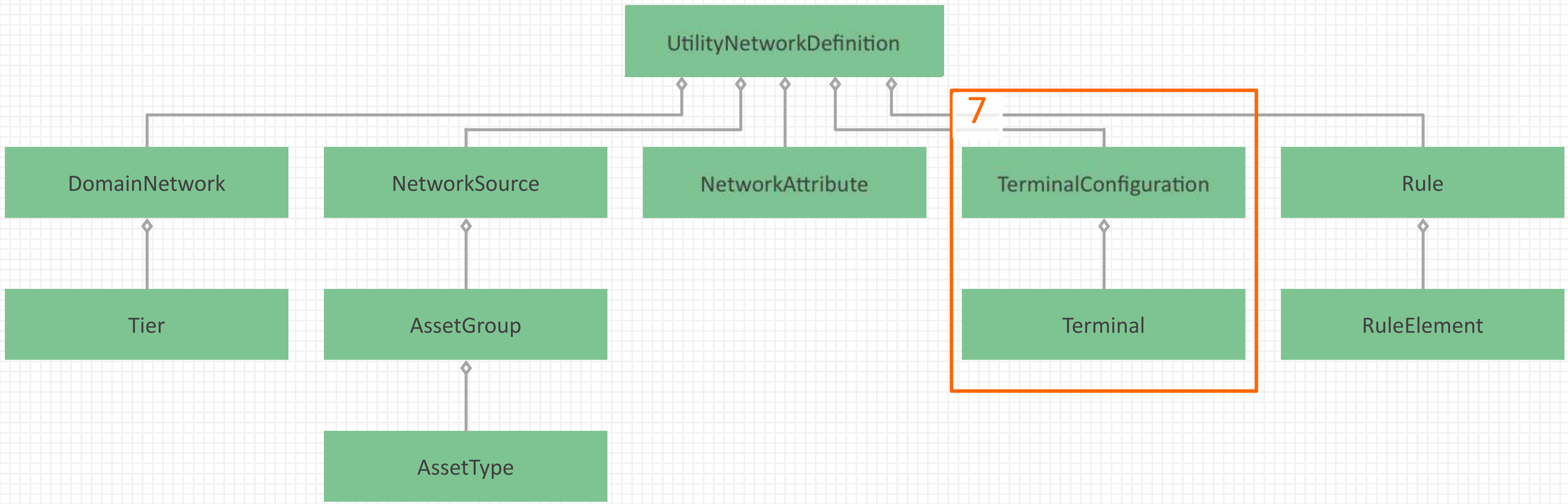
The NetworkAttributeAssignment Class

- 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

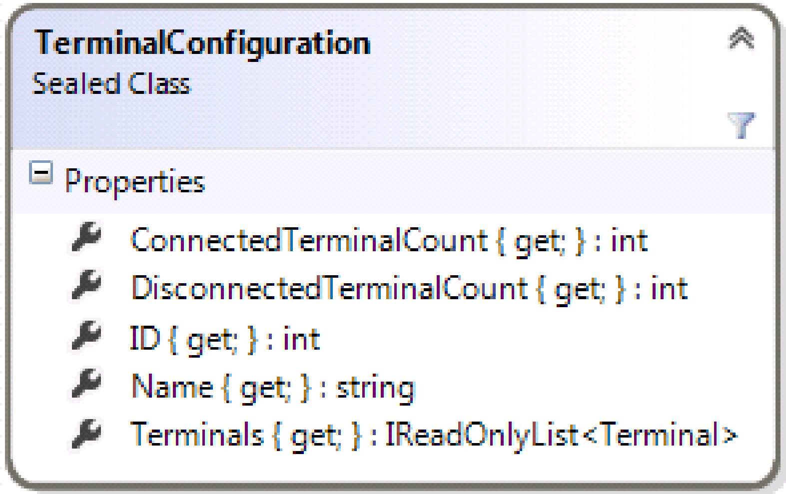
```
FeatureSource : FeatureSource

Field : Field
```



Terminals and Terminal Configurations



The TerminalConfiguration Class



- 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

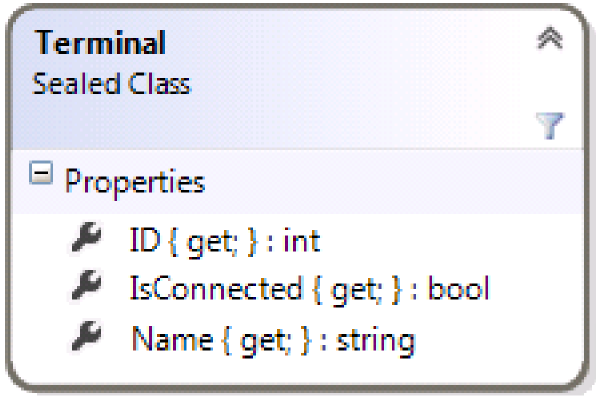
Name : string

- A user-readable string that describes the terminal configuration

Terminals : IReadOnlyList<Terminal>

This class will be redesigned in a future beta release

The Terminal Class



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

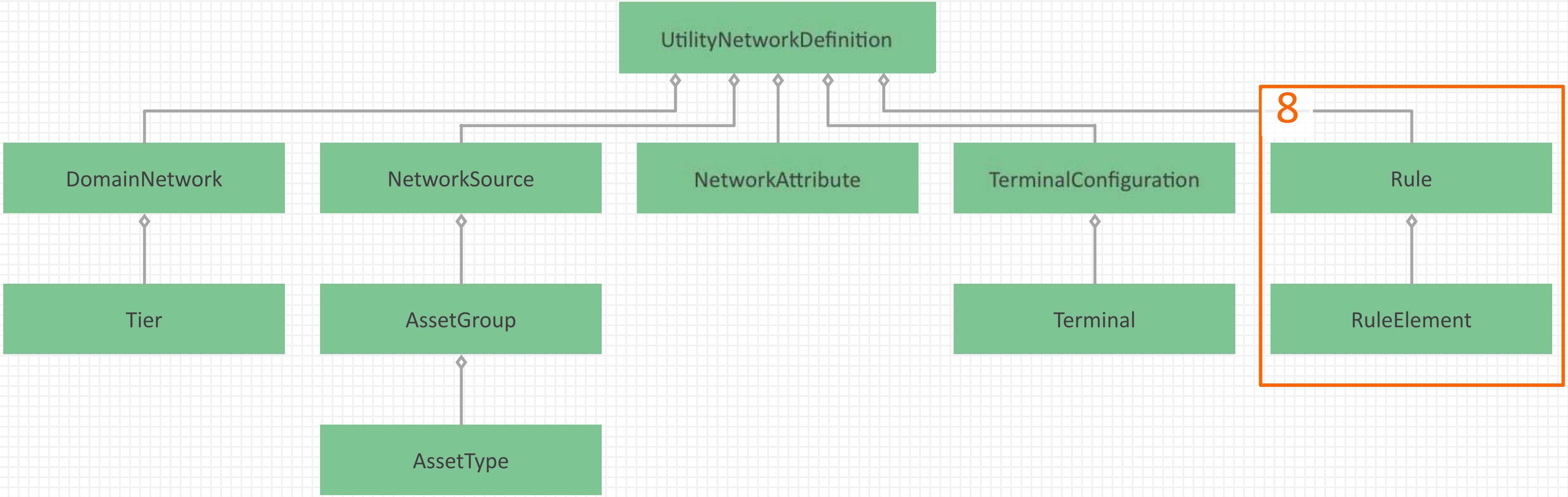
`ID : int`

`IsConnected : bool`

`Name : string`

- A user-readable string

Rules

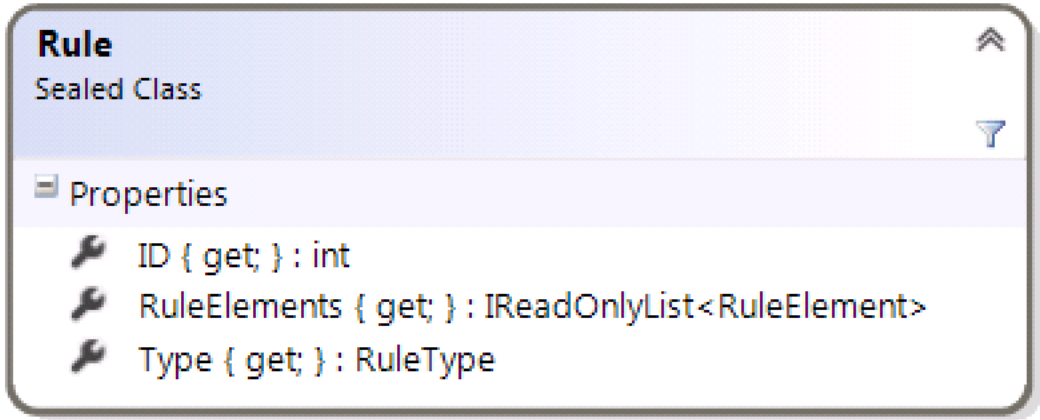


UtilityNetworkDefinition
DomainNetwork
NetworkSource
NetworkAttribute
TerminalConfiguration
Rule
Tier
AssetGroup
Terminal
RuleElement
AssetType
8

Rule

- Rules define which associations can be created between the different asset types in a utility network
- There are three broad types of association rules:
  - Connectivity rules enforce which types of features can be connected
  - Structural attachment rules enforce which types of features can be attached to a structure
  - Containment rules enforce which types of network features can be contained within container features
- Rule objects are obtained by calling
  - `UtilityNetworkDefinition.GetRules() : IReadOnlyList<Rule>`

The Rule Class



`Type : RuleType`

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

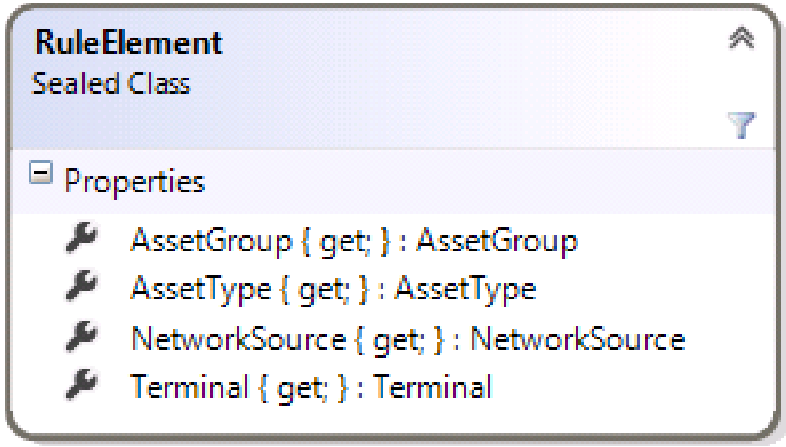
`ID : int`

- 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)

The RuleElement Class



- This class stores information about the elements of a rule

`NetworkSource : NetworkSource`

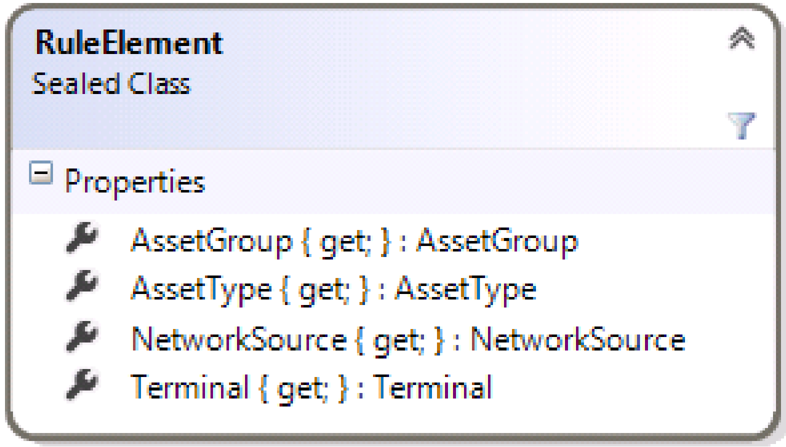
`AssetGroup : AssetGroup`

`AssetType : AssetType`

- If null, this means that all asset types of this asset group participate in the rule

`Terminal : Terminal`

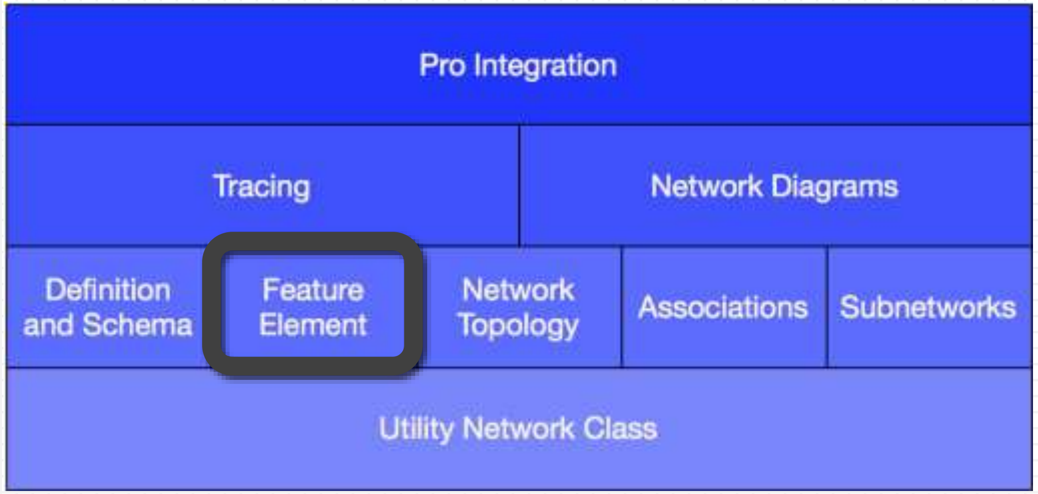
Meaning of RuleElement Properties



- 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.



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 API
- 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

- Feature elements are created using factory methods on the `UtilityNetwork` class
  - `CreateFeatureElement(AssetType assetType, Guid globalID, int terminalID) : FeatureElement`
  - `CreateFeatureElement(AssetType assetType, Guid gloalID) : FeatureElement`
  - `CreateFeatureElement(Row row, int terminalID) : FeatureElement`
  - `CreateFeatureElement(Row row) : FeatureElement`

FeatureElement Properties

`NetworkSource : NetworkSource`

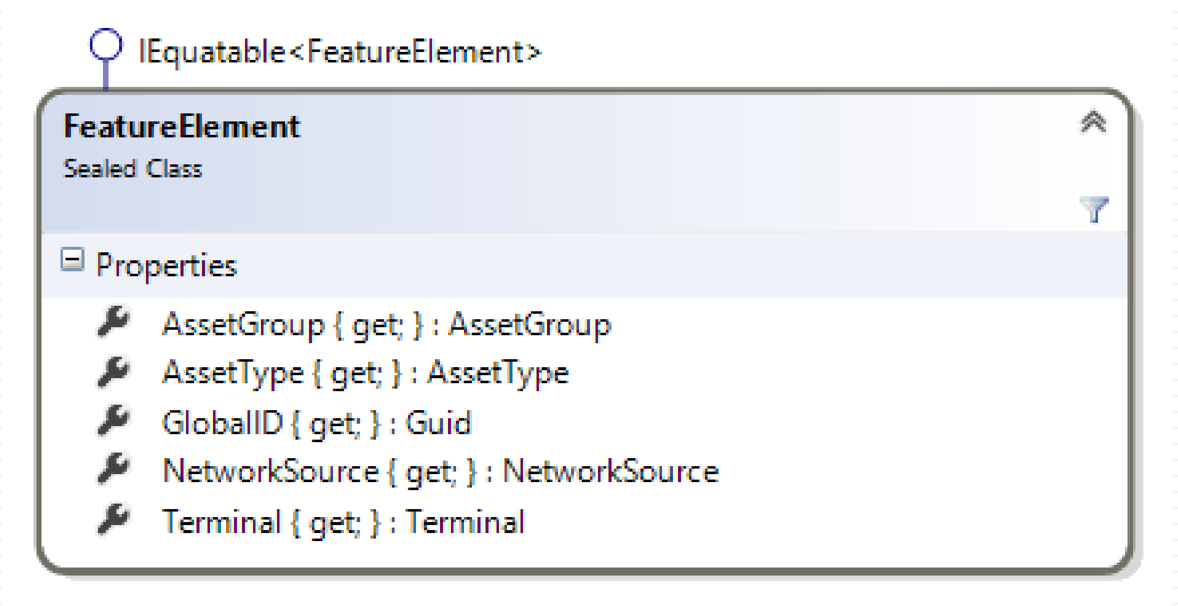
`AssetGroup : AssetGroup`

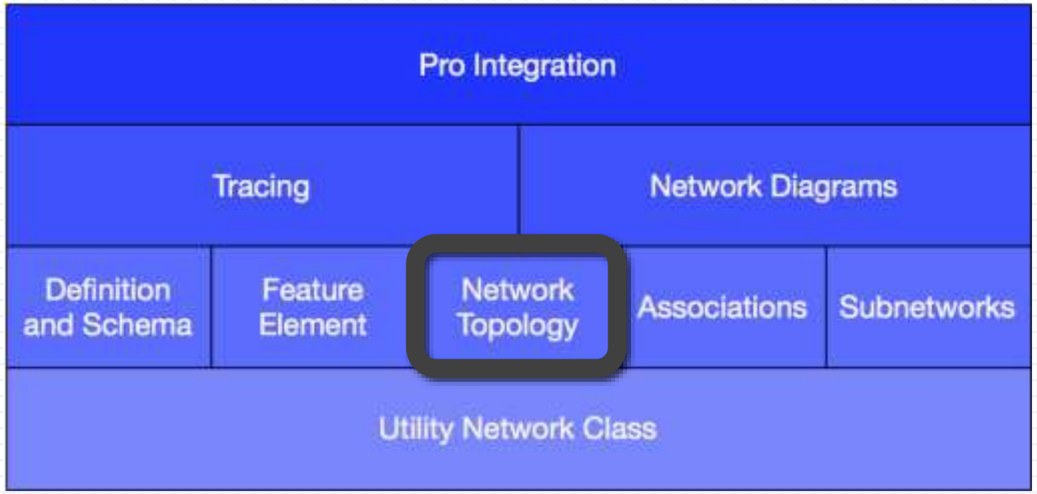
`AssetType : AssetType`

`GlobalID : Guid`

`Terminal : Terminal`

- `FeatureElement` objects can be compared for equality





Network Topology

Utility Network Topology

- The network topology stores connectivity, containment, and attachment information used by the utility network to facility 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 (e.g., Called from `Geodatabase.ApplyEdits()`)
- **Calls to ValidateNetworkTopology() must not be included inside a call to Geodatabase.ApplyEdits with other edits**
  - This is because ValidateNetworkTopology() *immediately* generates a REST call to the utility network service
  - Other calls inside ApplyEdits() are packaged up and executed in a single call to the feature service at the *end* of the method
  - The result is that validate occurs *before* the edits, regardless of it's placement within the ApplyEdits() delegate

The ValidationResult Class

- 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

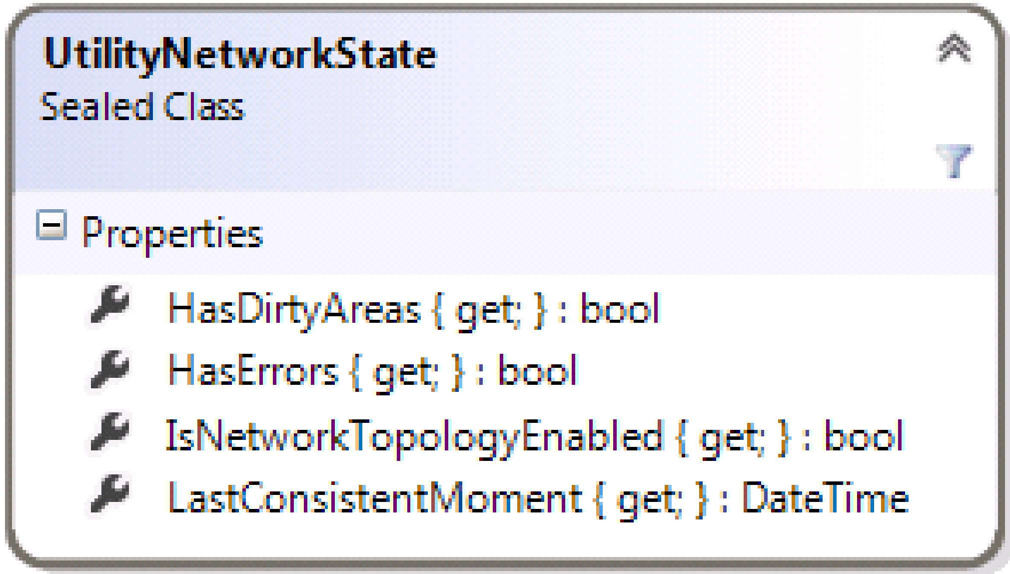
UtilityNetwork — Network Topology Queries

`GetExtent() : Envelope`

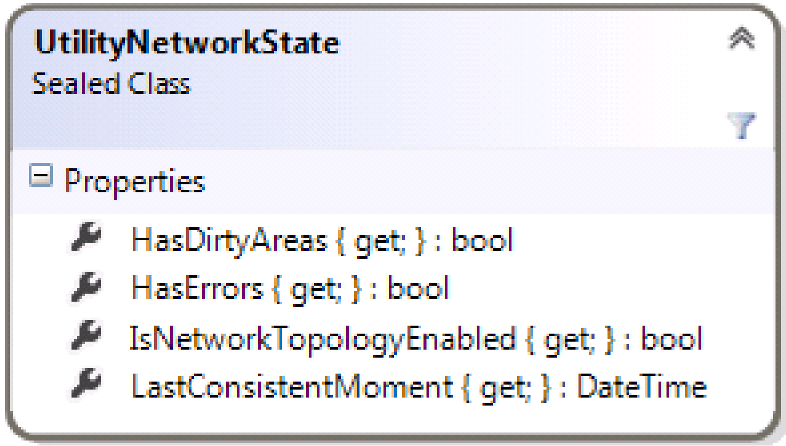
- Returns the extent of the feature classes within the network (same as other controller datasets in the product)

`GetState() : UtilityNetworkState`

- Returns the state of the network



The UtilityNetworkState Class



- 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`

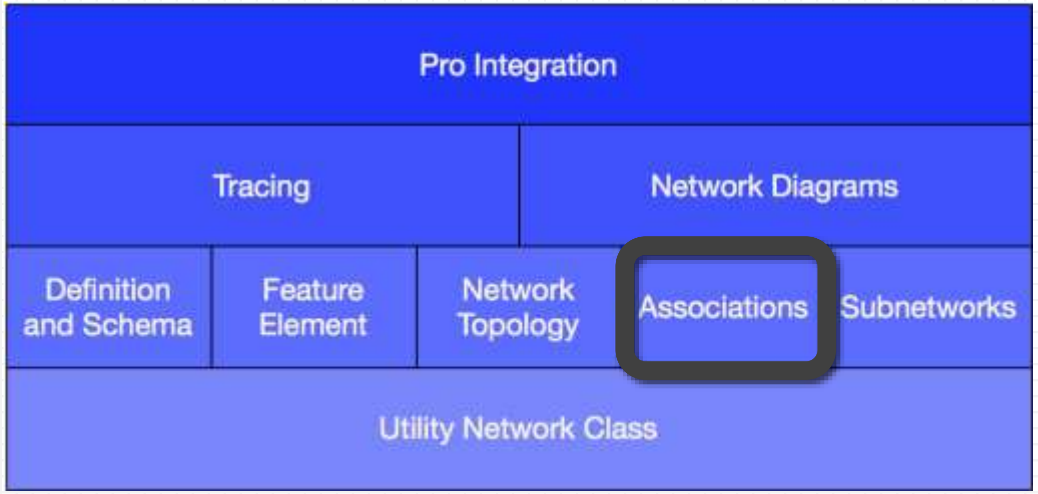
- Returns the `DateTime` of the last time that the utility network was fully validated

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 way



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, querying associations is the correct mechanism to use when building an editing tool, as they show the current edited state of the database

Connectivity Associations

- The `UtilityNetwork` class is used to create and delete connectivity 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

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

Containment Associations

- The `UtilityNetwork` class is used to create and delete containment associations
- These routines create dirty areas; they do not update network topology
- Calls to these routines must be wrapped in a transaction

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

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

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

Structural Attachment Associations

- The `UtilityNetwork` class is used to create and delete structural attachment associations
- These routines create dirty areas; they do not update network topology
- Calls to these routines must be wrapped in a transaction

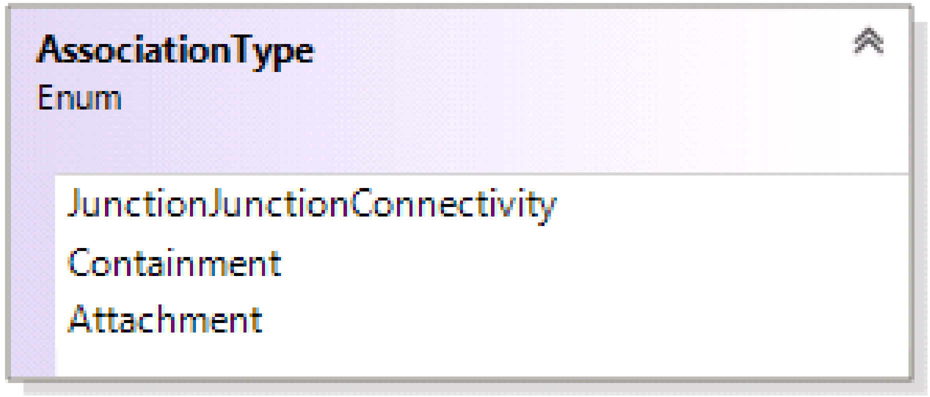
```
AddStructuralAttachmentAssociation(
      FeatureElement structureElement,
      FeatureElement attachmentElement) : void

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

Get Associations

- The `UtilityNetwork` class also provides routines to query associations

```
GetAssociations(FeatureElement featureElement,
     AssociationType type) : IReadOnlyList<Association>

