

# GENERATIVE COMPONENTS – API



## Table of Contents

1. Getting Started .....	6
1.1. Creating a GC add-in project .....	7
1.2. The two node-type architectures .....	20
1.3. Script Almighty .....	21
2. Nodes (General) .....	23
2.1. NodeUpdateResult .....	24
2.2. Customizing the pop-up property editor .....	31
2.2.1. File browser .....	34
2.2.2. Other custom editor behaviors (overview) .....	41
2.2.3. Custom lists.....	43
2.2.3.1. Letting the user override the provided choices.....	48
2.2.3.2. Controlling what's displayed for each item .....	49
2.2.3.3. Special considerations for string data.....	54
2.2.3.4. Grouping.....	58
2.2.4. Custom expression editors, part 1 .....	63
2.2.4.1. A new property on the BioChamber.....	64
2.2.4.2. Getting started with a custom expression editor.....	67
2.2.4.3. Making our custom expression editor useful.....	72
2.2.4.4. Wiring up our custom editor to our node .....	74
2.2.5. Custom expression editors, part 2.....	78
2.2.5.1. Revising the BioChamber property .....	78
2.2.5.2. The custom dialog .....	81
2.2.5.3. Modifying our custom editor.....	83

2.3.	Node port pinning.....	90
3.	Element-Based Nodes.....	93
3.1.	CopyTransform / MirrorCopy techniques.....	94
3.2.	Getting/setting node properties .....	98
3.2.1.	Way 1: Call GetPropertyValue / SetPropertyValue.....	99
3.2.2.	Way 2: C# class properties .....	101
3.2.3.	Way 3: Performance improvement .....	102
3.3.	Output-only properties .....	107
3.4.	Output-only properties that are always valid .....	109
3.4.1.	Way 1: Add that property to every technique.....	110
3.4.2.	Way 2: Property calculators.....	112
3.5.	Constituent nodes .....	116
3.6.	Shadow classes .....	120
3.7.	Technique attributes .....	121
4.	Utility Nodes.....	124
4.1.	Explicit framework.....	125
4.2.	Where are the technique attributes? .....	126
4.3.	Custom UI .....	132
5.	General Operations .....	138
5.1.	Creating and modifying nodes programmatically .....	139
5.1.1.	GC transactions.....	140
5.1.2.	Creating and performing a transaction.....	144
5.2.	Create new nodes that are attached to existing nodes .....	149
6.	Using GC as a Service .....	154
6.1.	Introduction .....	155

6.2.	How to start GC in service mode .....	156
6.2.1.	Service mode parameters .....	159
6.3.	Walkthrough: Create a small client application .....	160
6.3.1.	Getting started.....	161
6.3.2.	Adding the GC service reference .....	165
6.3.3.	Our first communications with the GC service.....	170
6.3.4.	Preparing a transaction file for use with the GC service .....	174
6.3.5.	Loading and controlling the transaction file (part 1) .....	179
6.3.6.	Loading and controlling the transaction file (part 2) .....	181
6.3.7.	Manipulating the GC model.....	185
6.4.	The GC service interface (reference) .....	191
7.	Code Samples.....	196
7.1.	Calculated properties.....	197
7.1.1.	SimpleLineNodeWithCalculatedLength.cs .....	197
7.2.	Comparison of element-based node and utility node .....	200
7.2.1.	BioChamberElementBasedNode.cs .....	200
7.2.2.	BioChamberUtilityNode.cs.....	203
7.3.	Constituent nodes .....	209
7.3.1.	SimpleLineNodeWithConstituentProperties.cs .....	209
7.4.	Custom expression editors, part 1 .....	212
7.4.1.	BioChamberNode.cs .....	212
7.4.2.	BioChamberDailySnacksEditor.xaml.....	215
7.4.3.	BioChamberDailySnacksEditor.xaml.cs.....	216
7.5.	Custom expression editors, part 2 .....	219
7.5.1.	BioChamberNode.cs .....	219

7.5.2.	BioChamberDailySnacksEditor.xaml .....	222
7.5.3.	ItemEntryDialog.xaml .