GetAssociations(FeatureElement featureElement) :
     IReadOnlyList<Association>
```

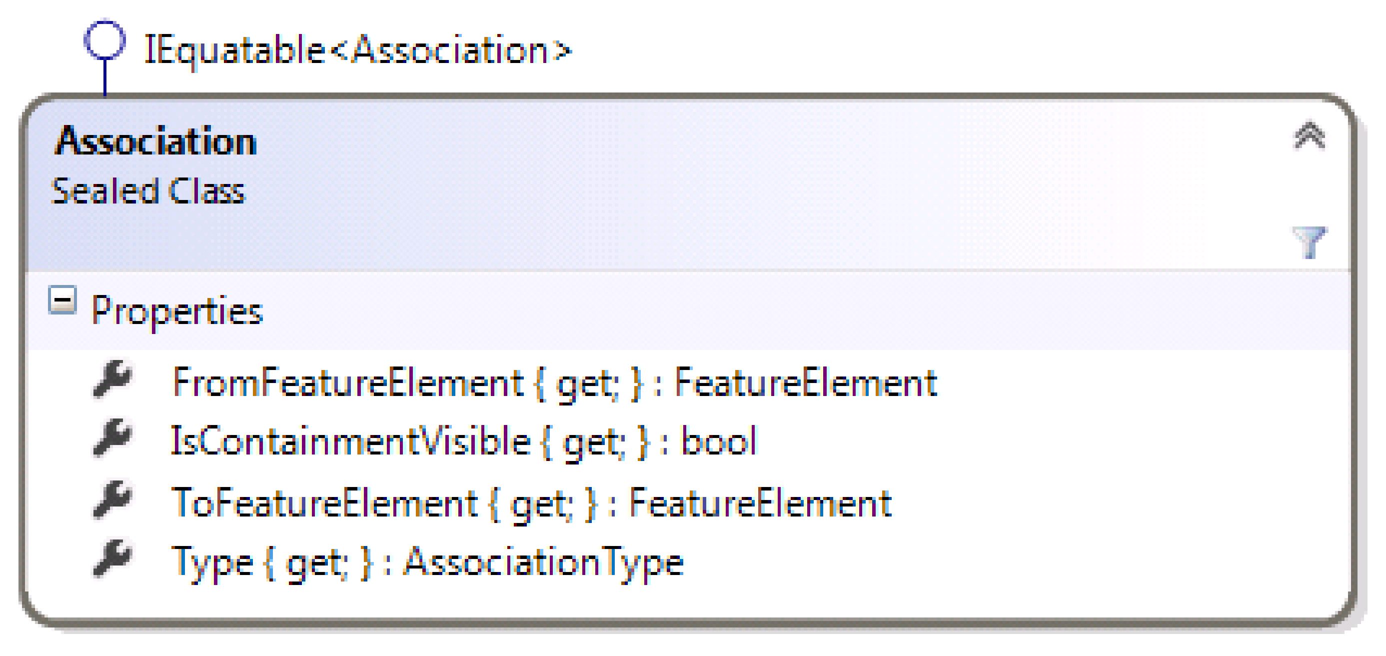
  - `AssociationType` is defined as follows:



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

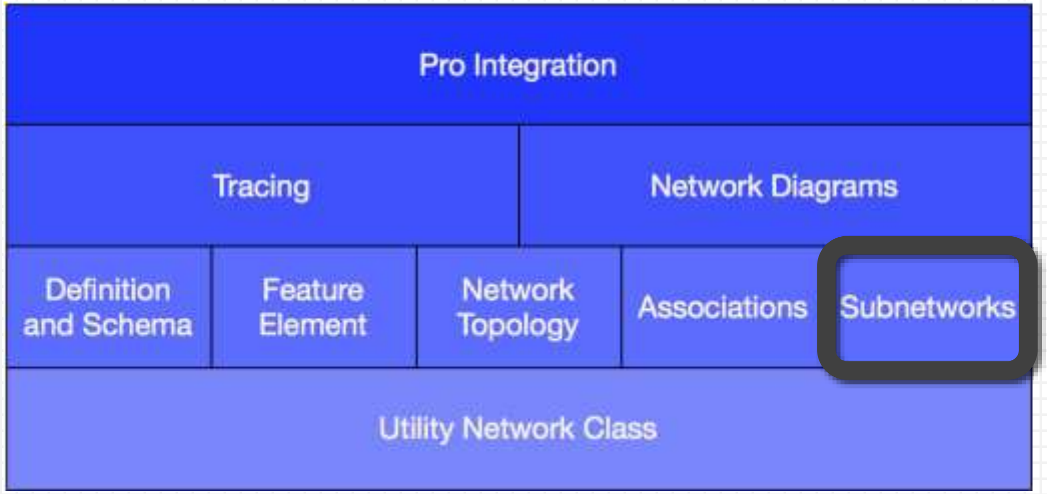
The Association Class

```
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



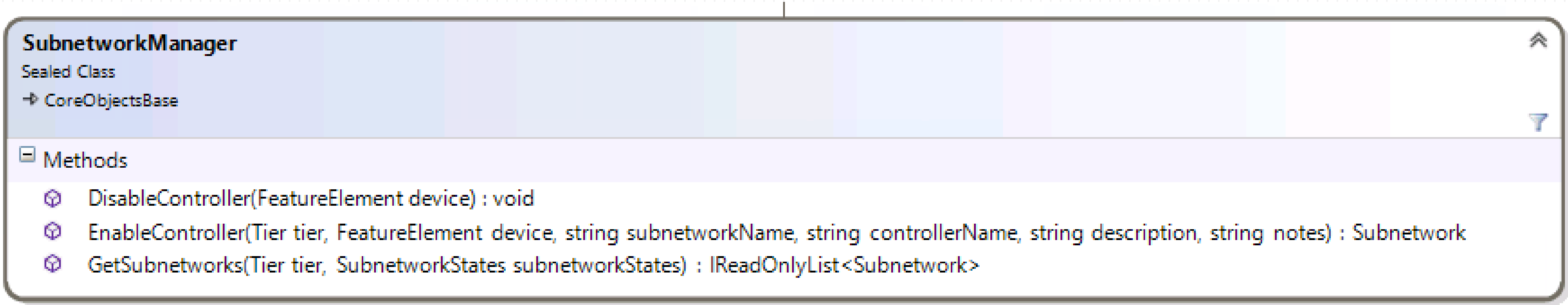
Subnetworks

Subnetworks

- Good network management depends on the reliability of paths in a network
- The management of these paths allows organizations to optimize the delivery of resources and track the status of a network
- In utility networks, paths are referred to as subnetworks
- A single subnetwork can be used to model such things as a circuit in electric networks, and a zone in gas and water networks
- More information about concepts relating to subnetworks can be found in the Subnetwork Management section of the online help

Subnetwork Manager

- The `SubnetworkManager` is a class that contains a collection of subnetwork management routines
- This provides a logical grouping of these routines
- The SubnetworkManager is obtained by calling a routine on the `UtilityNetwork`
  - `GetSubnetworkManager() : SubnetworkManager`



Adding Controllers

```
EnableController(Tier tier, FeatureElement device,
     string subnetworkName, string controllerName,
     string description, string notes) : Subnetwork
```

- This routine adds a subnetwork controller to an existing subnetwork, or creates a new subnetwork, depending on whether the `subnetworkName` already exists
- The `FeatureElement` must specify a terminal, and this terminal must be specified as a *Control Terminal* in the terminal definition
- The `controllerName` parameter is stored on the device feature in addition to being stored in the subnetworks table
- This routine creates a dirty area; it does not update the network topology
- Calls to this routine should **not** be wrapped in a transaction (e.g., `Geodatabase.ApplyEdits()`)

Removing Controllers

`DisableController(FeatureElement device) : void`

- This removes subnetwork controllers from the network
- If all subnetwork controllers are deleted from a subnetwork, the subnetwork moves to the `DirtyAndDeleted` state
- This routine creates a dirty area; the network topology is not updated
- Calls to this routine should **not** be wrapped in a transaction (e.g., `Geodatabase.ApplyEdits()`)

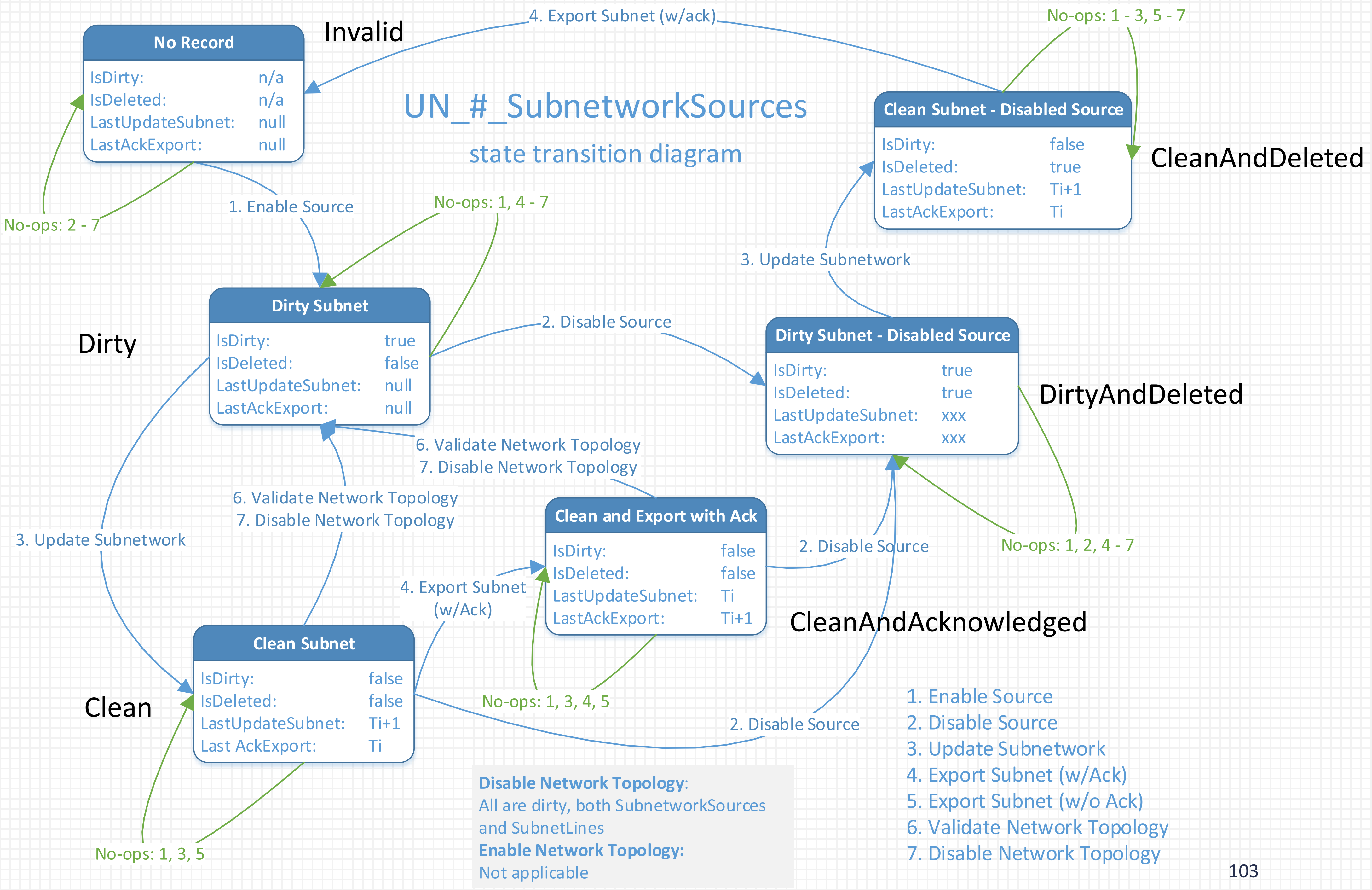
Getting Subnetworks

```
GetSubnetworks(Tier tier, SubnetworkStates subnetworkStates) :
     IReadOnlyList<Subnetwork>
```

- Returns a list of subnetworks of the specified type(s)

Values for SubnetworkStates

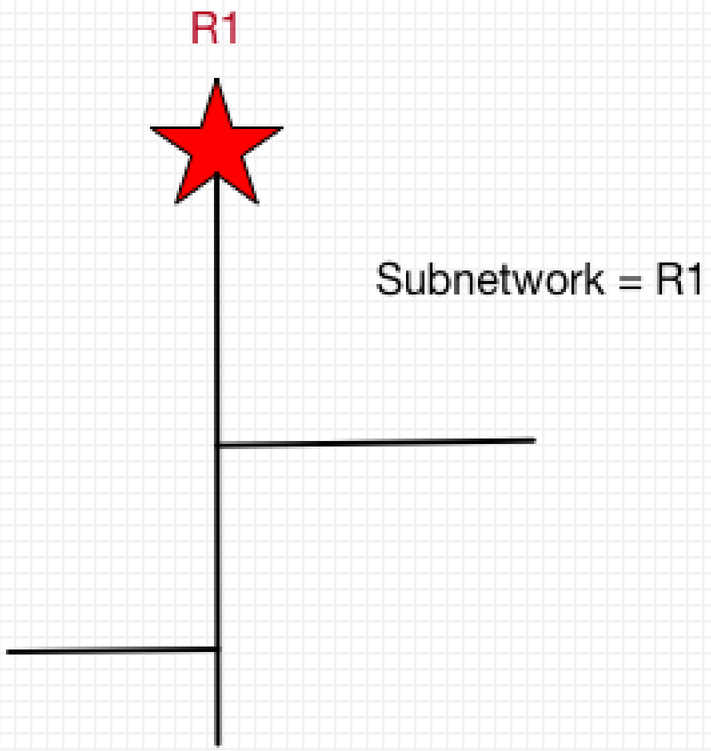
Value	Meaning
Dirty	Changes have been made to features within the subnetwork
Clean	Update() has been run subsequent to any edits made to the subnetwork
CleanAndAcknowledged	Export with the SetAcknowledged flag has been run after Update() and before additional edits
DirtyAndDeleted	All subnetwork controllers have been dropped from the subnetwork, but Update() has not yet been run
CleanAndDeleted	All subnetwork controllers have been dropped from the subnetwork, but Export() has not been run with the SetAcknowleded flag. Once a subnetwork is in this state, running Export() with the SetAcknowledged flag will delete the subnetwork
Invalid	The subnetwork has been completed deleted (no database rows exist), but the C# object remains. Once a subnetwork is in this state, most properties and methods will thrown an exception
All	This just ORs all the enum values together to provide an easy way to fetch all the subnetworks in a Tier


UN_#_SubnetworkSources
state transition diagram
No Record
IsDirty: n/a
IsDeleted: n/a
LastUpdateSubnet: null
LastAckExport: null
Invalid
No-ops: 2 - 7
1. Enable Source
No-ops: 1, 4 - 7
Dirty Subnet
IsDirty: true
IsDeleted: false
LastUpdateSubnet: null
LastAckExport: null
Dirty
4. Export Subnet (w/ack)
No-ops: 1 - 3, 5 - 7
Clean Subnet - Disabled Source
IsDirty: false
IsDeleted: true
LastUpdateSubnet: Ti+1
LastAckExport: Ti
CleanAndDeleted
3. Update Subnetwork
2. Disable Source
Dirty Subnet - Disabled Source
IsDirty: true
IsDeleted: true
LastUpdateSubnet: xxx
LastAckExport: xxx
DirtyAndDeleted
6. Validate Network Topology
7. Disable Network Topology
6. Validate Network Topology
7. Disable Network Topology
3. Update Subnetwork
Clean and Export with Ack
IsDirty: false
IsDeleted: false
LastUpdateSubnet: Ti
LastAckExport: Ti+1
2. Disable Source
No-ops: 1, 2, 4 - 7
CleanAndAcknowledged
4. Export Subnet
(w/Ack)
Clean Subnet
IsDirty: false
IsDeleted: false
LastUpdateSubnet: Ti+1
Last AckExport: Ti
Clean
No-ops: 1, 3, 4, 5
2. Disable Source
No-ops: 1, 3, 5
Disable Network Topology:
All are dirty, both SubnetworkSources and SubnetLines
Enable Network Topology:
Not applicable
1. Enable Source
2. Disable Source
3. Update Subnetwork
4. Export Subnet (w/Ack)
5. Export Subnet (w/o Ack)
6. Validate Network Topology
7. Disable Network Topology

Code Snippet: Creating a Radial Subnetwork

```
subnetworkManager.EnableController(mediumVoltageTier, featureElementR1, "R1", "R1", "my description", "my notes");
```

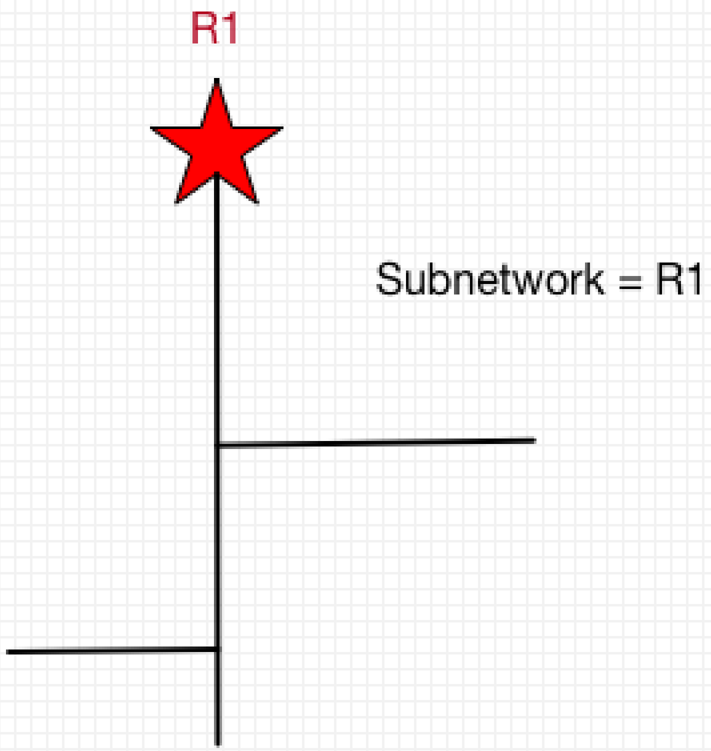
- Note that we have to give both the controller and the subnetwork itself a name
- Most clients will use the same name for both



Code Snippet: Deleting a Radial Network

```
subnetworkManager.DisableController(featureElementR1);
subnetworkR1.Update();
```

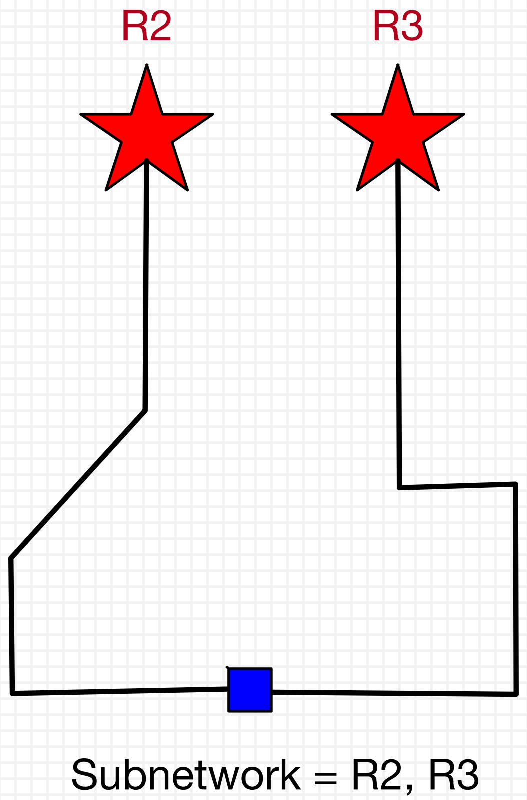
- `SubnetworkManager.DisableController()` will move the subnetwork to the `DirtyAndDeleted` state
- `Subnetwork.Update()` will move the subnetwork to the `CleanAndDeleted` state
- Running the Export Subnetwork geoprocessing tool with the export acknowledged field will delete the subnetwork



Code Snippet: Creating a Multi-feed Radial Subnetwork

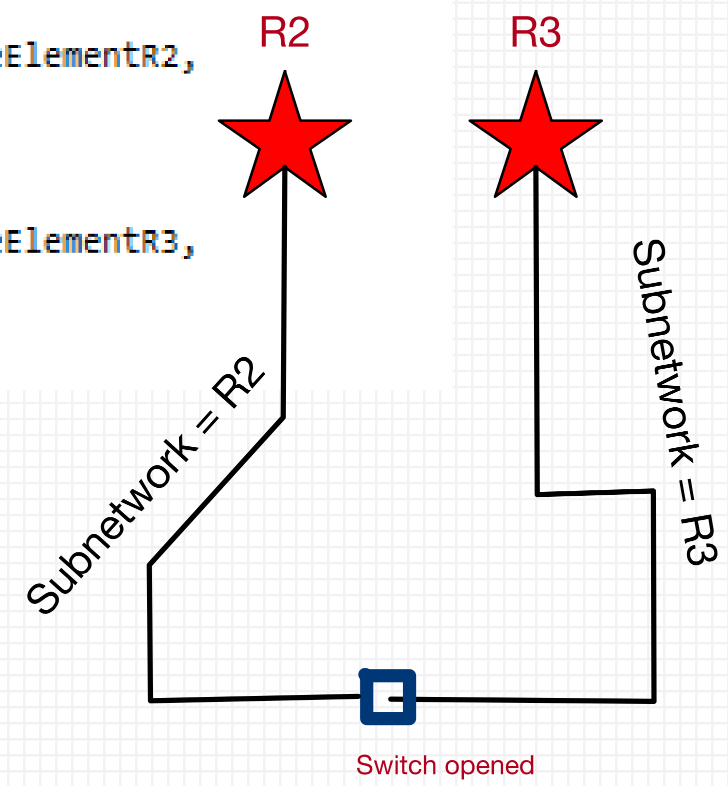
```
subnetworkManager.EnableController(mediumVoltageMultifeedTier, featureElementR2, "R2, R3", "R2", "my description", "my notes");
subnetworkManager.EnableController(mediumVoltageMultifeedTier, featureElementR3, "R2, R3", "R3", "my description", "my notes");
```

- Many utilities prefer to name their multi-feed subnetworks by concatenating the names of the control devices that feed it
- This must be done manually, as shown above by passing in `"R2, R3"` as an input argument



Code Snippet: Splitting a Multi-feed Radial Network

```
subnetworkManager.DisableController(featureElementR2);
subnetworkManager.DisableController(featureElementR3);
subnetworkR2R3.Update();
// Run the Export Subnetwork geoprocessing tool with the export acknowledged field to delete the subnetwork

Subnetwork subnetworkR2 = subnetworkManager.EnableController(mediumVoltageTier, featureElementR2,
  "R2", "R2", "my description", "mynotes");
subnetworkR2.Update();

Subnetwork subnetworkR3 = subnetworkManager.EnableController(mediumVoltageTier, featureElementR3,
  "R3", "R3", "my description", "mynotes");
subnetworkR3.Update();
```

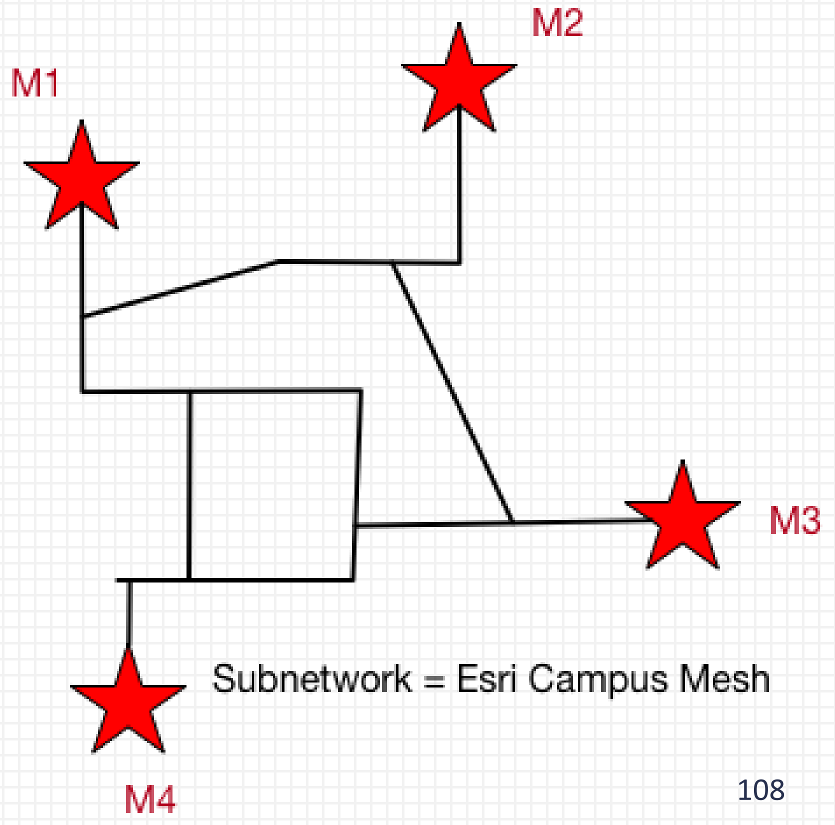


- In existing geometric network partner solutions, opening a tie devices in multi-feed radial networks will correctly regenerate subnetwork names based on the remaining sets of control devices on each side of the tie device
- In the utility network, partner solutions must continue to provide this functionality themselves

Code Snippet: Creating a Mesh

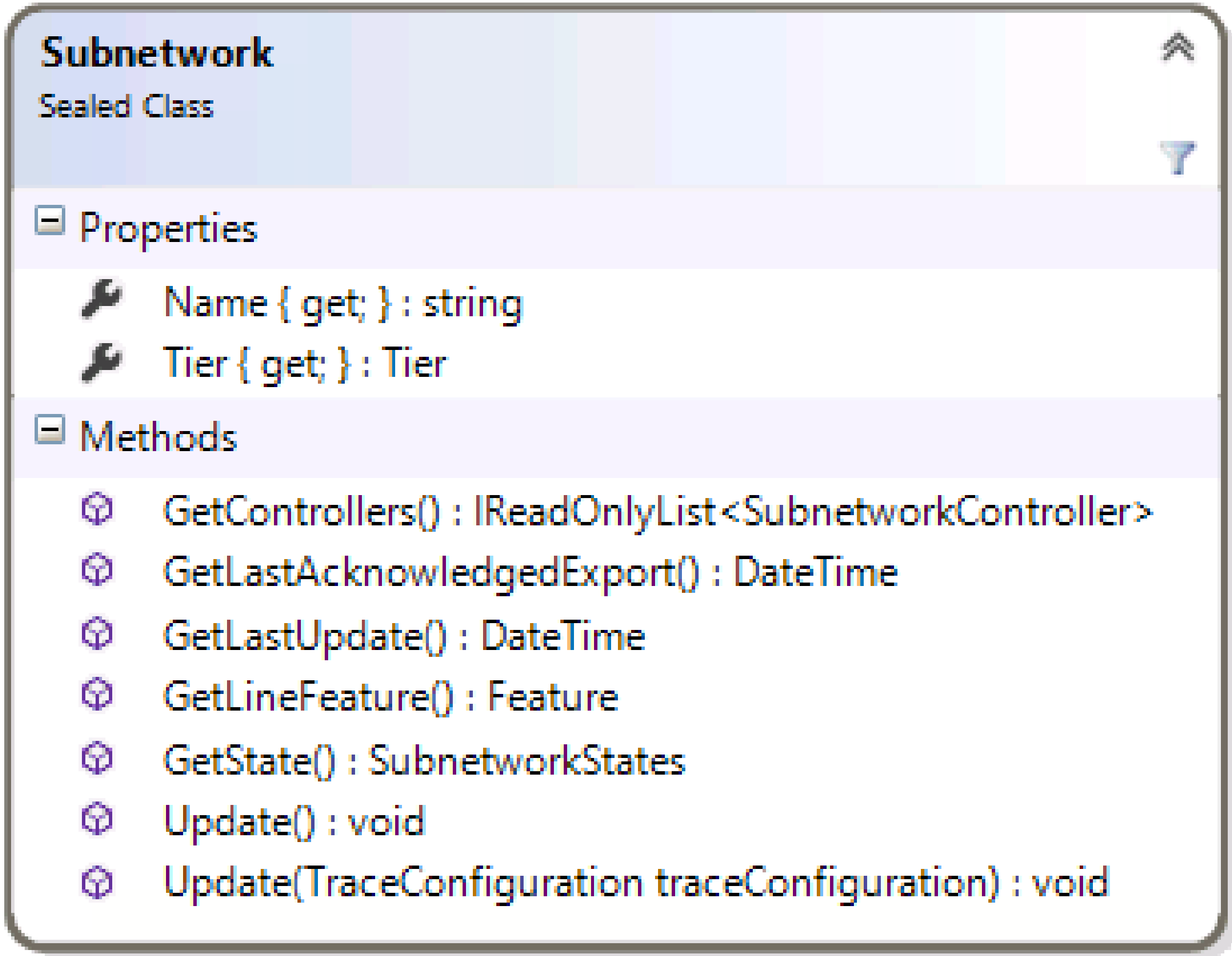
```
subnetworkManager.EnableController(mediumVoltageMeshTier, featureElementM1, "Esri Campus Mesh", "M1", "my description", "my notes");
subnetworkManager.EnableController(mediumVoltageMeshTier, featureElementM2, "Esri Campus Mesh", "M2", "my description", "my notes");
subnetworkManager.EnableController(mediumVoltageMeshTier, featureElementM3, "Esri Campus Mesh", "M3", "my description", "my notes");
subnetworkManager.EnableController(mediumVoltageMeshTier, featureElementM4, "Esri Campus Mesh", "M4", "my description", "my notes");
```

- In this example, we give each control device the same subnetwork name ("Esri Campus Mesh")



The Subnetwork Class

- Represents a subnetwork
- Provides access to subnetwork properties and operations



Subnetwork — Basics

`Name : String`

- The name of the subnetwork

`Tier : Tier`

- The Tier that contains this subnetwork

`GetControllers() : IReadOnlyList<SubnetworkController>`

- The list of one or more control devices that define this subnetwork

`GetLastUpdate() : DateTime`

- The last time that the subnetwork was updated

`GetLastAcknowledgedExport() : DateTime`

- The last time that the subnetwork was exported with the acknowledged flag

`GetState() : SubnetworkStates`

- The state of the subnetwork

Subnetwork — Updating

`Update() : void`

`Update(TraceConfiguration traceConfiguration) : void`

- Updates the subnetwork. The default `TraceConfiguration` stored on the tier is used, unless an override is specified
- If the preexisting subnetwork state is `Dirty`, or `Clean`, the new state will be `Clean` if successful
- If the preexisting subnetwork state is `DirtyAndDeleted`, or `CleanAndDeleted`, the new state will be `CleanAndDeleted` if successful
- All edits in the current edit session must be saved before calling this routine
- This routine should **not** be called within a transaction (e.g., `Geodatabase.ApplyEdits()`)

Subnetwork — Miscellaneous

`GetLineFeature() : Feature`

- Returns the row from the SubnetLine table that corresponds to this subnetwork
- Note that if `UpdateSubnetwork()` has not yet been called, or if the subnetwork does not contain any of the features whose subtypes are included in `Tier.ValidSubnetworkLines`, the feature may not exist

Code Snippets: Updating a Subnetwork

- Using the default subnetwork definition for the Tier
  - Network Attribute filter: {Lifecycle Status = In Service }
  - Termination Filter: { Device Status = Open}

```
subnetwork.Update();
```

- Using the same network, but include features with Lifecycle Status = In Design

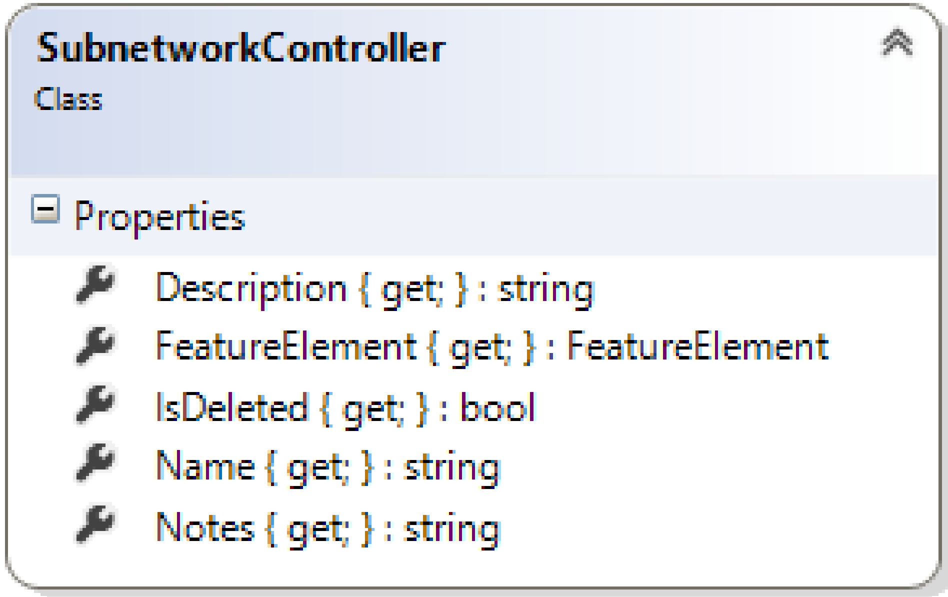
```
TraceConfiguration traceConfiguration = mediumVoltageTier.TraceConfiguration;
traceConfiguration.TraversalFilter = new Or(
  new NetworkAttributeFilter(lifecycleNetworkAttribute, FilterOperator.Equal, InService),
  new NetworkAttributeFilter(lifecycleNetworkAttribute, FilterOperator.Equal, InDesign));

subnetwork.Update(traceConfiguration);
```

Code Snippet: Updating All Dirty Subnetworks

```
IReadOnlyList<Subnetwork> subnetworks =
  subnetworkManager.GetSubnetworks(tier, SubnetworkStates.Dirty | SubnetworkStates.DirtyAndDeleted);
foreach (Subnetwork subnetwork in subnetworks)
{
  subnetwork.Update();
}
```

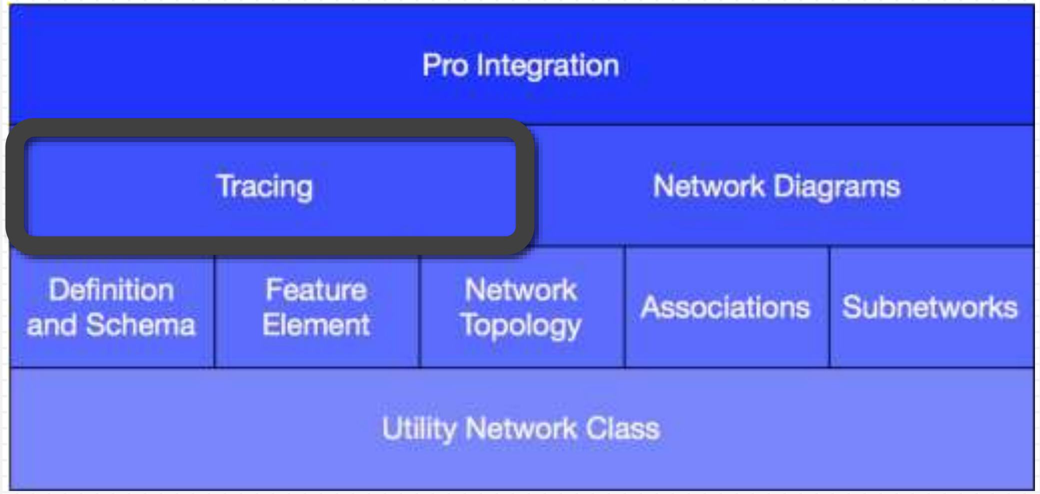
- It's important to note that the MCT will block during each of the calls to `Subnetwork.Update()`
- If updating subnetworks is expected to take a great deal of time, a stand-alone app rather than an ArcGIS Pro add-in might be more appropriate
- Calling the REST routine directly on multiple threads would produce the most throughput, as potentially each call could be executed on a different server process

The SubnetworkController Class



Subnetwork State and SubnetworkController.IsDeleted

- Remember that each subnetwork can be defined by multiple subnetwork controllers
- Calling `SubnetworkManager.DisableController()` will do the following:
  - Set `SubnetworkController.IsDeleted` to `True`
  - If **all** of the controllers of a subnetwork are marked as deleted, the subnetwork state is set to `DirtyAndDeleted`
  - If only some of the controllers of a subnetwork are marked as deleted, the subnetwork state is set to `Dirty`
- Calling `Subnetwork.Update()` will set the subnetwork state to `Clean` or `CleanAndDeleted` as appropriate
- Calling the Export Subnetwork geoprocessing tool with the export acknowledged field will set the subnetwork state to `CleanAndAcknowledged` as appropriate


Pro Integration
Tracing
Network Diagrams
Definition and Schema
Feature Element
Network Topology
Associations
Subnetworks
Utility Network Class

Tracing

What is Tracing?

- Tracing entails assembling a subset of utility network elements that meet a specified criteria
- Tracing uses network data to provide business value to utilities
  - Answers questions and solves problems about the current state of the network
    - What valves need to be opened to shut off gas to this location?
    - If these three houses lost power during a storm, what device is the culprit?
  - Helps design future facilities
    - How many houses are fed by this transformer, and can the equipment handle another connection?
  - Helps organize business practices
    - How can I create a circuit map to give to my work crews for damage assessment after an ice storm?

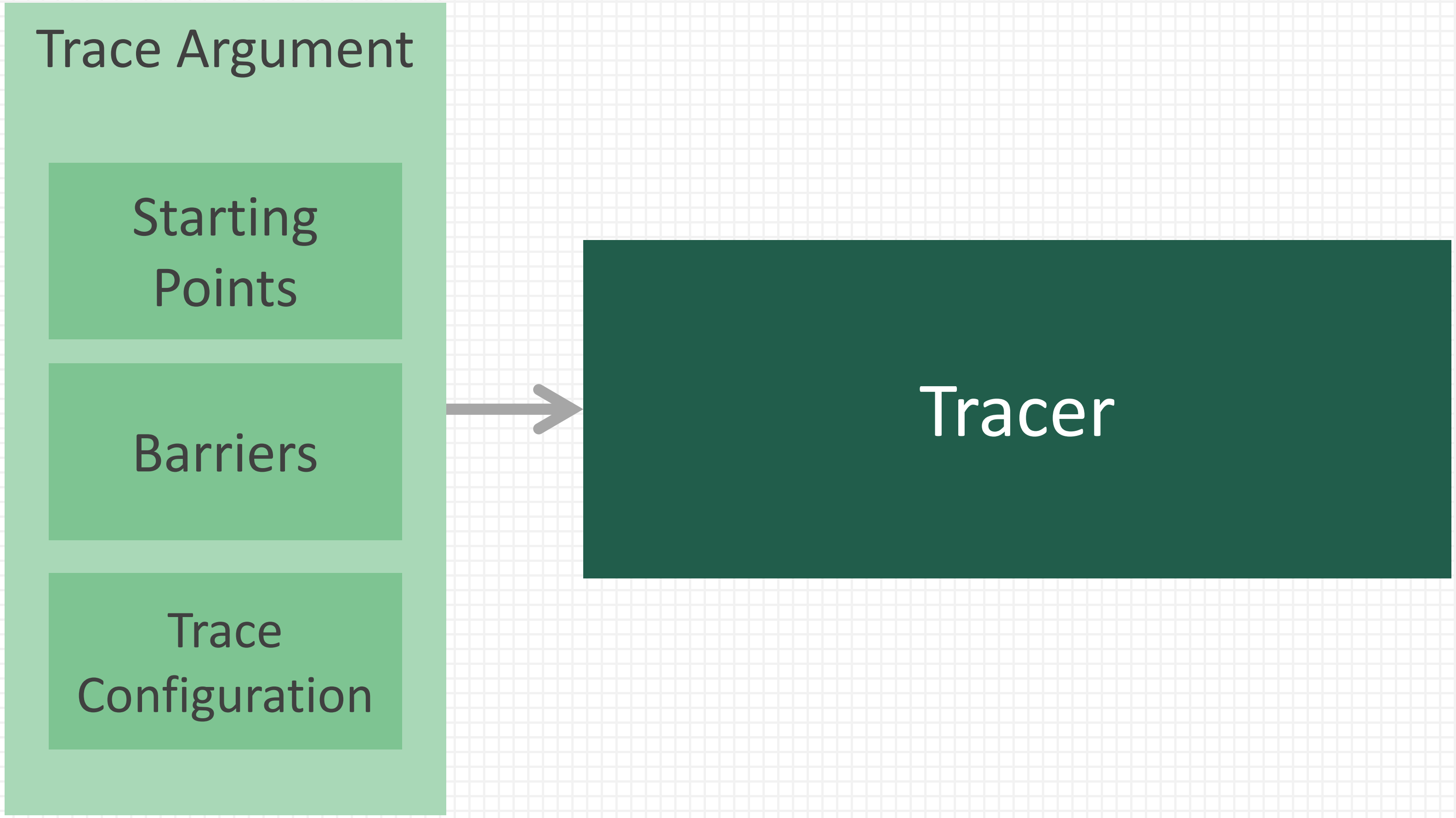
Anatomy of a Trace

- Different kinds of traces are implemented with Tracer objects

Tracer

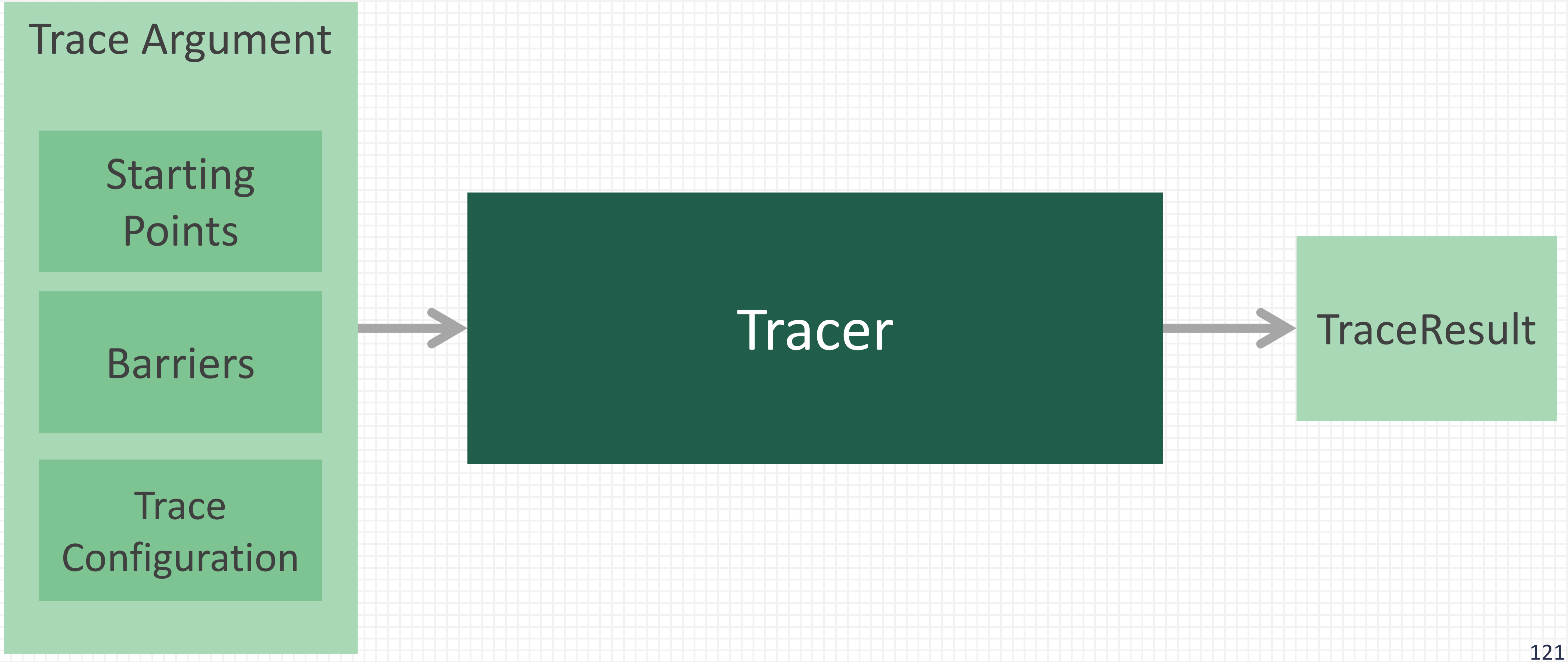
Anatomy of a Trace

- Starting Points, Barriers, and a Trace Configuration are inputs to the Tracer object



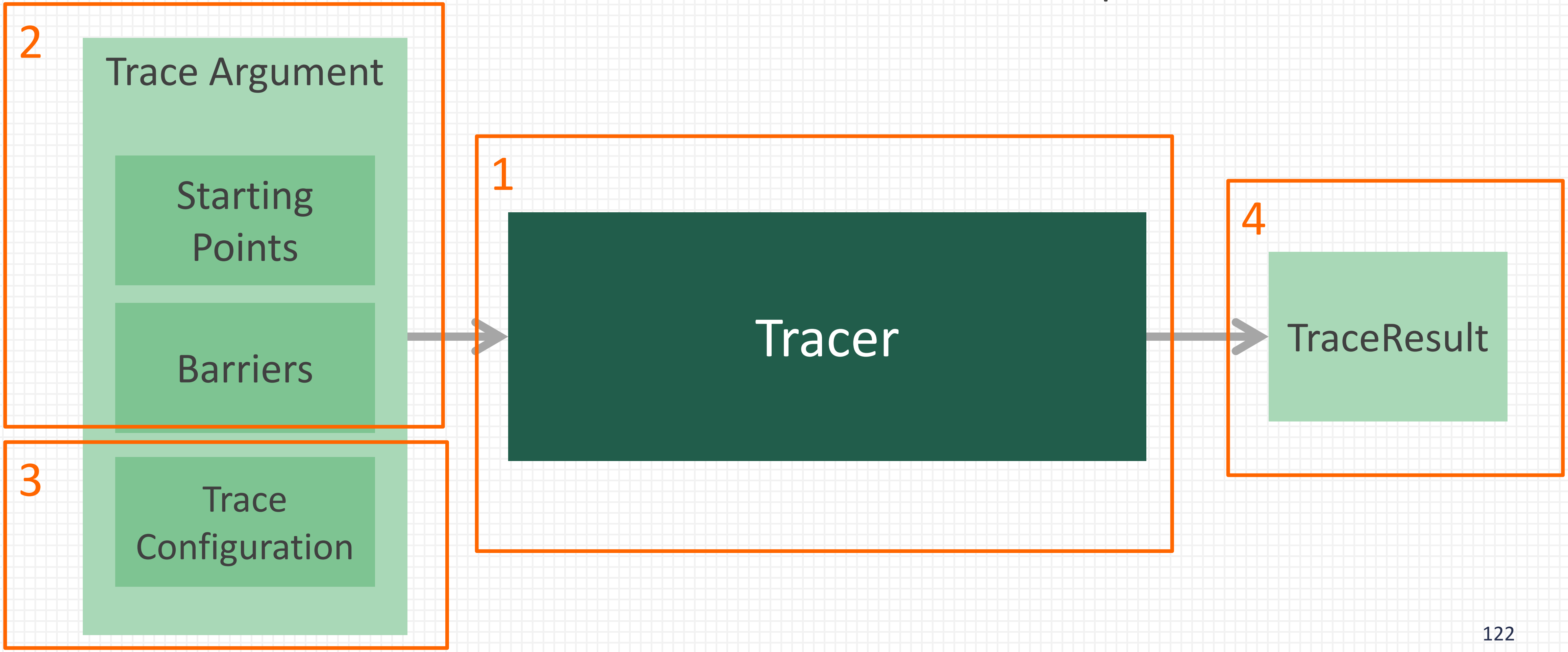
Anatomy of a Trace

- The Tracer generates a Trace Result object as output



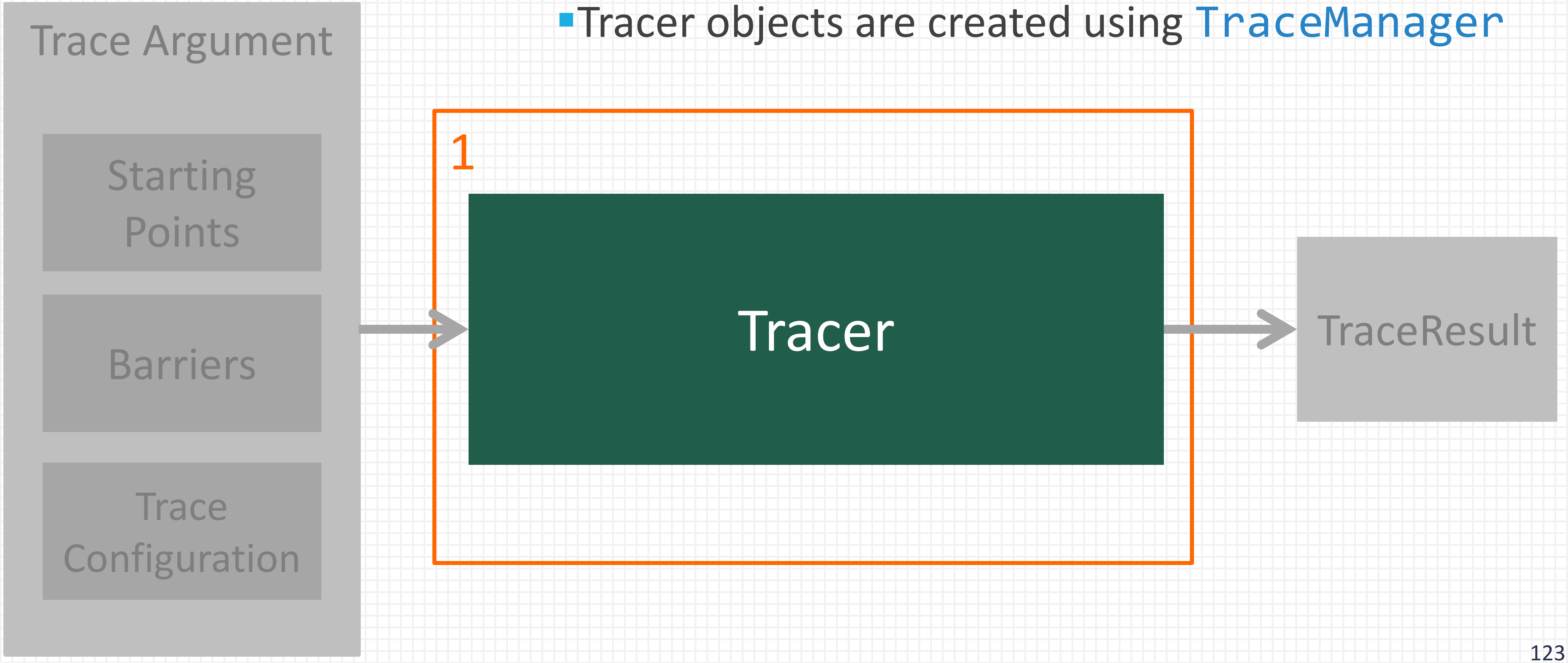
Anatomy of a Trace

We'll cover each of these parts in more detail



1 Tracer

- Tracers define the tracing algorithm to be used
- `Tracer` itself is an abstract base class
- Concrete subclasses implement specific tracing algorithms
- Tracer objects are created using `TraceManager`



The TraceManager class

- The `TraceManager` class is a central hub to the tracing portions of the SDK
- Trace manager objects are obtained through a call to `UtilityNetwork.GetTraceManager()`

`GetTracer<T>() : T`

- Returns a `Tracer` object of the specified type

The Tracer Abstract Class

`Name : string`

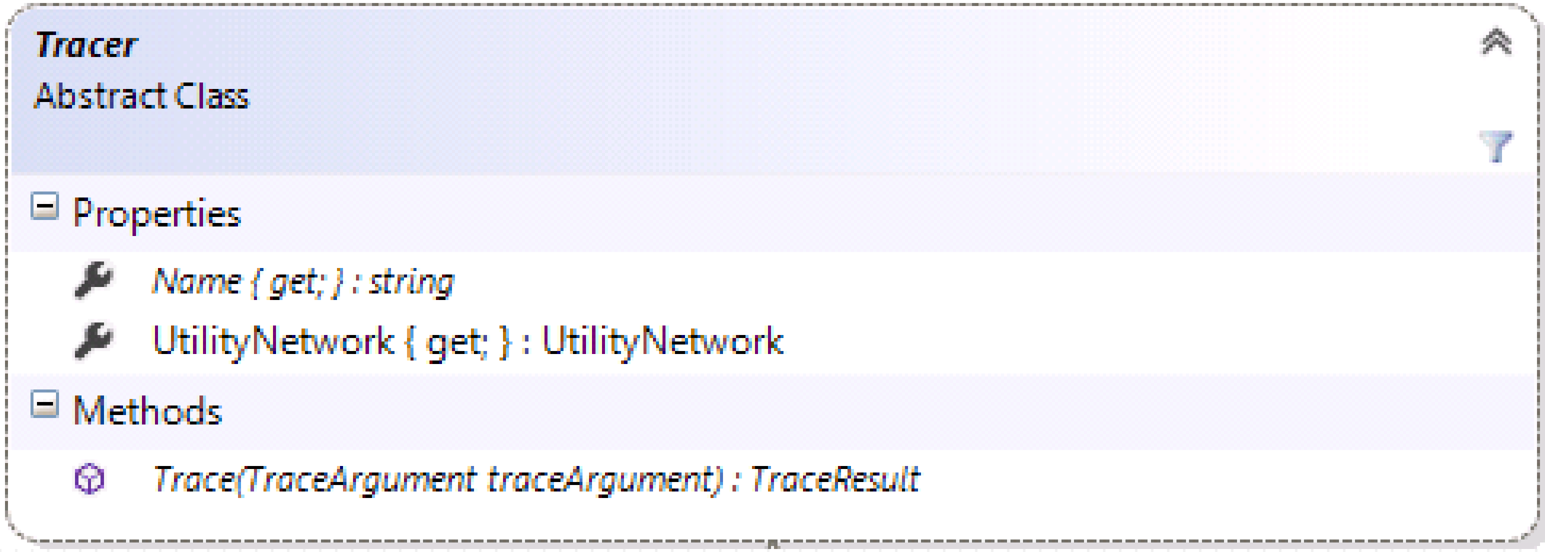
- The name of the Tracer object

`UtilityNetwork : UtilityNetwork`

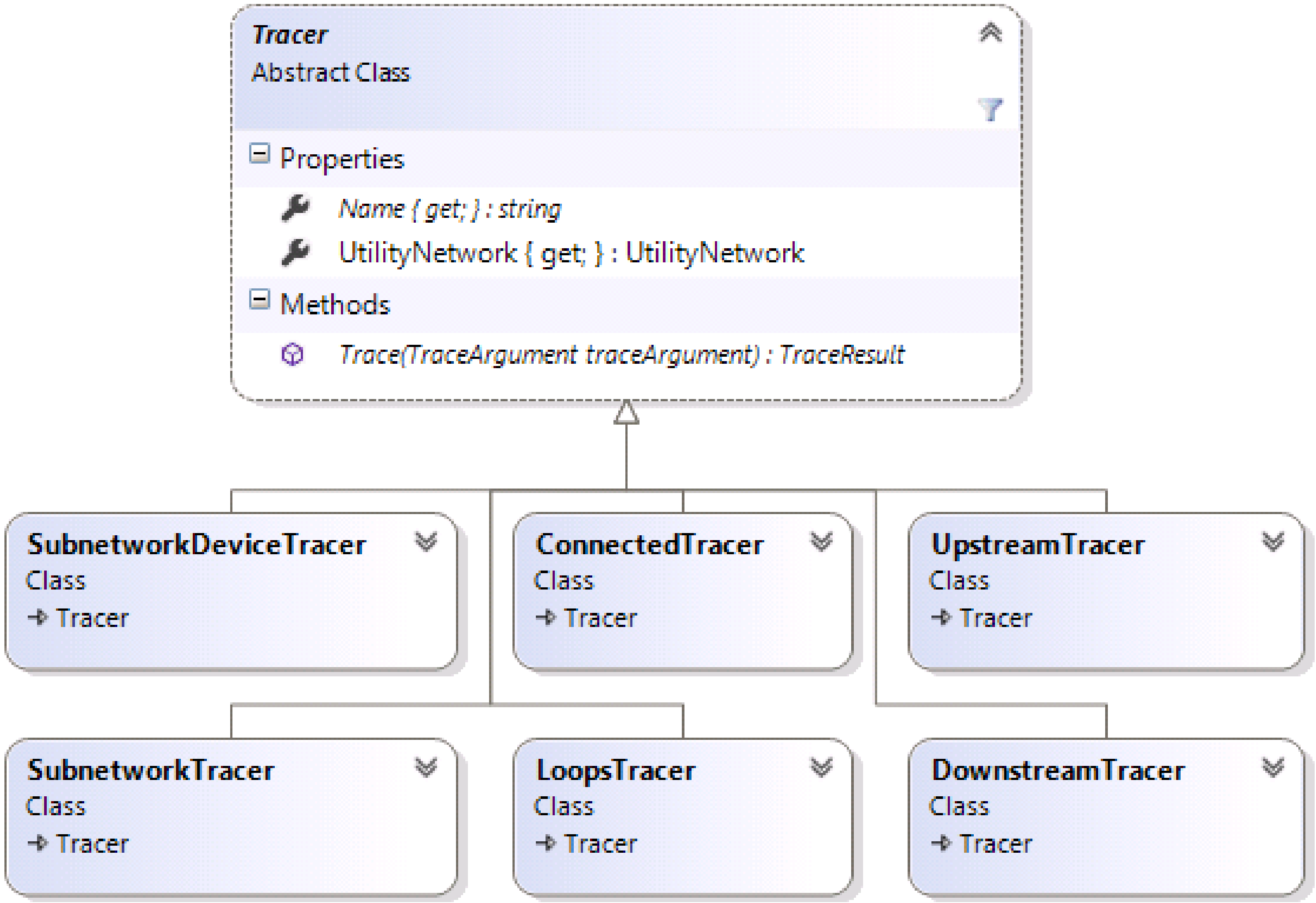
- A pointer back to the utility network core object

`Trace(TraceArgument traceArgument) : TraceResult`

- Perform a trace



Tracer Concrete Classes



Code Snippet: Creating a DownstreamTracer

```
TraceManager traceManager = utilityNetwork.GetTraceManager();

DownstreamTracer downstreamTracer = traceManager.GetTracer<DownstreamTracer>();
```

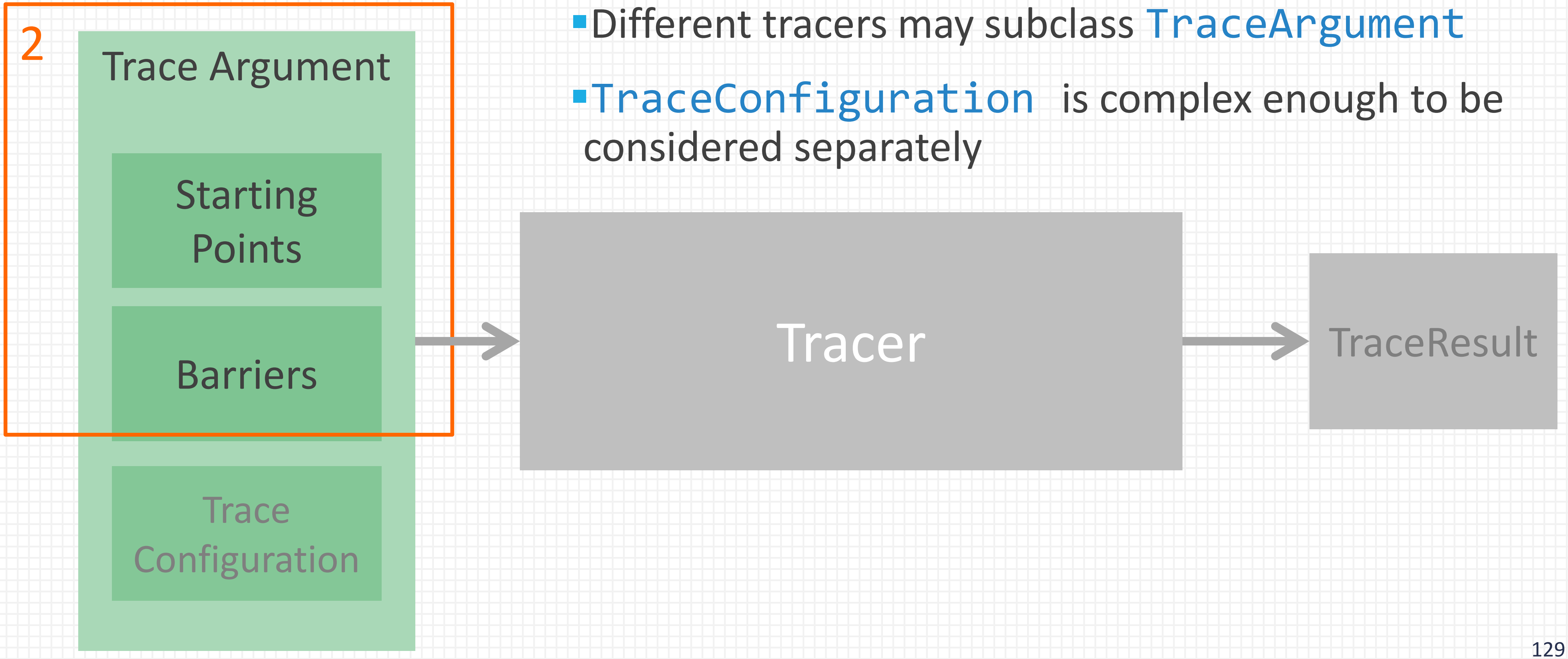
Extending Tracer

- 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` object 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

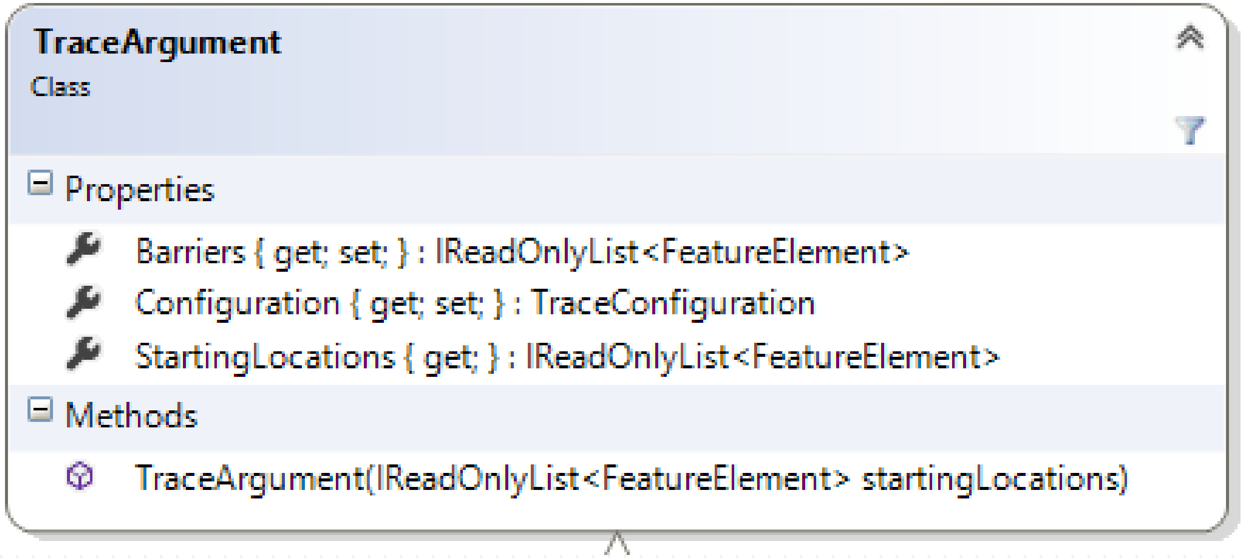
2 Trace Argument

- The `TraceArgument` class consolidates trace parameters
- Different tracers may subclass `TraceArgument`
- `TraceConfiguration` is complex enough to be considered separately



The TraceArgument Class

```
Barriers : IReadOnlyList<FeatureElement>
StartingLocations: IReadOnlyList<FeatureElement>
Configuration: TraceConfiguration
```

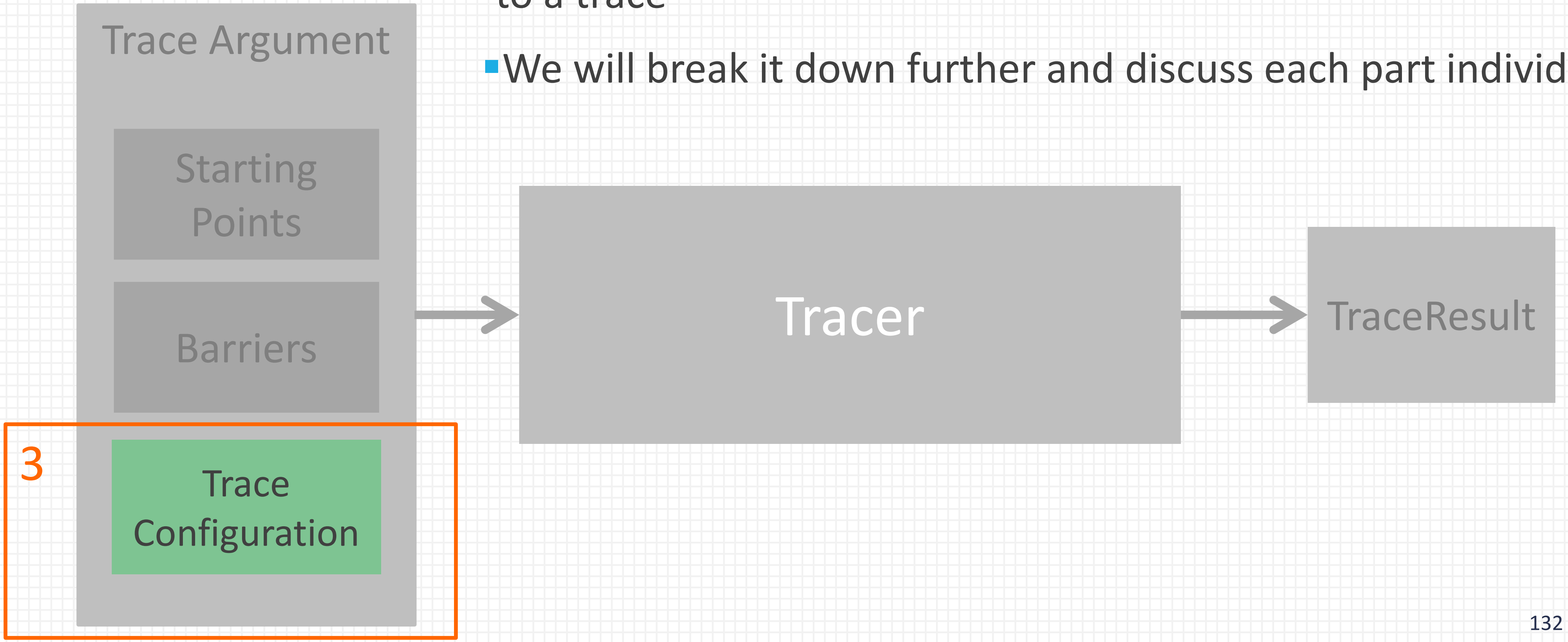


Code Snippet: Creating a TraceArgument

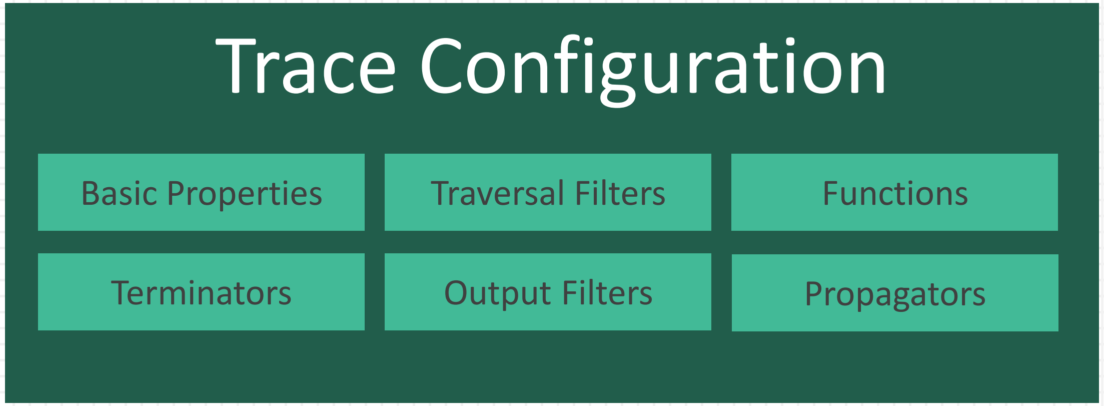
```
IReadOnlyList<FeatureElement> startingPointList = new List<FeatureElement>();

// Code to fill in list of starting points goes here...

TraceArgument traceArgument = new TraceArgument(startingPointList);

TraceConfiguration traceConfiguration = new TraceConfiguration();

// Code to fill in trace configuration goes here...

traceArgument.Configuration = traceConfiguration;
```

3 Trace Configuration

- The `TraceConfiguration` object holds all of the other inputs to a trace
- We will break it down further and discuss each part individually



Trace Configuration



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

Trace Configuration — Basic Properties

Trace Configuration

Basic Properties	Traversal Filters	Functions
Terminators	Output Filters	Propagators

`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
- `IncludeContainers` is recursive
  - i.e., if true, if a result element is inside a nested container, both containers are returned

Trace Configuration — Basic Properties (cont.)

Trace Configuration

Basic Properties	Traversal Filters	Functions
Terminators	Output Filters	Propagators

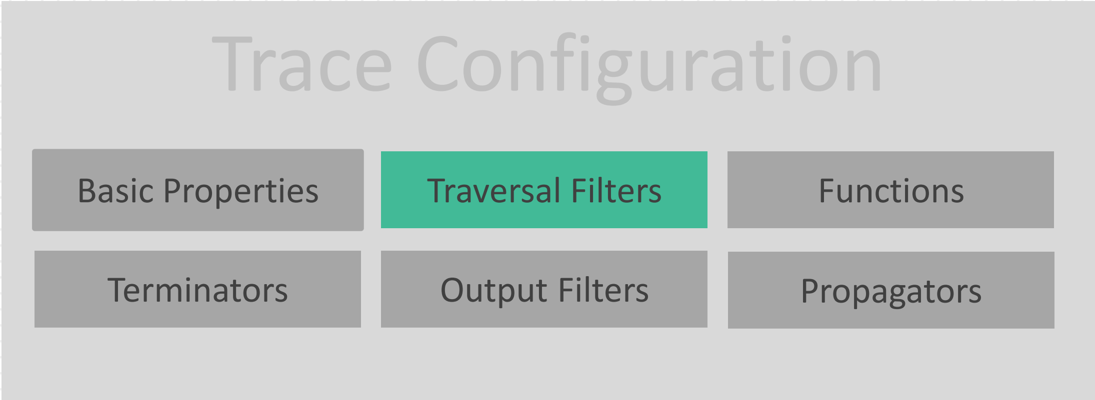
`SourceTier : Tier`

- This property is optional.
- If not null, the trace code 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

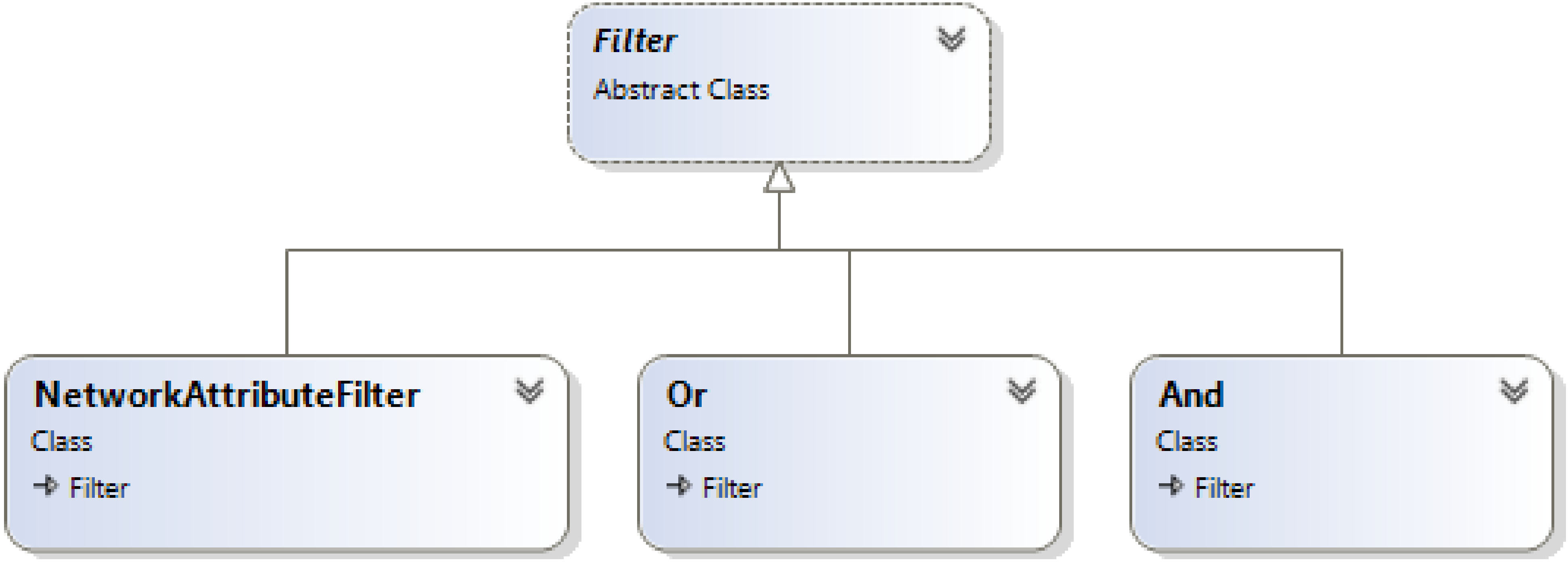
Trace Configuration — Traversal Filters



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

```
TraversalFilter : Filter
```

Traversal Filter Class Hierarchy



The NetworkAttributeFilter Class

- `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`

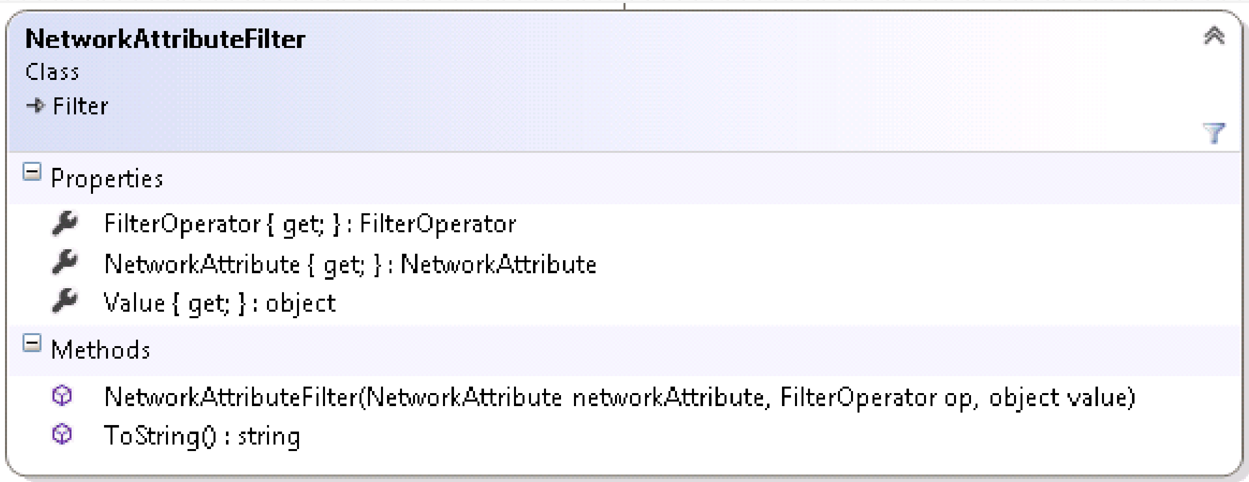
- Returns the boolean operator used to defined the filter. Possible values are `Equal`, `NotEqual`, `BitwiseAnd`, `GreaterThan`, `GreaterThanEqual`, `LessThan`, `LessthanEqual`, and `BitwiseAny`

`NetworkAttribute : NetworkAttribute`

- The network attribute to test

`Value : object`

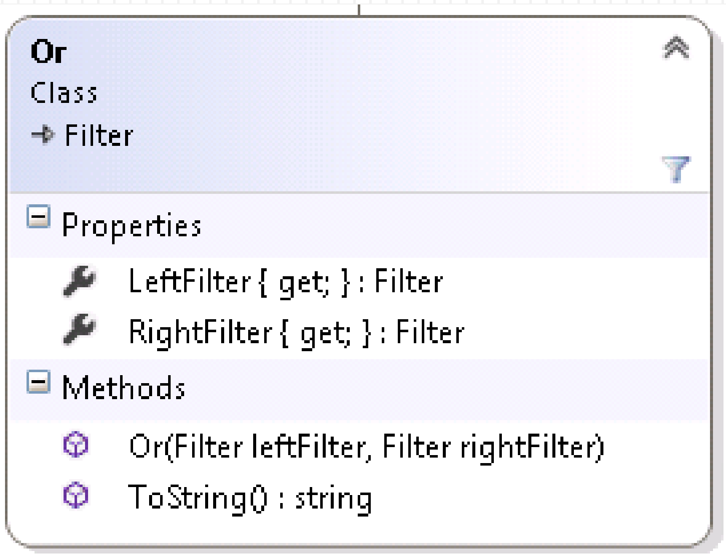
- The value to test against



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)
```



Code Snippet: Creating a Traversal Filter

```
// Get a NetworkAttribute object for the Lifecycle network attribute from the UtilityNetworkDefinition
NetworkAttribute lifecycleNetworkAttribute = utilityNetworkDefinition.GetNetworkAttribute("Lifecycle");

// Create a NetworkAttributeFilter to test Lifecycle = "In Design" (represented by the constant InDesign)
NetworkAttributeFilter inDesignNetworkAttributeFilter = new NetworkAttributeFilter(lifecycleNetworkAttribute, FilterOperator.Equal, InDesign);

// Create a NetworkAttributeFilter to test Lifecycle = "In Service" (represented by the constant InService)
NetworkAttributeFilter inServiceNetworkAttributeFilter = new NetworkAttributeFilter(lifecycleNetworkAttribute, FilterOperator.Equal, InService);

// Combine these two filters together with "Or"
Or lifecycleFilter = new Or(inDesignNetworkAttributeFilter, inServiceNetworkAttributeFilter);

// Final traversal filter tests Lifecycle = "In Design" or Lifecycle = "In Service"
traceConfiguration.TraversalFilter = lifecycleFilter;
```

Trace Configuration — Functions

Trace Configuration

Basic Properties	Traversal Filters	Functions
Terminators	Output Filters	Propagators

- The caller can specify a collection of functions for a trace
  - These functions calculate values based on a network attribute
- At the conclusion of the trace, function results can be obtained globally and for each applicable feature
  - Function results on an individual feature level are not yet implemented
  - Function results are described later in this document

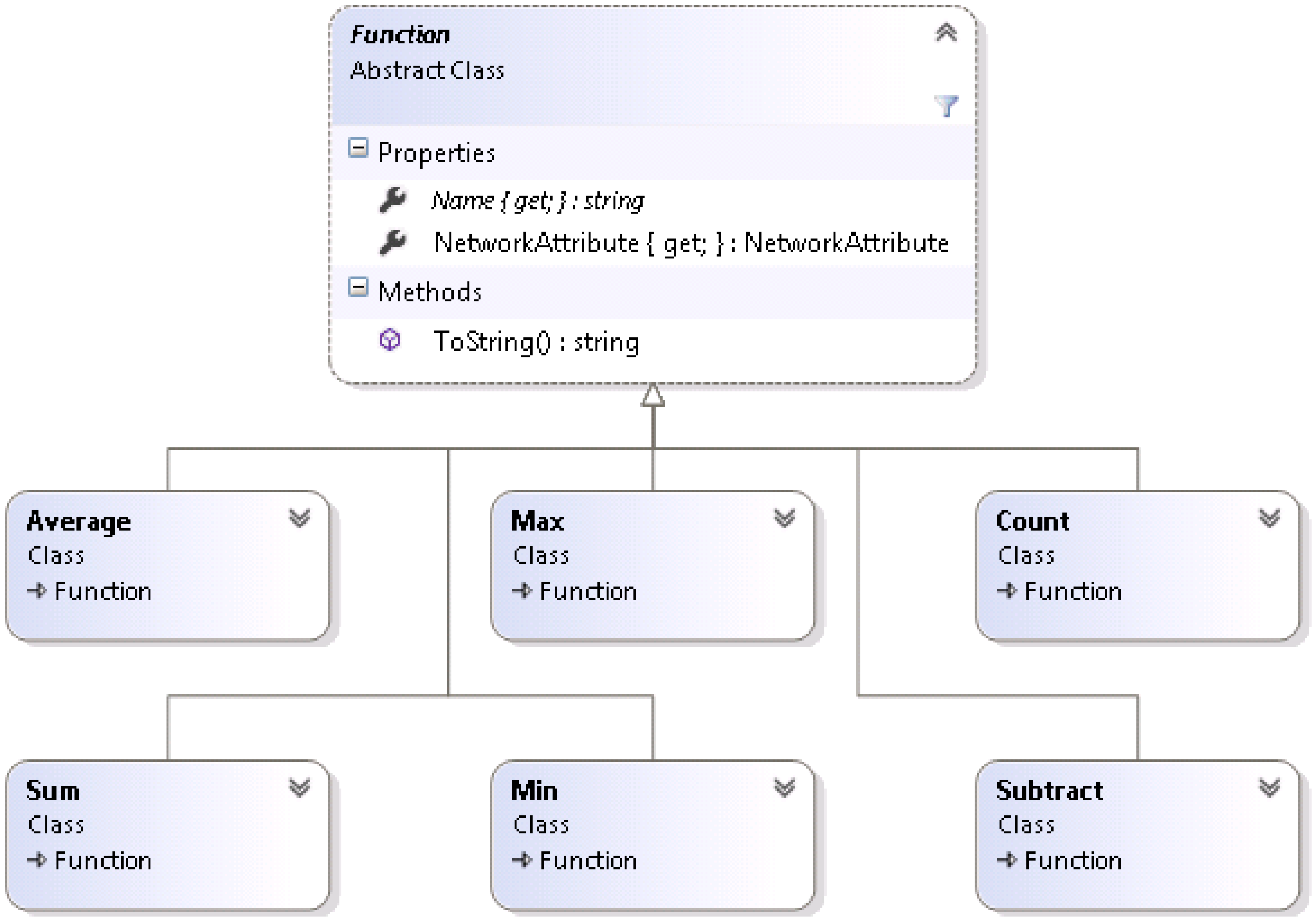
```
Functions : IReadOnlyList<Function>
```

How are Functions Applied to Features?

- These functions are evaluated at each *applicable* feature which has the assigned network attribute
- The meaning of *applicable* varies per trace
  - For an upstream trace, the functions are evaluated for each upstream feature
  - For a subnetwork trace, the functions are evaluated for each feature in the subnetwork
  - For a subnetwork controllers trace, the functions are evaluated for each subnetwork controller feature
- Note that functions are calculated *before* output filters are applied
  - Output filters are described later in this document

Function Class Hierarchy

- Custom functions may be supported in a future release



Meaning of Trace Functions

Function	Meaning
Sum	Sums up the value of the network attribute on each applicable feature
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 feature
Average	Averages the value of the network attribute on each applicable feature
Count	Counts the number of applicable features
Min	The minimum value of the network attribute on each applicable feature
Max	The maximum value of the network attribute on each applicable feature

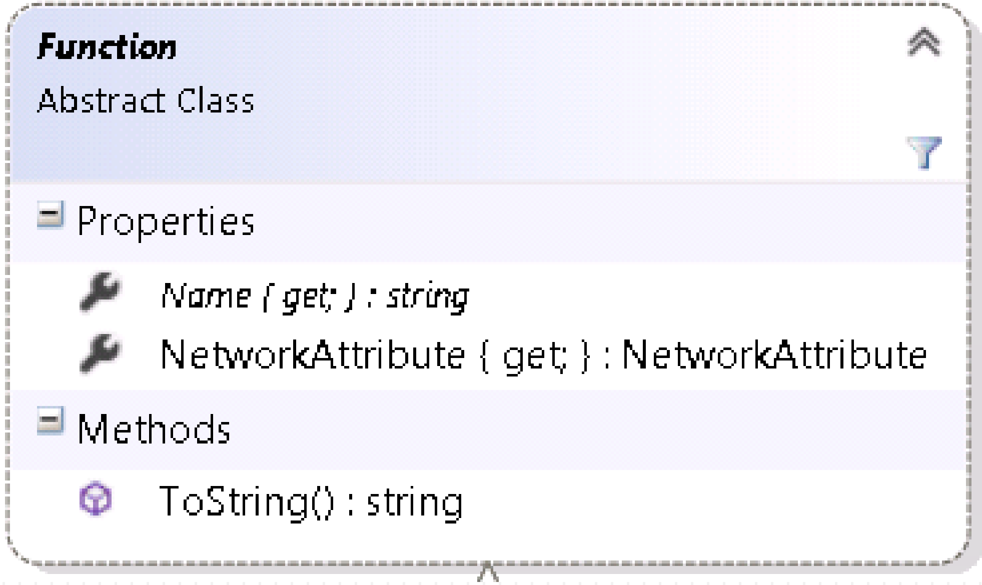
The Function Class

`NetworkAttribute : NetworkAttribute`

- The network attribute used as an input to the function

`Name: string`

- The name of the function



Code Snippet: Creating a Function

```
// Get a NetworkAttribute object for the Load network attribute from the UtilityNetworkDefinition
NetworkAttribute loadNetworkAttribute = utilityNetworkDefintion.GetNetworkAttribute("Load");

// Create a function to sum the Load
Sum sumLoadFunction = new Sum(loadNetworkAttribute);

// Add this function to our trace configuration
traceConfiguration.Functions = new List<Function>() { sumLoadFunction };
```

Trace Configuration — Terminators

Trace Configuration

Basic Properties	Traversal Filters	Functions
Terminators	Output Filters	Propagators

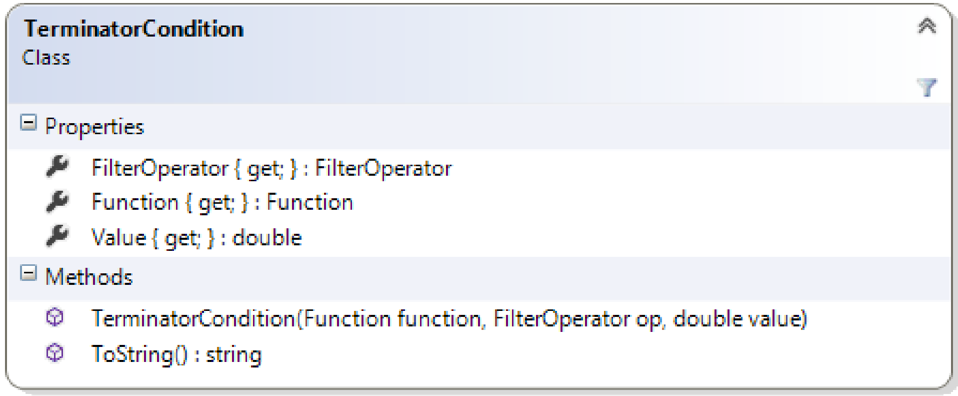
- Terminators provide a way to terminate further traversal when visiting a feature
- Teminators take three different forms
  - Terminator Categories
  - Terminator Conditions
  - Terminator Filters
- If **any** category or condition or filter is satisfied, further traversal is terminated
  - That particular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

Terminator Categories

`TerminatorCategories : IReadOnlyList<string>`

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

The TerminatorCondition Class



TerminatorConditions: IReadOnlyList<TerminatorCondition>

- Traversal stops if the TerminatorCondition evaluates to true
- Terminator conditions are based on comparing a function result to a specified value
- Teminator conditions 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 Filters

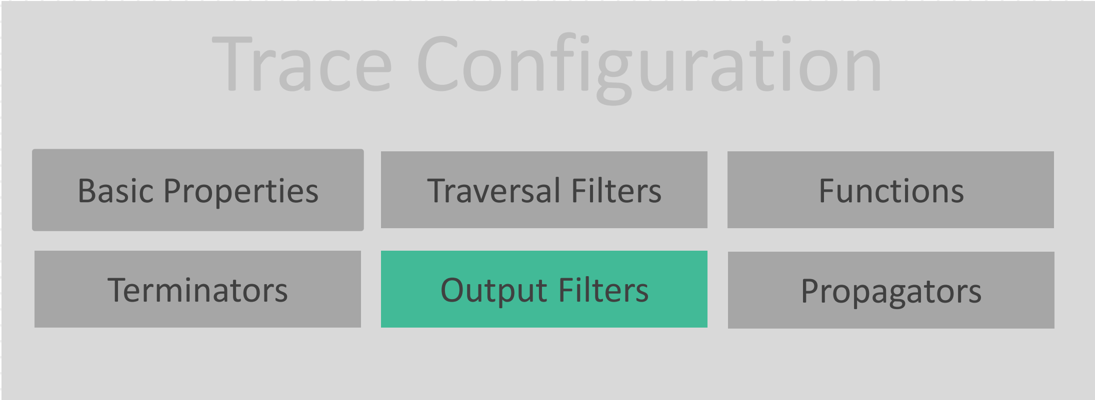
`TerminatorFilter : Filter`

- Terminator filters use the same `Filter` type as a `TraversalFilter`
- Terminator filters are **not** the same as negating a `TraversalFilter`
- For traversal filters…
  - If the visited element does not meet the filter criterial, 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

Code Snippet: Creating a Terminator Filter

```
// Get a NetworkAttribute object for the device status attribute from the UtilityNetworkDefinition
NetworkAttribute deviceStatusAttribute = utilityNetworkDefinition.GetNetworkAttribute("Device status");

// Create a filter to identify open devices (represented by the constant Open)
NetworkAttributeFilter deviceStatusFilter = new NetworkAttributeFilter(deviceStatusAttribute, FilterOperator.Equal, Open);

// Add this as a terminator filter to stop traversal at open devices
traceConfiguration.TerminatorFilter = deviceStatusFilter;
```

Trace Configuration — Output Filtering



- The TraceConfiguration 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

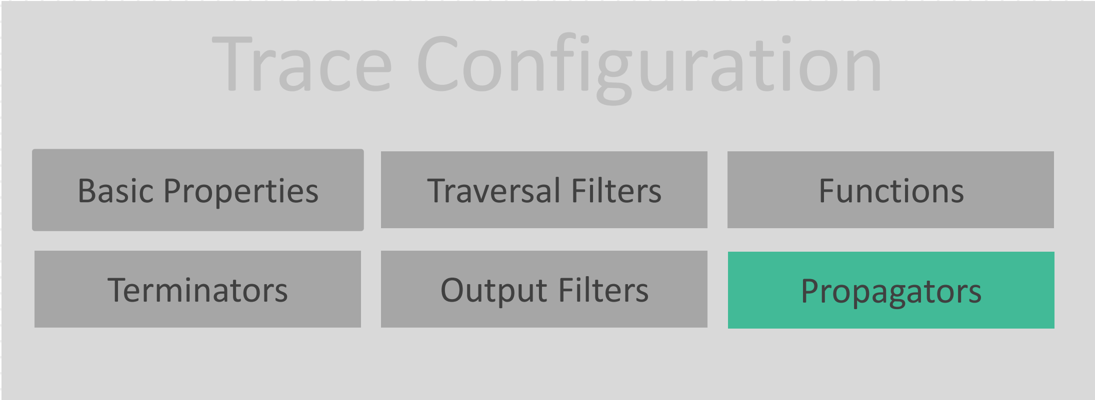
OutputAssetTypes: IReadOnlyList<AssetType>

- The network element is included in the result set if it references a feature that has any asset type in this list
- Using OutputCategories is the preferred technique to avoid hardcoding your add-in to a particular data model

Code Snippet: Creating an Output Category

```
// Create an output category to filter the trace results to only include
// features with the "Service Point" category assigned
traceConfiguration.OutputCategories = new List<string>() { "Service Point" };
```

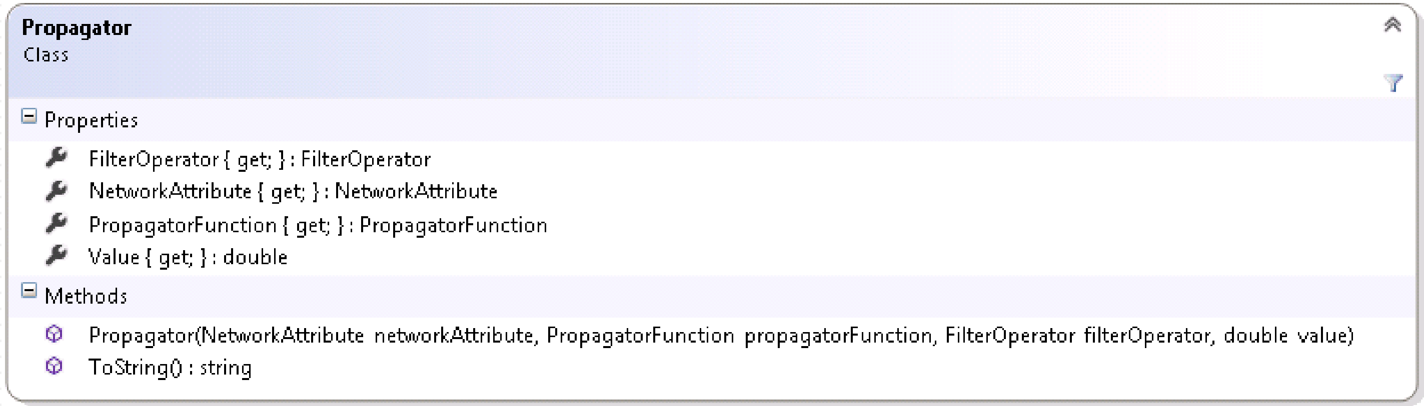
Trace Configuration — Propagators



- 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)
- The canonical example is phase propagation- open devices along the network will restrict some phases from continuing along the trace

```
Propagators : IReadOnlyList<Propagator>
```

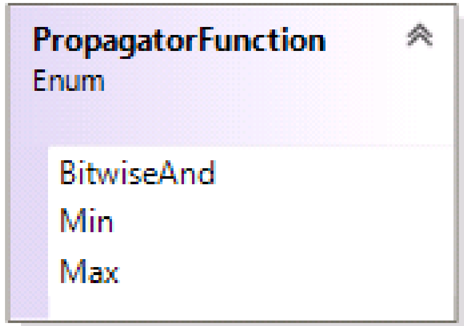
The Propagator Class



`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 on a per-feature basis
  - `TraceResult.PropagatorOutput`
  - This property is not yet implemented

Propagator Class Summary

Propagator
Class

Properties

- FilterOperator { get; } : FilterOperator
- NetworkAttribute { get; } : NetworkAttribute
- PropagatorFunction { get; } : PropagatorFunction
- Value { get; } : double

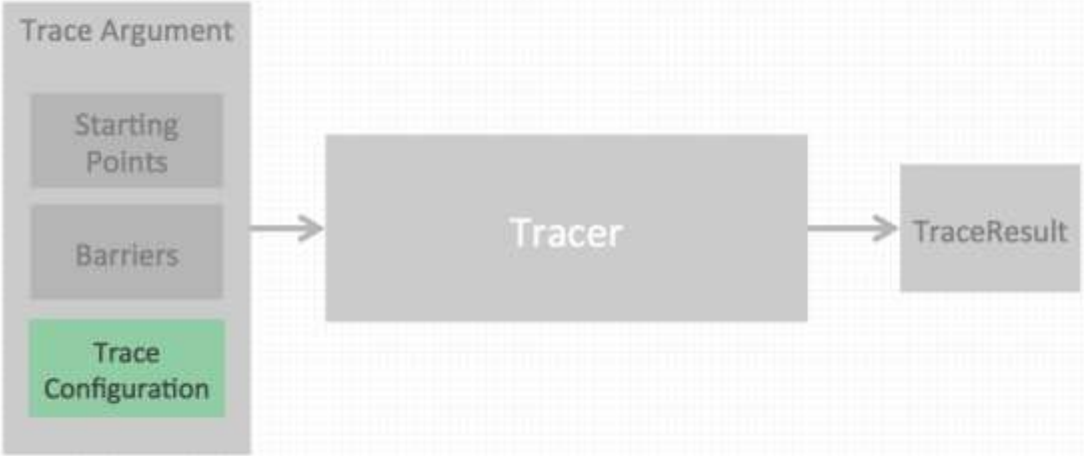
Methods

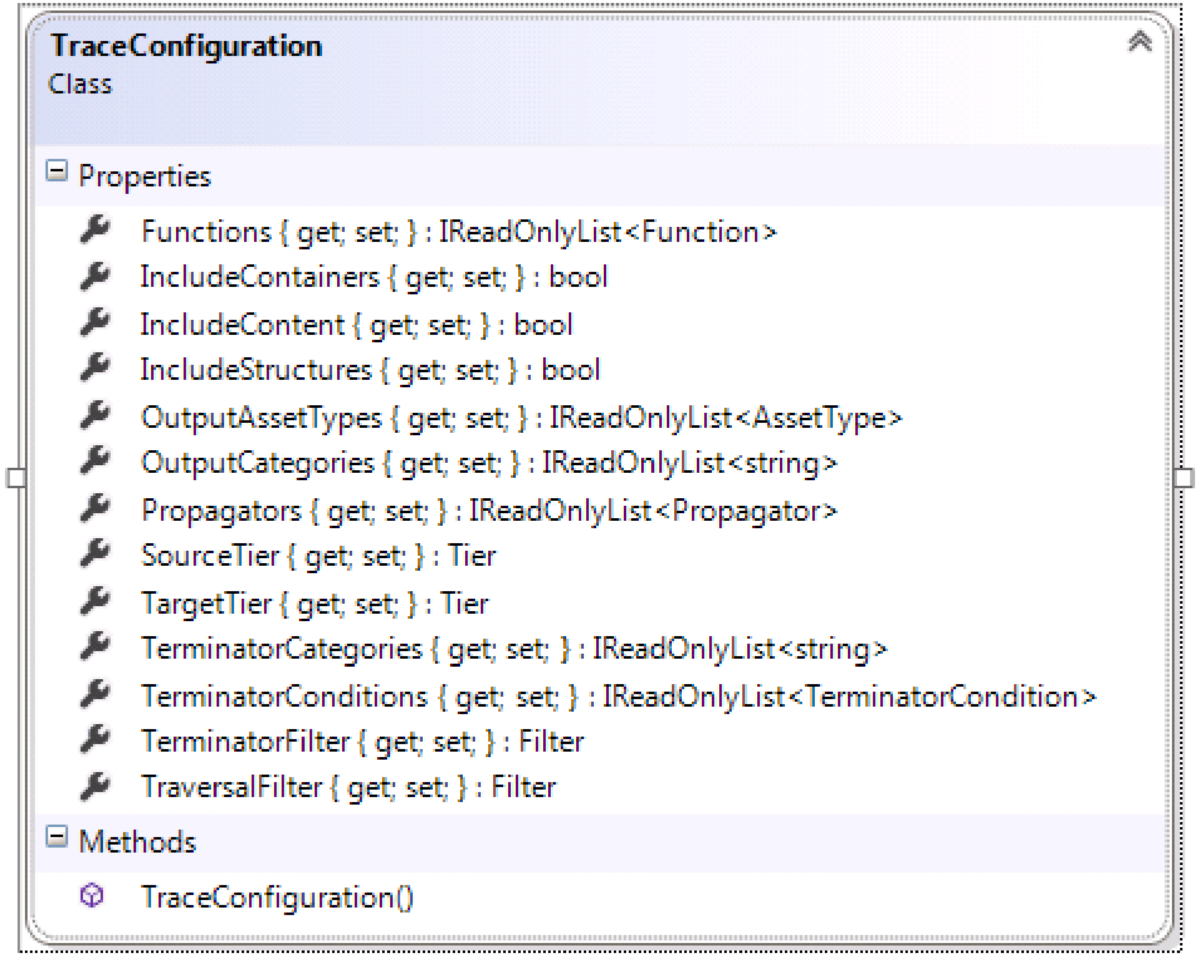
- Propagator(NetworkAttribute networkAttribute, PropagatorFunction propagatorFunction, FilterOperator filterOperator, double value)
- ToString() : string

Code Snippet: Creating a Phase Propagator

```
// Get a NetworkAttribute object for the Phases Normal attribute from the UtilityNetworkDefinition
NetworkAttribute normalPhaseAttribute = utilityNetworkDefinition.GetNetworkAttribute("Phases Normal");

// Create a propagator to propagate the Phases Normal attribute downstream from the source, using a Bitwise And function
// Allow traversal to continue as long as the Phases Normal value includes any of the ABC phases
// (represented by the constant ABCPhase)
Propagator phasePropagator = new Propagator(normalPhaseAttribute, PropagatorFunction.BitwiseAnd, FilterOperator.BitwiseAny, ABCPhase);

// Assign this propagtor to our trace configuration
traceConfiguration.Propagators = new List<Propagator>() { phasePropagator };
```

Trace Configuration — Class Summary

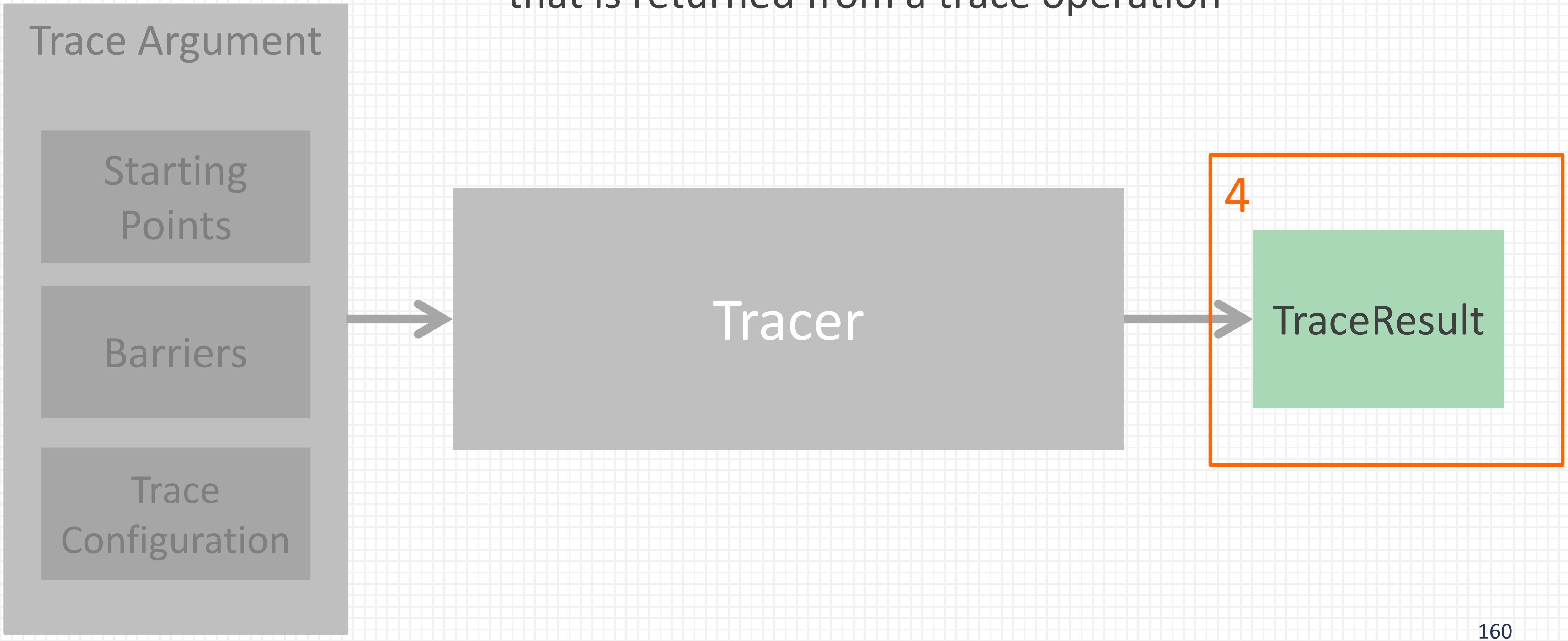




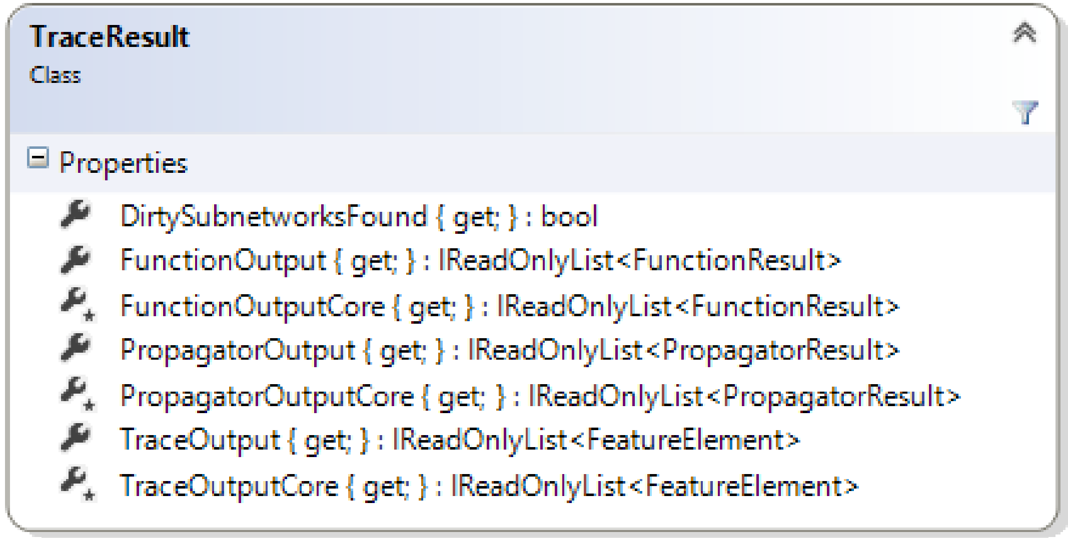
1. Basic Properties
2. Traversal Filters
3. Functions
4. Terminators
5. Output Filters
6. Propagators

4 Trace Result

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



The TraceResult Class



- The TraceResult class returns a set of information from the tracing operation

TraceOutput : IReadOnlyList<FeatureElement>

- The actual feature elements that are returned by the filter

DirtySubnetworksFound : bool

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

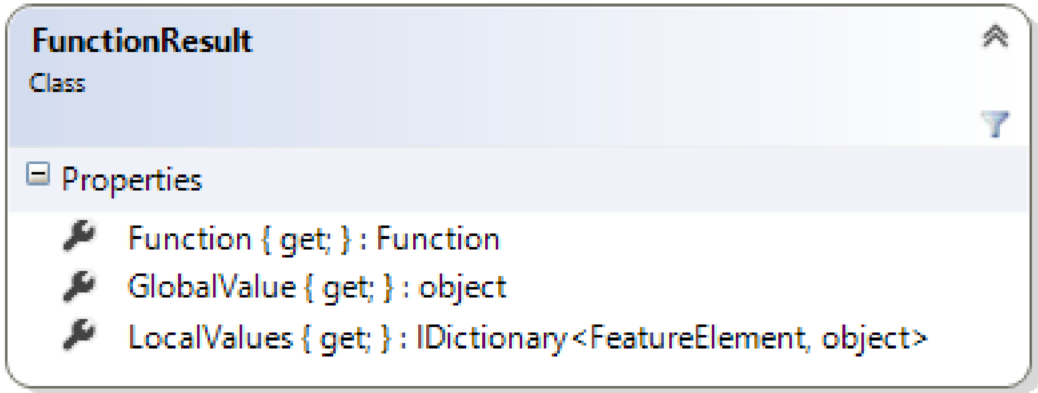
FunctionOutput : IReadOnlyList<FunctionResult>

- 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

The FunctionResult Class



- One `FunctionResult` object is returned for each `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 traversed features

`LocalValues : IDictionary<FeatureElement, object>`

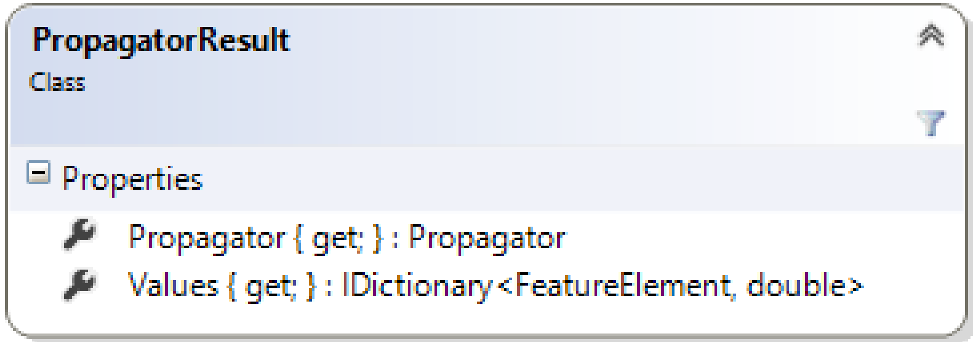
- Returns the result of the function at *each* feature element
- This functionality is not yet implemented

Code Snippet: Using Function Results

- This snippet uses the function that was created earlier in this presentation to sum the Load attribute

```
// Get the FunctionResult from the TraceResult
// First() can be used here because only one Function was included in the TraceConfiguration.Functions collection
FunctionResult loadFunctionResult = traceResult.FunctionOutput.First();

// Extract the total load from the GlobalValue property
double totalLoad = (double)loadFunctionResult.GlobalValue;
```

The PropagatorResult Class



- 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

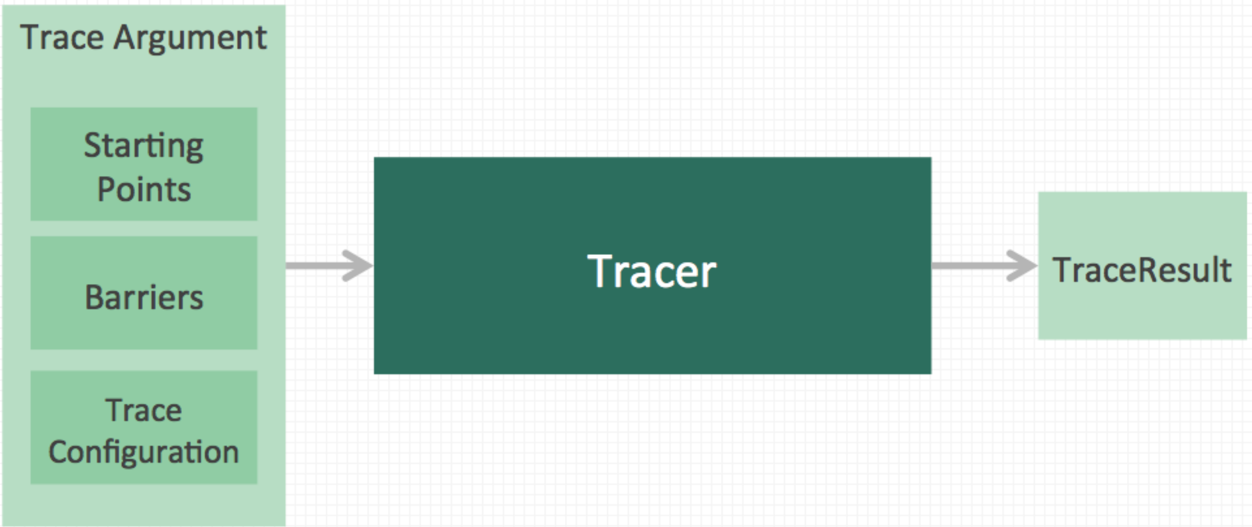
Extending the TraceResult class

- As previously discussed, developers can extend `Tracer` to create their own custom tracer object
- Likewise, developers can subclass the `TraceResult` class to override certain properties
  - `FunctionOutput`
  - `PropagatorOutput`
  - `TraceOutput`
- The `TraceResult` object provides three protected properties for developers who wish to access the core results

```
FunctionOutputCore : IReadOnlyList<FunctionResult>
PropagatorOutputCore : IReadOnlyList<PropagatorResult>
TraceOutputCore : IReadOnly<FeatureElement>
```

Putting it All Together

- A holistic code sample that demonstrates tracing functionality
  - Traversal filters
  - Terminator filters
  - Function
  - Output categories
- Outputs the total load and count of service points on a particular phase of an electric utility network



```csharp
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);
  }
}
```

The TraceConfiguration Class in Other Contexts

- In addition to its use with tracing, the `TraceConfiguration` class is used in other places in the SDK
- `TraceConfiguration` is a property on the `Tier` class
  - Assigned by the Set Subnetwork Definition geoprocessing tool
  - Defines the defaults for subnetwork traces, updates, and exports
- `Subnetwork.Update`
  - An override of `Subnetwork.Update()` allows the default trace configuration (taken from `Tier.TraceConfiguration`) to be overridden with custom settings
- These alternative uses of the `TraceConfiguration` class are described in the slides that follow

TraceConfiguration Property on Tier

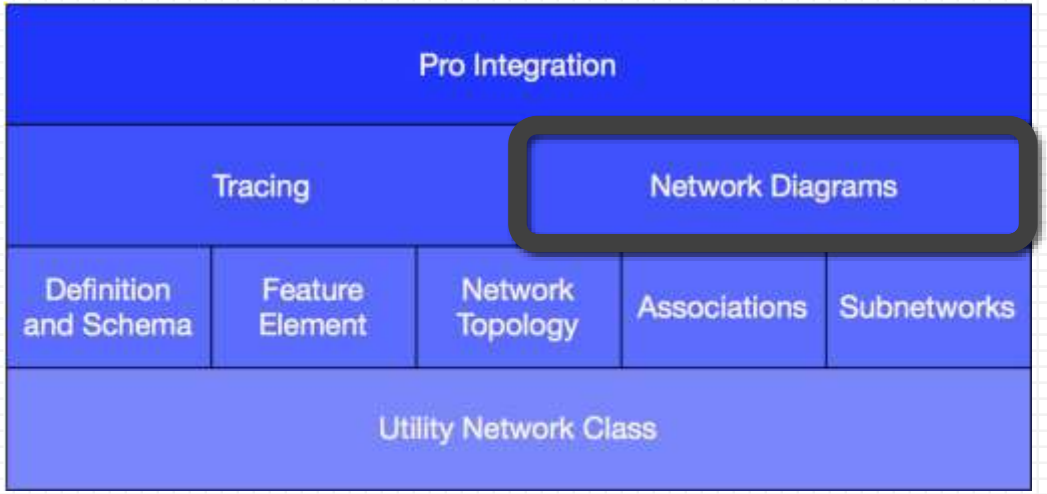
- The `Tier.TraceConfiguration` property returns the default tracing parameters for a given tier
- Not all the `TraceConfiguration` properties apply in this scenario
- Non-applicable properties are left empty, as shown in the table at right

Property	Applicable
Functions	✓
IncludeContainers	
IncludeContent	
IncludeStructures	
OutputAssetTypes	
OutputCategories	
Propagators	✓
SourceTier	
TargetTier	
TerminatorCategories	✓
TerminatorConditions	✓
TerminatorFilter	✓
TraversalFilter	✓

TraceConfiguration Parameter on Subnetwork.Update()

- `Subnetwork.Update()` typically uses the default tracing configuration defined by `Tier.TraceConfiguration`
- This can be overridden with the `Subnetwork.Update( TraceConfiguration traceConfiguration)`
- Not all the `TraceConfiguration` properties apply in this scenario
- Non-applicable properties are left empty, as shown in the table at right

Property	Applicable
Functions	
IncludeContainers	
IncludeContent	
IncludeStructures	
OutputAssetTypes	
OutputCategories	
Propagators	✓
SourceTier	
TargetTier	
TerminatorCategories	✓
TerminatorConditions	✓
TerminatorFilter	✓
TraversalFilter	✓



Network Diagrams

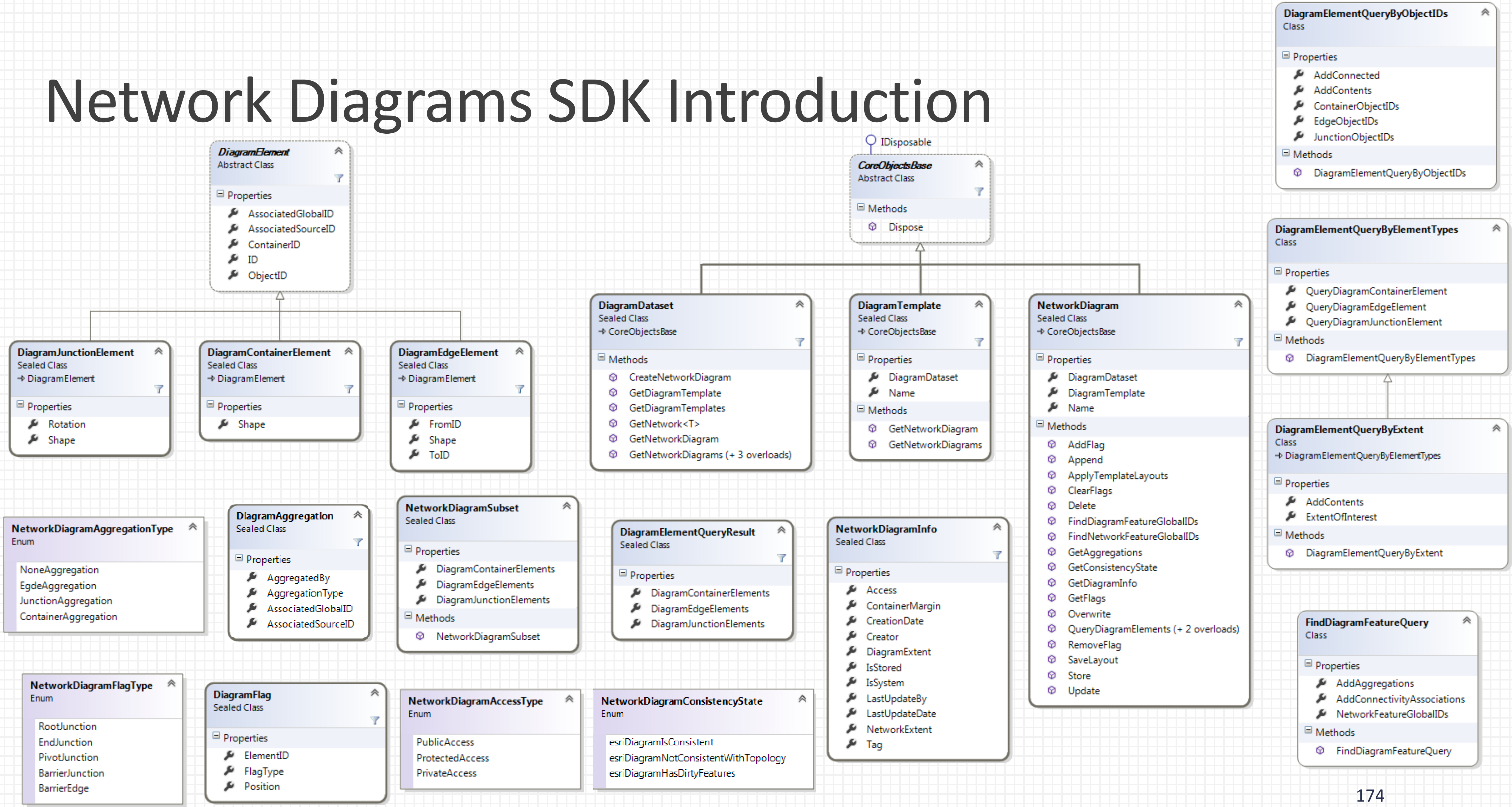
Network Diagrams SDK Introduction

- Provides access to the network diagrams framework, allowing developers to manage network diagrams
- Provides classes to allow:
  - Retrieving existing diagram templates
  - Creating, storing, and deleting network diagrams
  - Retrieving stored network diagrams that cover a given extent or containing a particular utility network element
  - Updating, overwriting, and appending network diagrams
  - Coding custom layout on a network diagram

What the Network Diagram SDK Doesn't Do

- 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`

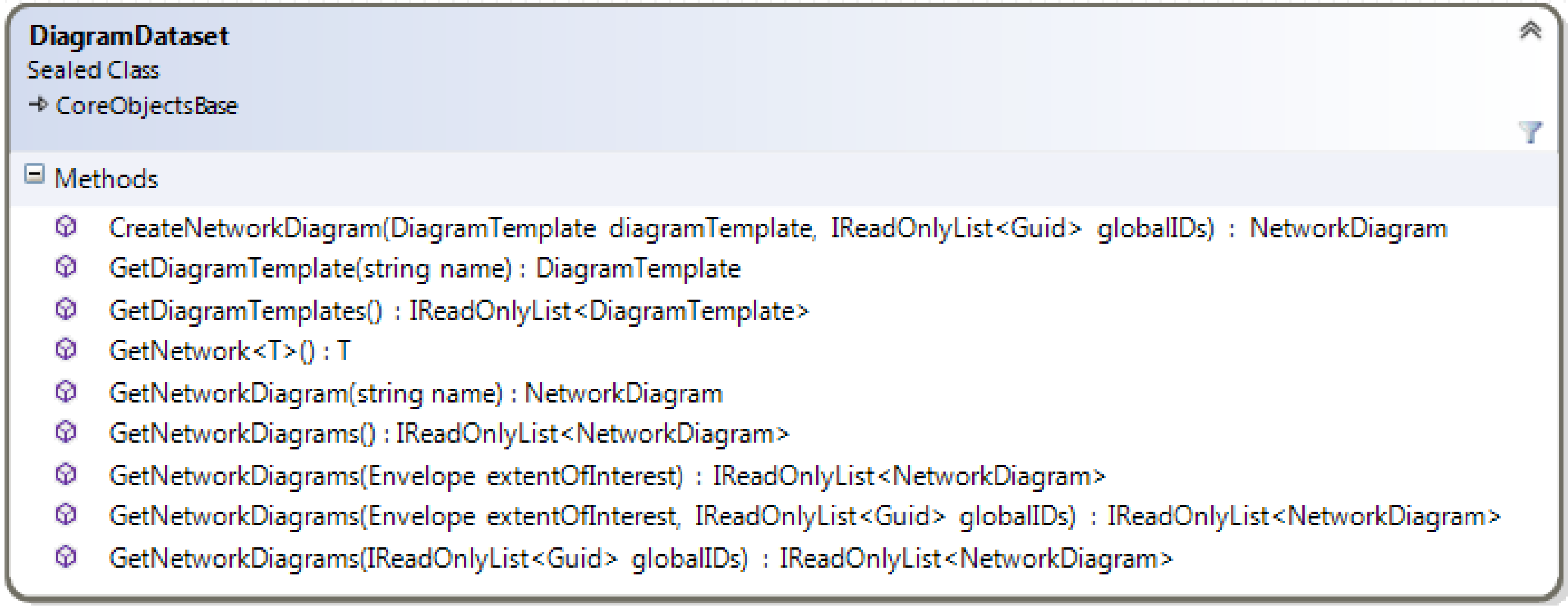
Network Diagrams SDK Introduction



1. Diagram Dataset

The DiagramDataset Class

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



DiagramDataset — Retrieve Related Network

`GetNetwork<T>() : T`

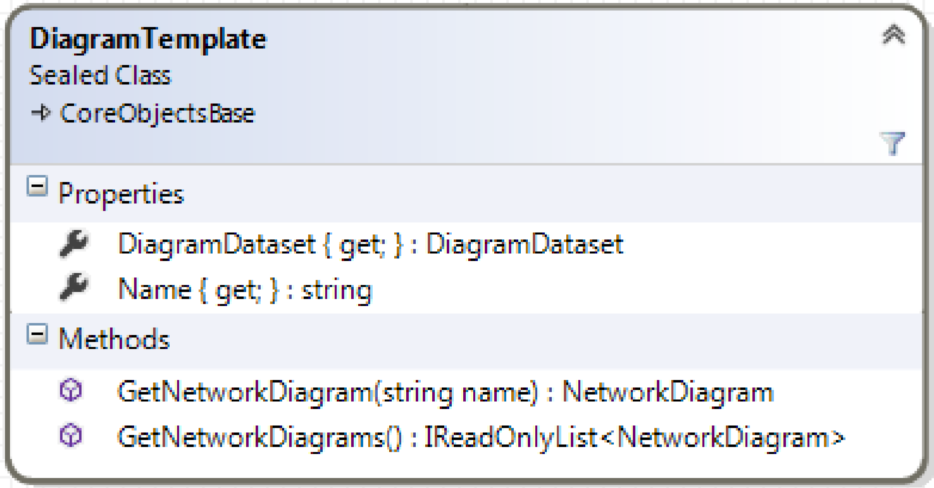
- This routine returns a network object of the specified type to which the diagram dataset is related

2. Diagram Templates

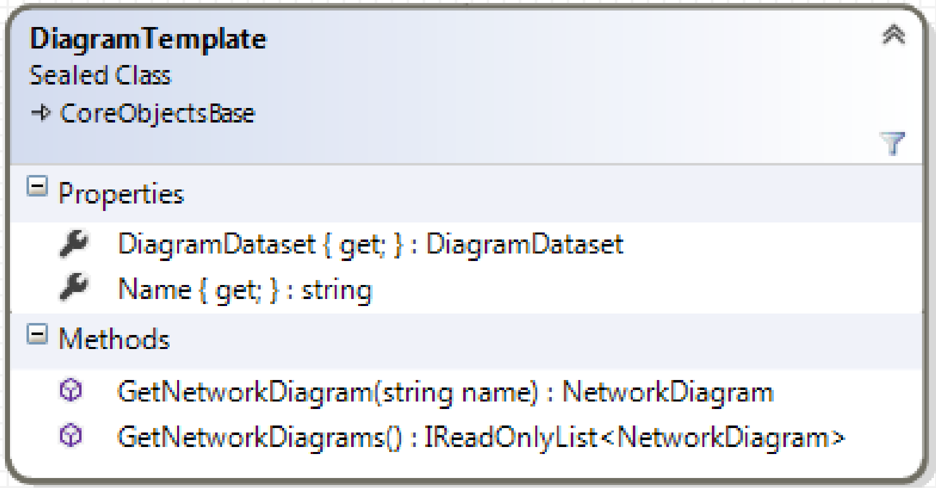
The DiagramTemplate Class

- A diagram template holds the configuration properties defining the content (diagram builder definition), and presentation (diagram layer definition) of a type of network diagrams

- It is obtained from the diagram dataset using:
  - `DiagramDataset.GetDiagramTemplates`
  - `DiagramDataset.GetDiagramTemplate(string name)`



The DiagramTemplate Class



DiagramDataset : DiagramDataset

- Returns the DiagramDataset related to the diagram template

Name : string

- Returns the name of the diagram template

GetNetworkDiagrams() : IReadOnlyList<NetworkDiagram>

- To retrieve all the network diagrams based on the diagram template

GetNetworkDiagram(string name) : NetworkDiagram

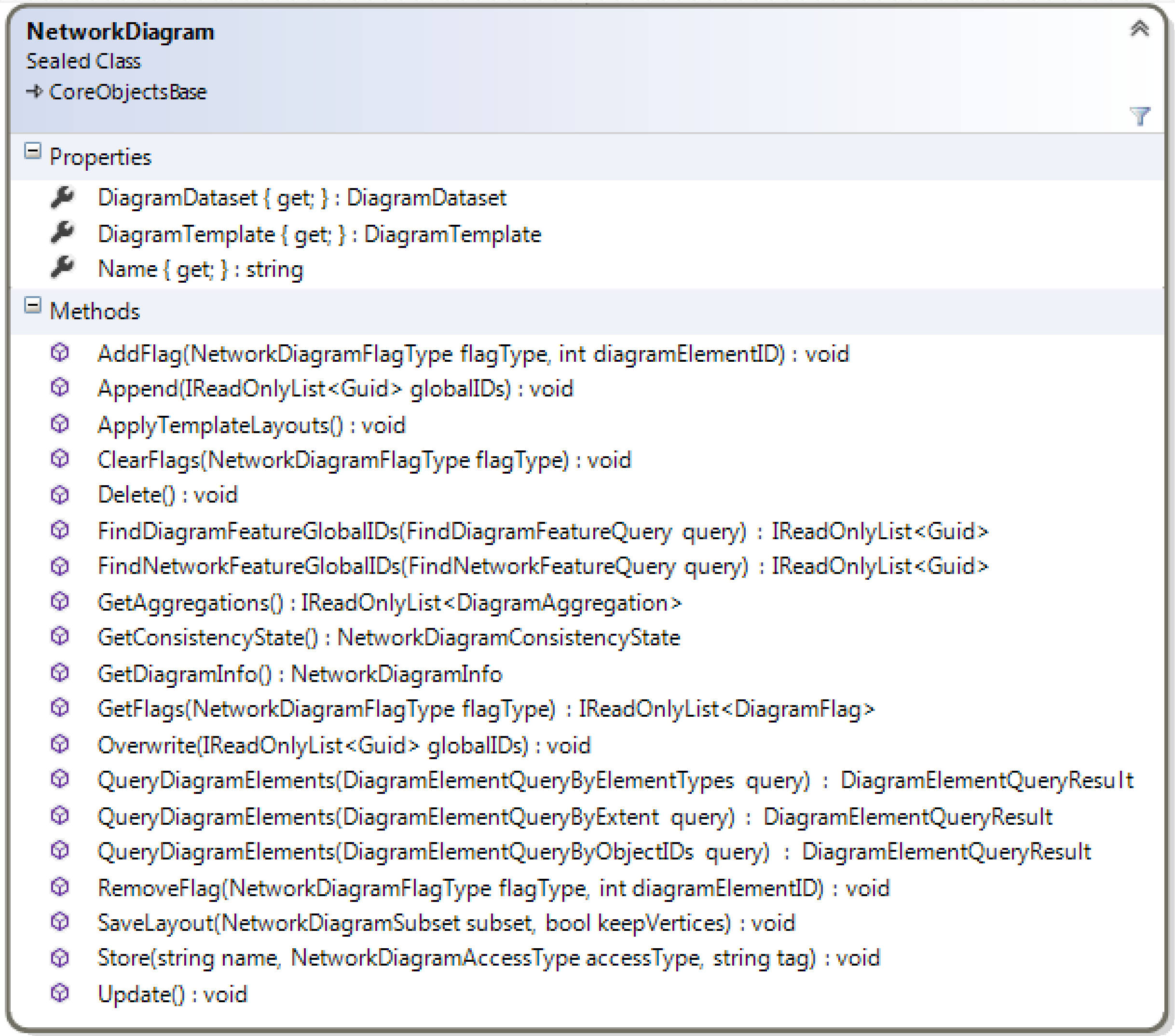
- To retrieve a particular network diagram based on the diagram template using its name

3. Network Diagram

Basics

The NetworkDiagram Class

- This class represents a diagram generated from a portion of the utility network



Creating Network Diagrams

- New network diagrams can be created using a factory method on the diagram dataset
- `DiagramDataset.CreateNetworkDiagram(DiagramTemplate diagramTemplate,`
  `IReadOnlyList<Guid> globalIDs) : NetworkDiagram`
    - This routine creates a temporary network diagram from a set of utility network feature GlobalIDs

Retrieving Existing Network Diagrams

- Network diagrams that already exist in the diagram dataset can be retrieved
  - Directly from the diagram dataset
    - `DiagramDataset.GetNetworkDiagram(string name): NetworkDiagram`
    - `DiagramDataset.GetNetworkDiagrams(): IReadOnlyList<NetworkDiagram>`
    - `DiagramDataset.GetNetworkDiagrams(Envelope extentOfInterest): IReadOnlyList<NetworkDiagram>`
    - `DiagramDataset.GetNetworkDiagrams(IReadOnlyList<Guid> globalIDs): IReadOnlyList<NetworkDiagram>`
    - `DiagramDataset.GetNetworkDiagrams(Envelope extentOfInterest, IReadOnlyList<Guid> globalIDs): IReadOnlyList<NetworkDiagram>`
  - From the diagram template they are based on
    - `DiagramTemplate.GetNetworkDiagram(string name) : NetworkDiagram`
    - `DiagramTemplate.GetNetworkDiagrams() : IReadOnlyList<NetworkDiagram>`

NetworkDiagram — Basics

```
DiagramDataset: DiagramDataset
DiagramTemplate: DiagramTemplate
Name: string
```

- Three read-only properties on the `NetworkDiagram` class to get
  - the diagram dataset related to the network diagram,
  - the diagram template it is based on,
  - the name of the network diagram

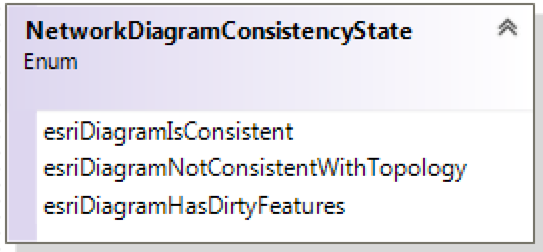
NetworkDiagram — Accessing Diagram Properties

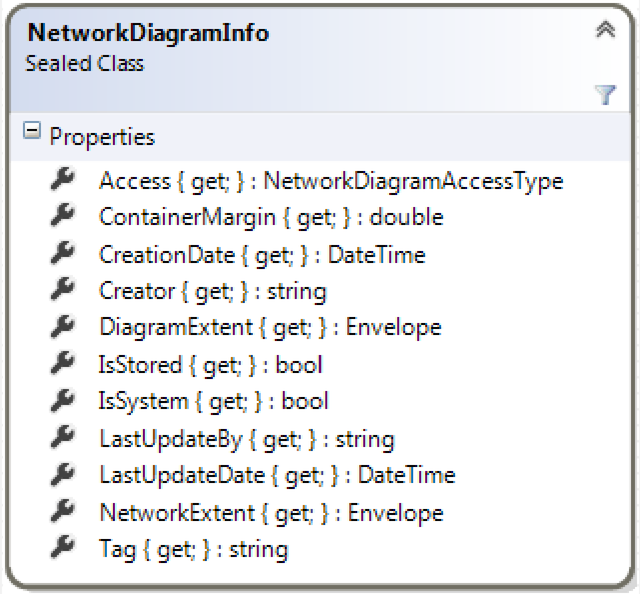
`GetConsistencyState(): NetworkDiagramConsistencyState`

- Returns the consistency state of the network diagram

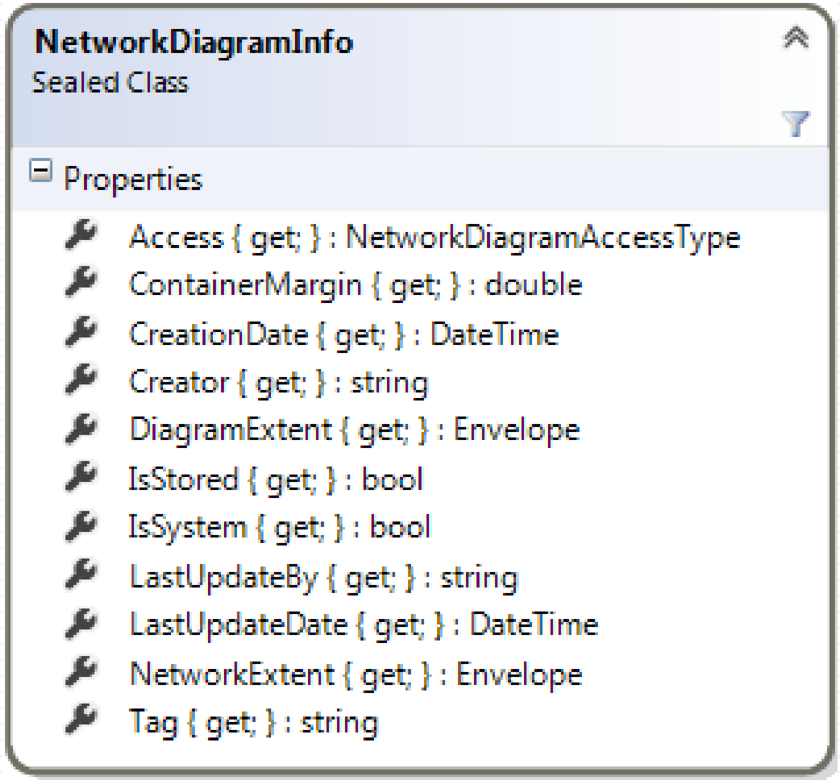
`GetDiagramInfo(): NetworkDiagramInfo`

- Returns all info related to the network diagram, `NetworkDiagramInfo`



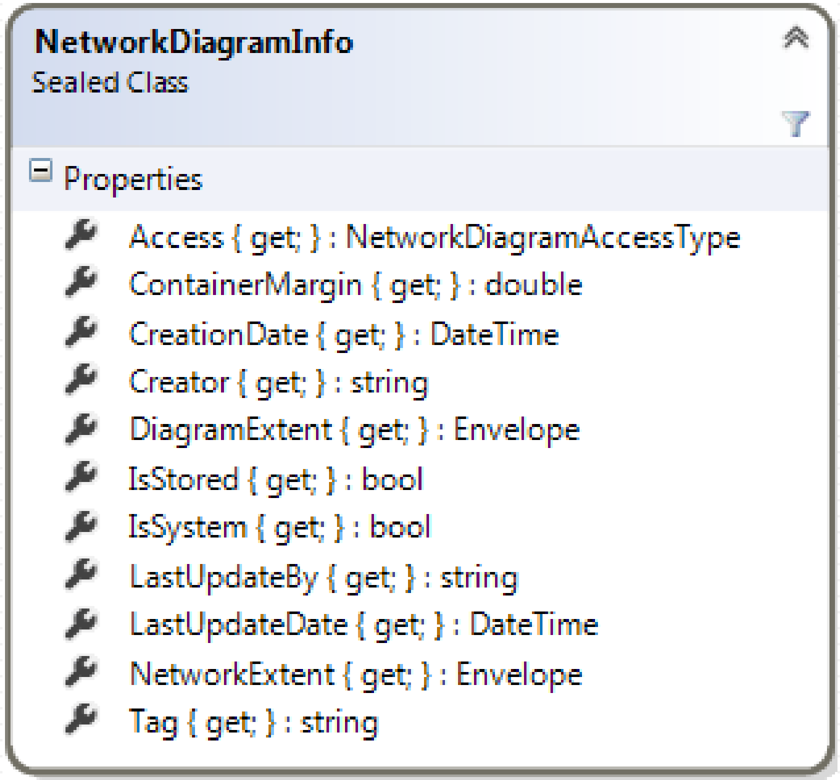


The NetworkDiagramInfo Class — 1



- `Access`
- `ContainerMargin`
  - The internal margin of the diagram's container polygon
- `IsStored`
- `IsSystem`
  - Returns true if this is system-generated subnetwork diagram
- `Tag`

The NetworkDiagramInfo Class — 2



`CreationDate`

`Creator`

`LastUpdateDate`

`LastUpdateBy`

`DiagramExtent`

- The extent of the diagram itself

`NetworkExtent`

The extent formed by the set of utility network features that participated in the diagram creation, whether or not those features are represented in the diagram

Code Snippet: Retrieving Inconsistent Non-system Diagrams

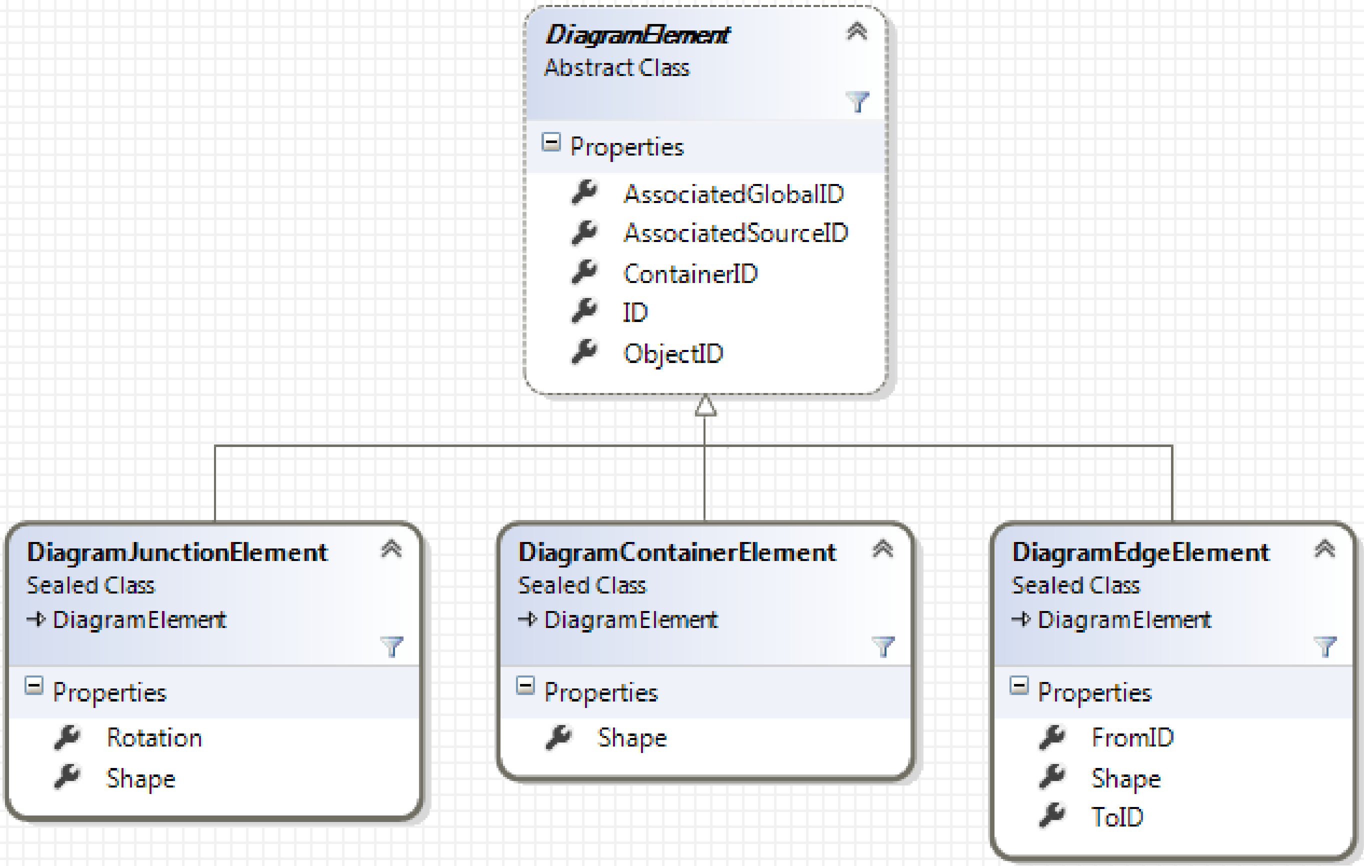
```
public List<NetworkDiagram> GetInconsistentDiagrams(UtilityNetwork unNetwork)
{
    List<NetworkDiagram> myList = new List<NetworkDiagram>();
    {
        DiagramDataset diagramDataset = unNetwork.GetDiagramDataset();

        // Getting the list of Diagram Templates
        IReadOnlyList<DiagramTemplate> listTemplate = diagramDataset.GetDiagramTemplates();

        foreach (DiagramTemplate diagTemplate in listTemplate)
        {
            // Getting the list of the Network Diagrams per template
            IReadOnlyList<NetworkDiagram> listDiagram = diagTemplate.GetNetworkDiagrams();

            // Analyzing each network diagram to filter out the non-system diagrams which are inconsistent
            foreach (NetworkDiagram nd in listDiagram)
            {
                NetworkDiagramInfo networkDiagramInfo = nd.GetDiagramInfo();
                if (!networkDiagramInfo.IsSystem && nd.GetConsistencyState() != NetworkDiagramConsistencyState.esriDiagramIsConsistent)
                    myList.Add(nd);
            }
        }
    }
    return myList;
}
```

3. Network Diagram

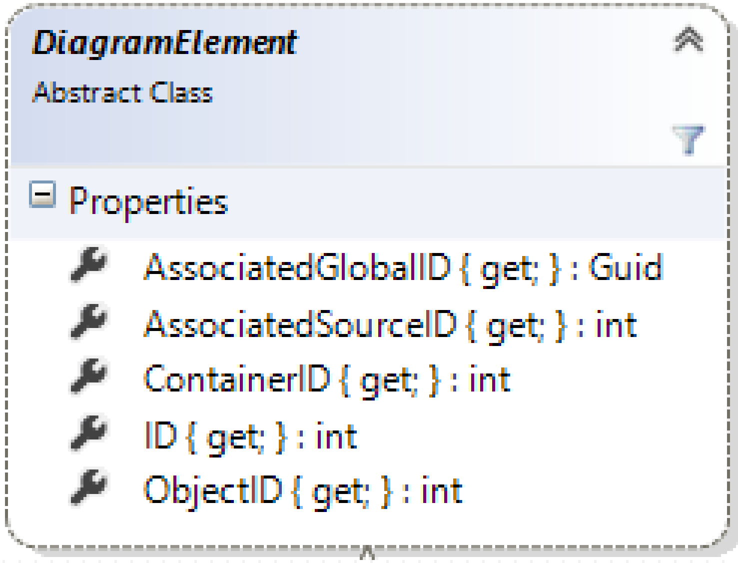
Diagram Elements

Diagram Element Classes

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



The DiagramElement Base Class



`AssociatedGlobalID : Guid`

`AssociatedSourceID : int`

- These two properties retrieve the utility network feature/network source for the diagram element

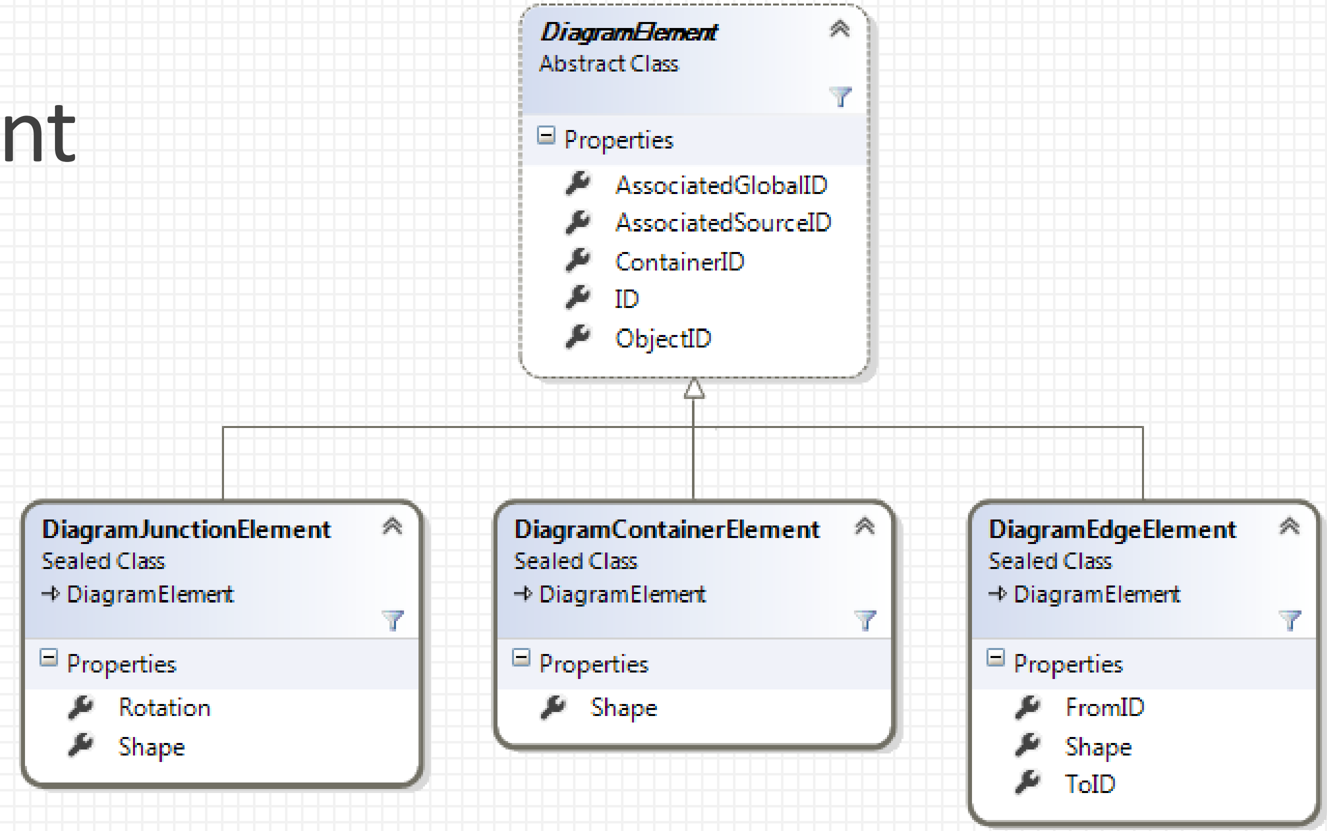
`ContainerID : int`

- Returns the internal diagram id of the diagram container element
- Returns 0 if this diagram element is not content

`ID: int`

- Returns the internal diagram id of the element

Subclasses of DiagramElement



`Shape`

- The geometry of the diagram element

`Rotation`

- The rotation applied to a diagram element (`DiagramJunctionElement` only)

`FromID`

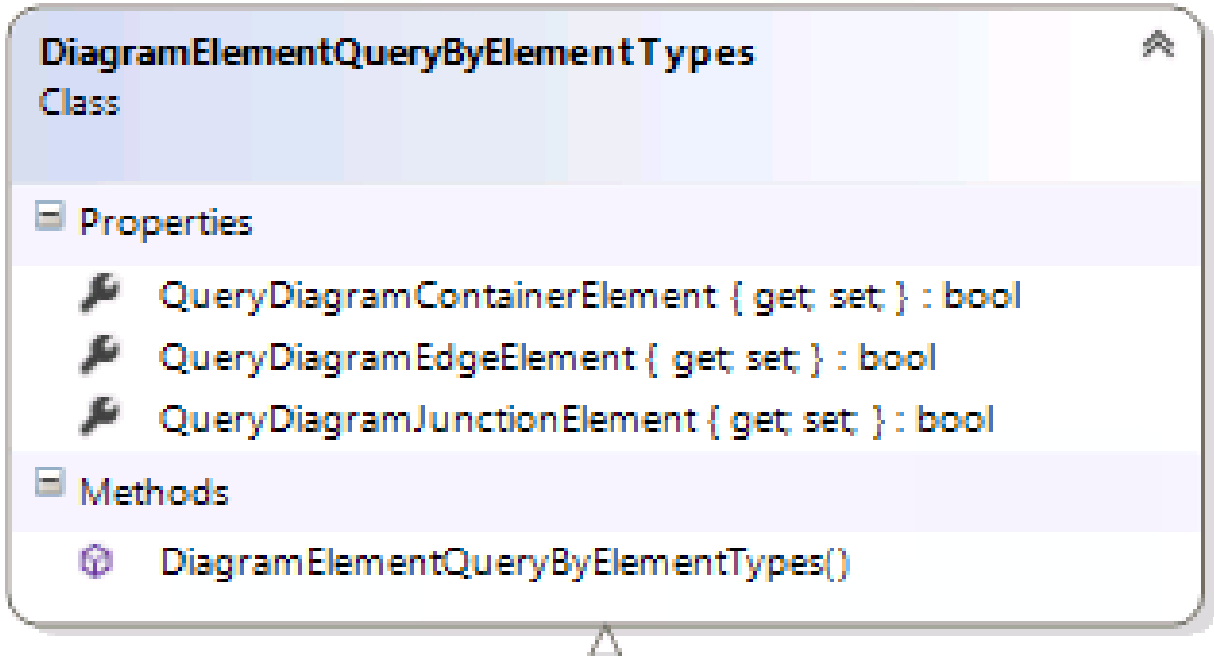
`ToID`

The diagram element IDs for the junctions connected to a `DiagramEdgeElement`

NetworkDiagram — Retrieving Diagram Elements by Type

`QueryDiagramElements(DiagramElementQueryByElementTypes query): DiagramElementQueryResult`

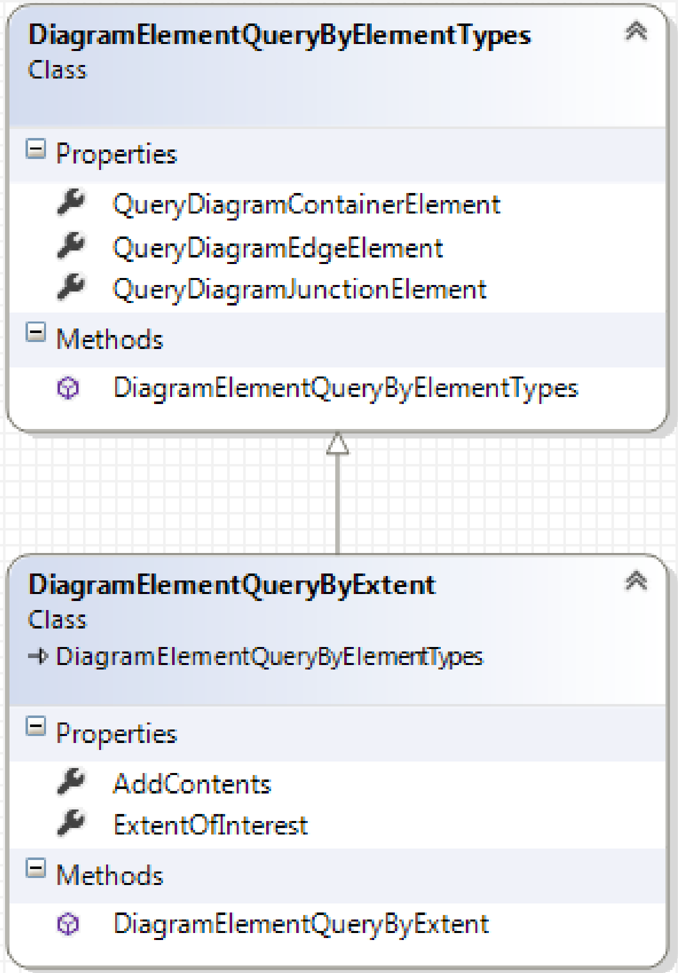
- Queries the set of edge, junction and/or container elements in the diagram, as specified by the `DiagramElementQueryByElementTypes` object



NetworkDiagram — Retrieving Diagram Elements by Type and Extent

`QueryDiagramElements(DiagramElementQueryByExtent query): DiagramElementQueryResult`

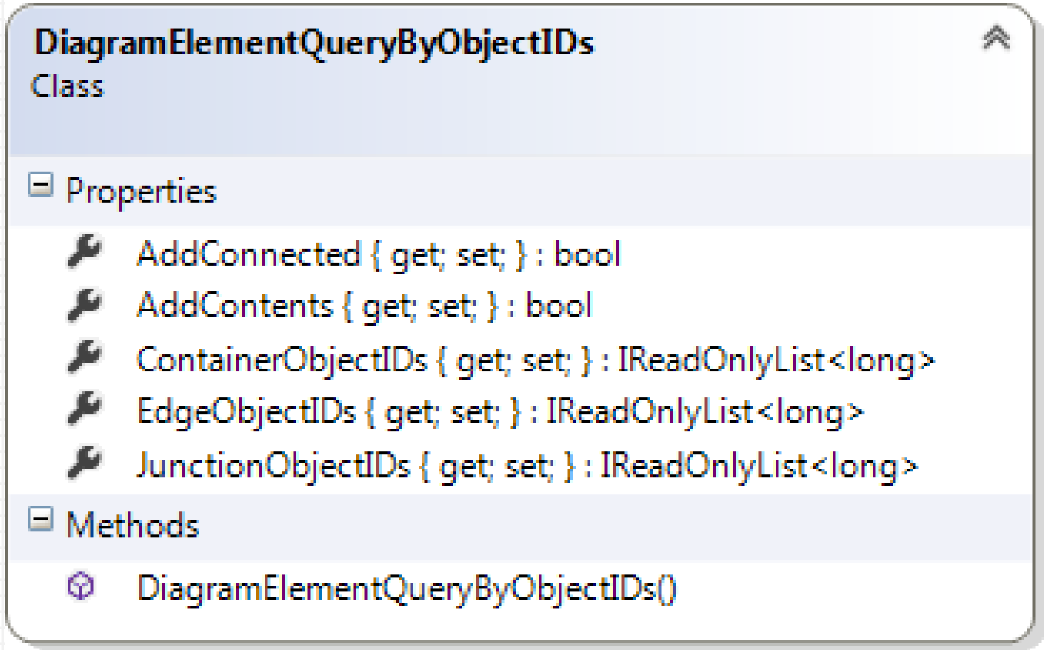
- Queries the set of edge, junction and/or container elements in the diagram, as specified by the `DiagramElementQueryByExtent` object



NetworkDiagram — Retrieving Diagram Elements by ObjectIDs

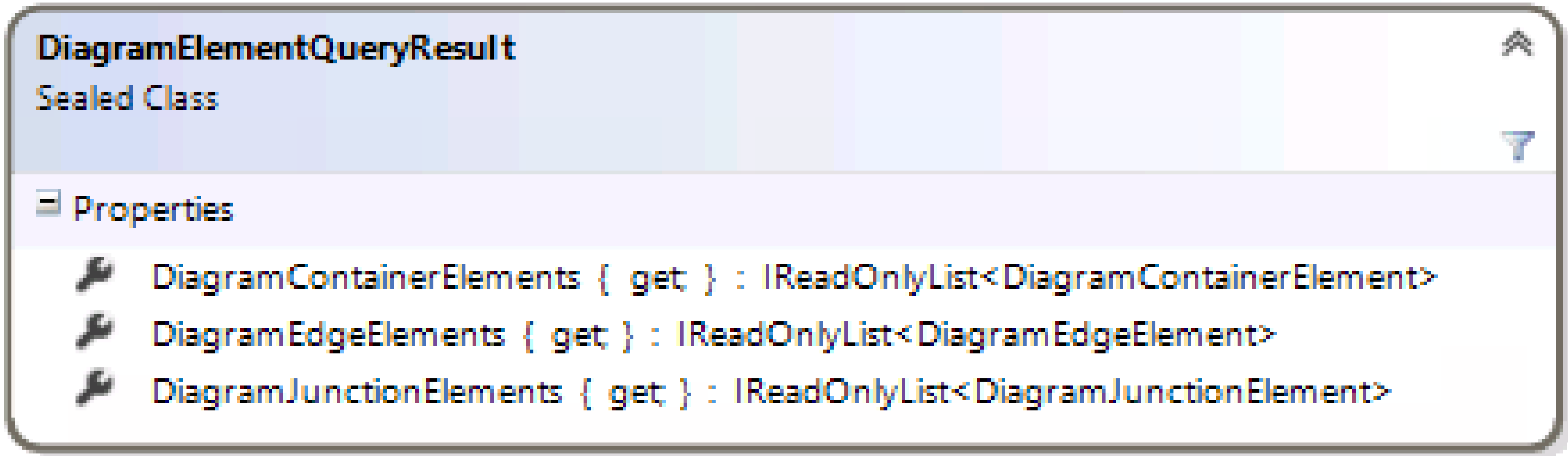
`QueryDiagramElements(DiagramElementQueryByObjectIDs query): DiagramElementQueryResult`

- Queries the set of edge, junction and/or container elements in the diagram, as specified by the `DiagramElementQueryByObjectIDs` object



The DiagramElementQueryResult Class

- The `DiagramElementQueryResult` class contains lists of the elements inside the diagram



Code Snippet: Retrieving Diagram Elements

```
// Create a DiagramElementQueryByExtent to retrieve diagram element junctions which extent intersects the active map extent.
DiagramElementQueryByExtent query = new DiagramElementQueryByExtent();
query.ExtentOfInterest = MapView.Active.Extent;
query.AddContents = false;
query.QueryDiagramJunctionElement = true;
query.QueryDiagramEdgeElement = false;
query.QueryDiagramContainerElement = false;

// Use this DiagramElementQueryByExtent query as an argument of the QueryDiagramElements method
DiagramElementQueryResult result = diagram.QueryDiagramElements(query);
```

3. Network Diagram

Diagram Aggregations

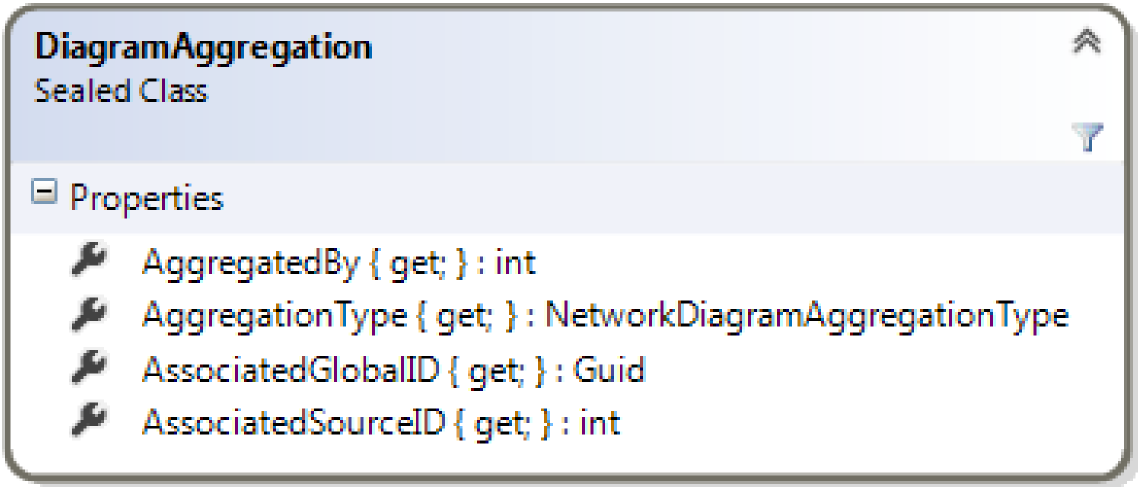
Diagram Aggregations

- Diagram aggregations are created when a utility network feature is reduced or collapsed in a network diagram
  - The utility network feature itself does therefore not correspond to a single diagram element

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

  - Returns details on all the diagram elements that are aggregated in the diagram; that is, reduced or collapsed

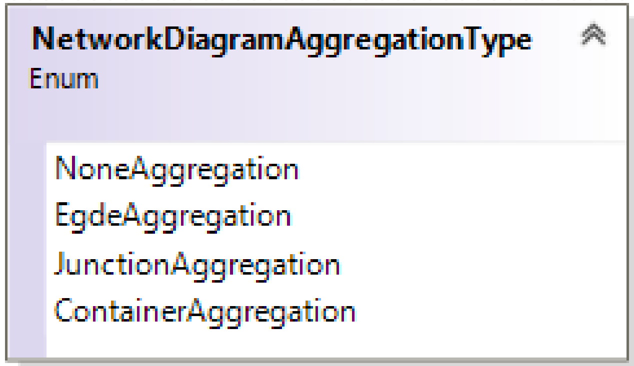
The DiagramAggregation Class



AggregatedBy : int

- The diagram element ID for the diagram element under which it has been reduced or collapsed

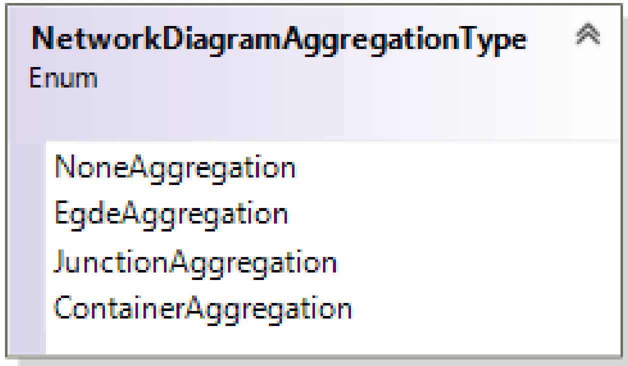
AggregationType : NetworkDiagramAggregationType



AssociatedGlobalID : GUID

AssociatedSourceID : int

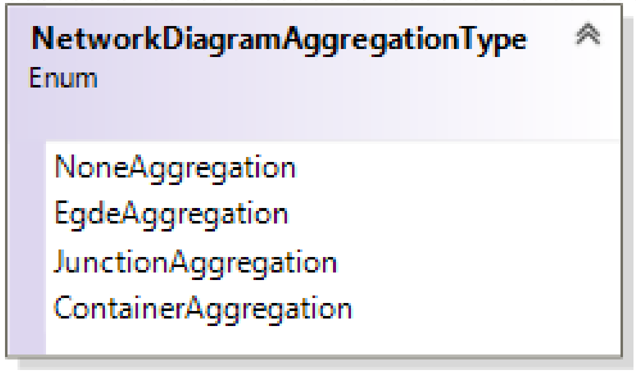
- These two properties retrieve the utility network feature/network source for the feature that was not included in the diagram



Aggregation Types — 1

- `JunctionAggregation`—for utility network features aggregated under a diagram junction element. This diagram junction may correspond to:
  - A 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 element. This diagram edge may correspond to:
  - A 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

Aggregation Types — 2



- `ContainerAggregation`—for utility network features aggregated under a diagram polygon. This diagram polygon may correspond to:
  - A 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

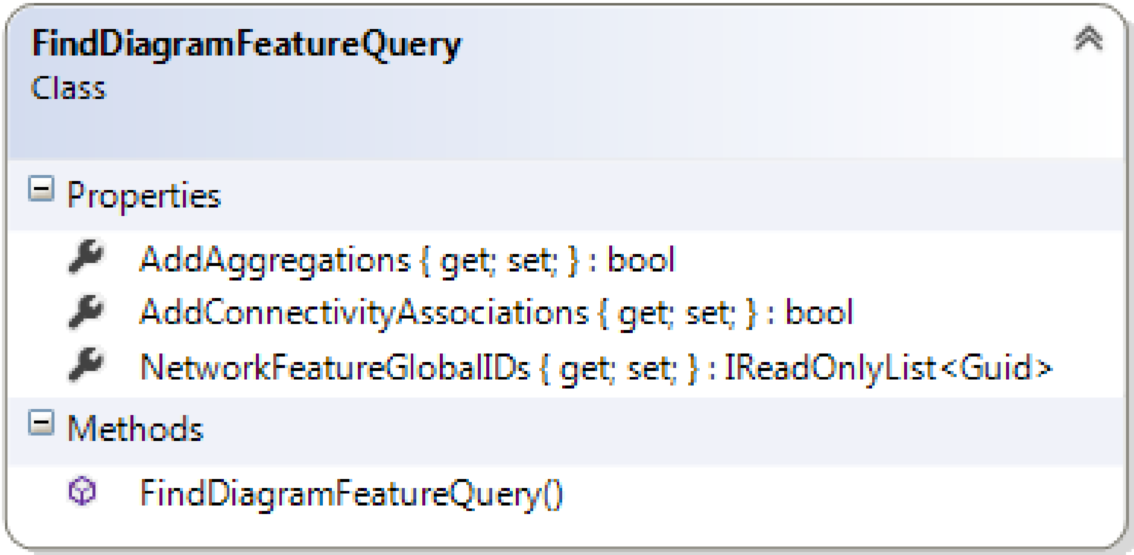
- `NoneAggregation`—for utility network features aggregated in the diagram without a corresponding diagram element
  - For example, unconnected system junctions are systematically reduced to nothing when running a Reduce Junction rule that processes unconnected junctions
  - In the same way, depending on the Reduce Junction rule settings, certain disconnected portions of the utility network may be reduced to nothing in the resulting diagram

3. Network Diagrams

Querying Network Features and Diagram Elements

NetworkDiagram — Retrieve Particular Diagram Features

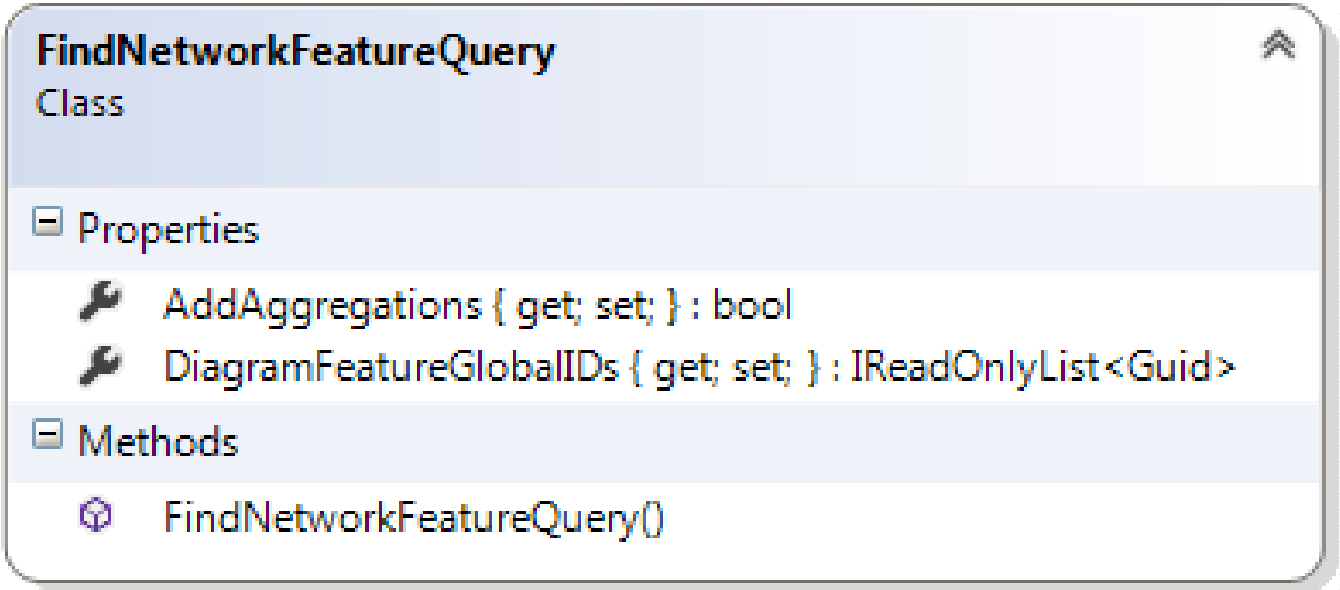
- FindDiagramFeatureGlobalIDs(FindDiagramFeatureQuery query): IReadOnlyList<Guid>
  - You must create a FindDiagramFeatureQuery object and use this query object as an argument of the FindDiagramFeatureGlobalIDs routine to retrieve the list of GUIDs corresponding to all features in the diagram, that:
    - Are associated with a set of utility network feature GUIDs
    - Represent connectivity associations
    - Are clearly represented in the diagram or aggregated; that is, collapsed or reduced, or not



NetworkDiagram — Retrieve Related Utility Network Features

FindNetworkFeatureGlobalIDs(FindNetworkFeatureQuery query):
    IReadOnlyList<Guid>

- You must create a FindNetworkFeatureQuery object and use this query object as an argument of the FindNetworkFeatureGlobalIDs routine to retrieve the list of GUIDs corresponding to all utility network features that:
  - Are associated with a set of diagram feature GUIDs
  - Whether or not those diagram features are aggregated; that is, collapsed or reduced, or not



3. Network Diagrams

Editing

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 `Subnetwork.Update()`

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 `Subnetwork.Update()`

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 `Subnetwork.Update()`

NetworkDiagram — Store and Delete

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

```
Delete() : void
```

- The `Delete` routine cannot process subnetwork system diagrams

Network Diagram Transaction Semantics

- Network diagram editing routines must be included within an editing transaction
  - E.g., `Geodatabase.ApplyEdits()`
- All existing edits should be saved prior to this transaction
- The transaction should **only** contain a single network diagram editing routine; no other edits should be included
- These rules apply to the following routines:
  - `Update()`
  - `Append()`
  - `Overwrite()`
  - `Store()`
  - `Delete()`

3. Network Diagrams

Diagram Flags

NetworkDiagram — Managing Flags

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

- Retrieves all diagram flags of the specified type

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

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

- Flags or un-flags a particular diagram element

`ClearFlags(NetworkDiagramFlagType flagType): void`

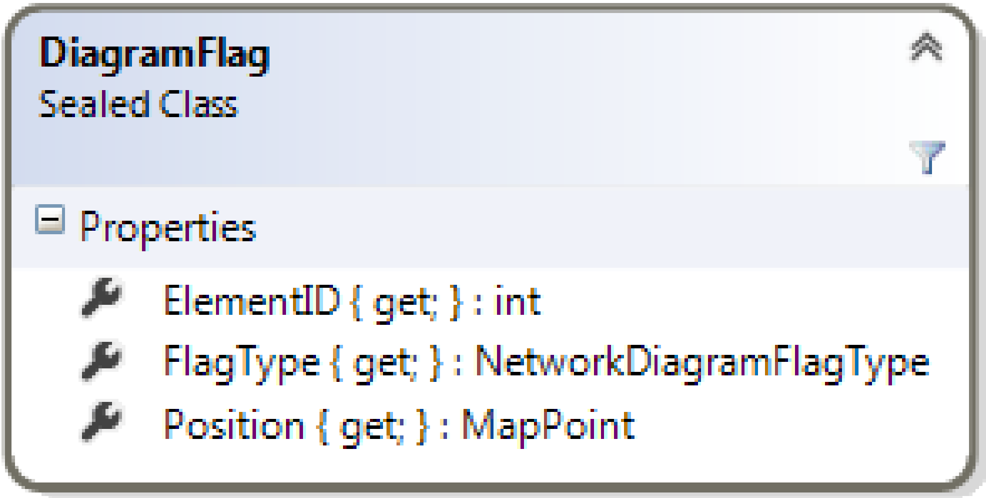
- Clears all diagram flags of the specified type

- The `AddFlag()`, `RemoveFlag()` and `ClearFlags()` routines are transactional operations; that is, their calls must be wrapped in a transaction
  - Called from `Geodatabase.ApplyEdits()`

Network Diagram Flag Transaction Semantics

- Network diagram editing routines must be included within an editing transaction
  - E.g., `Geodatabase.ApplyEdits()`
- All existing edits should be saved prior to this transaction
- The transaction should **only** contain a single network diagram editing routine; no other edits should be included
- These rules apply to the following routines:
  - `AddFlag()`
  - `RemoveFlag()`
  - `ClearFlags()`

The DiagramFlag Class



`ElementID : int`

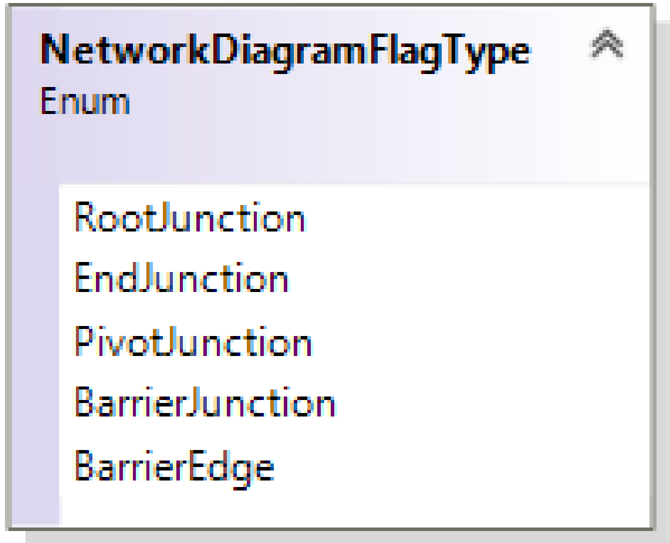
- The diagram element ID for the flag

`FlagType : NetworkDiagramFlagType`

- Type of flag
- Root, end, pivot and barrier flags are specific junctions recognized by certain algorithm layouts that process the resulting diagram layout accordingly

`Position: MapPoint`

- Location on the diagram

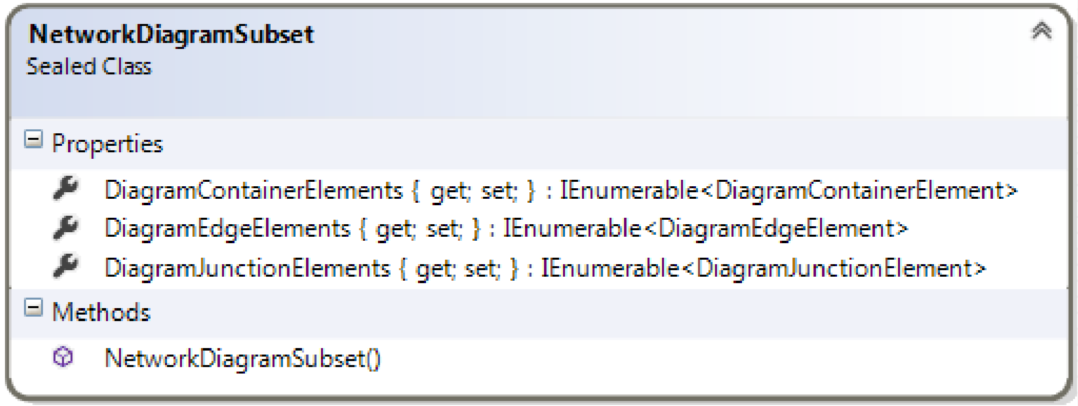


3. Network Diagrams

Diagram Layouts

The NetworkDiagramSubset Class

- This class specifies a subset of diagram elements:
  - DiagramContainerElements: IEnumerable<DiagramContainerElement>
  - DiagramEdgeElements: IEnumerable<DiagramEdgeElement>
  - DiagramJunctionElements: IEnumerable<DiagramJunctionElement>
- You must create a NetworkDiagramSubset object and use it as an argument to NetworkDiagram.SaveLayout() to save any custom layout you may have coded to edit the elements of a network diagram



Code Snippet: Using the DiagramElementQueryResult and NetworkDiagramSubset Classes

```
// Retrieving a diagram
diagramTest = diagramDataset.GetNetworkDiagram(diagramName);

// Creating a DiagramElementQueryByElementTypes query object to get the diagram elements we want to work with
DiagramElementQueryByElementTypes query = new DiagramElementQueryByElementTypes();
query.QueryDiagramJunctionElement = true;
query.QueryDiagramEdgeElement = true;
query.QueryDiagramContainerElement = true;

// Retrieving those diagram elements
DiagramElementQueryResult elements = diagramTest.QueryDiagramElements(query);

// Creating a NetworkDiagramSubset object to edit this set of diagram elements
NetworkDiagramSubset subset = new NetworkDiagramSubset();
subset.DiagramJunctionElements = elements.DiagramJunctionElements;
subset.DiagramEdgeElements = elements.DiagramEdgeElements;
subset.DiagramContainerElements = elements.DiagramContainerElements;

// Editing the diagram elements
TranslateDiagramElements(dX, dY, subset);

// Saving the lay out of the diagram elements
geodatabase.ApplyEdits(() => diagramTest.SaveLayout(subset, true));
```

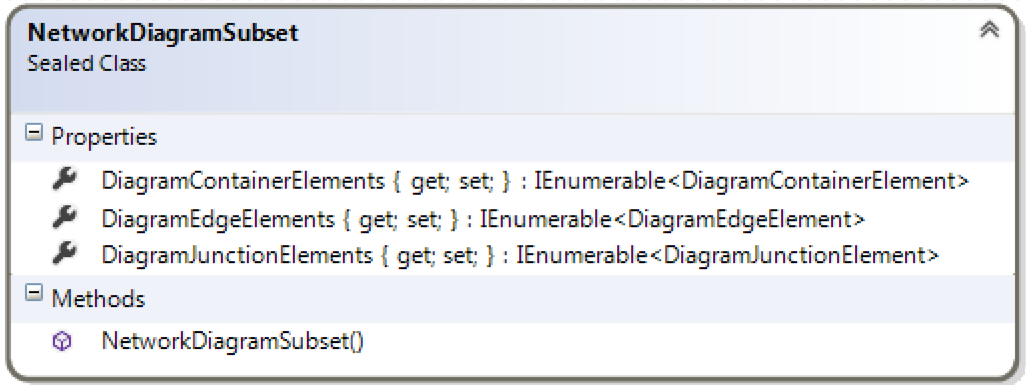
NetworkDiagram — Lay out the diagram content

`SaveLayout(NetworkDiagramSubset subset, bool keepVertices) : void`

- Saves any custom layout you may have coded to edit a subset of diagram junctions, edges and containers geometry field (Shape)
- Create a NetworkDiagramSubset object and use this object as an argument of the SaveLayout routine

`ApplyTemplateLayouts() : void`

- Applies the list of the layout algorithms currently configured on the template the input network diagram layer is based on

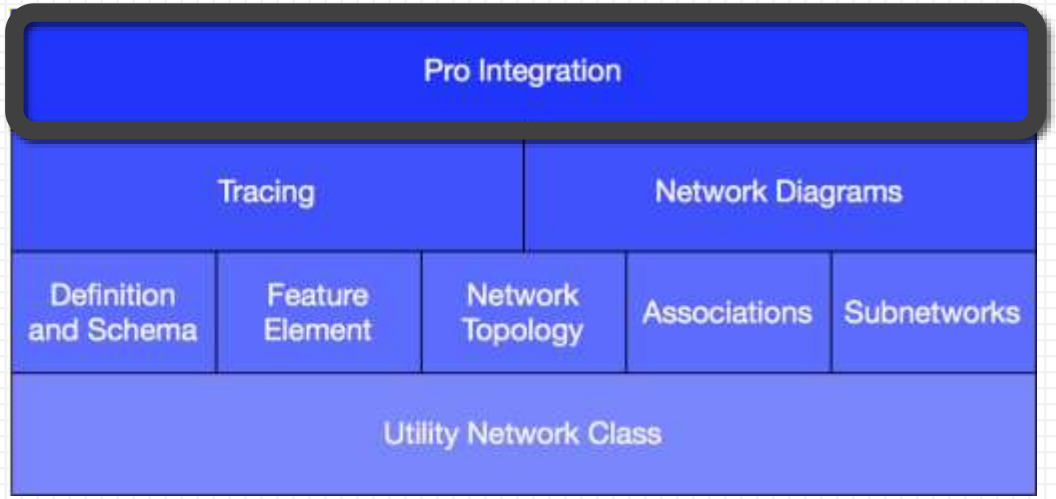


Code Snippet: Saving a Layout

```
// Create a NetworkDiagramSubset object with all the diagram features that have been edited
NetworkDiagramSubset nds = new NetworkDiagramSubset();
nds.DiagramContainerElements = containersToSave;
nds.DiagramEdgeElements = edgesToSave;
nds.DiagramJunctionElements = junctionsToSave;
// SaveLayout on the edited edited diagram features
m_geodatabase.ApplyEdits(() => m_diagram.SaveLayout(nds, m_KeepVertices));
```

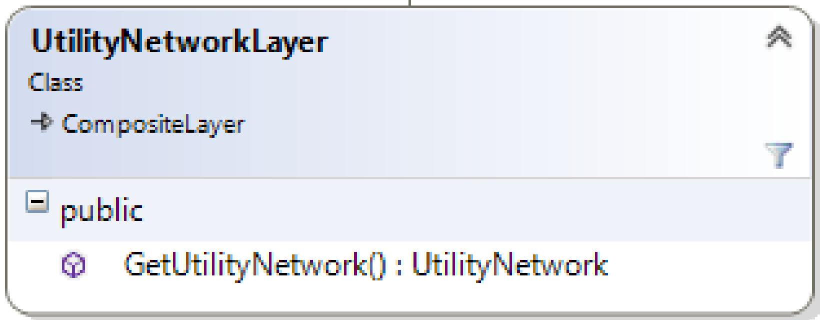
Network Diagram Transaction Semantics

- Network diagram editing routines must be included within an editing transaction
  - E.g., `Geodatabase.ApplyEdits()`
- All existing edits should be saved prior to this transaction
- The transaction should **only** contain a single network diagram editing routine; no other edits should be included
- These rules apply to the following routines:
  - `SaveLayout()`
  - `ApplyTemplateLayouts()`



Pro Integration

The UtilityNetworkLayer Class



- 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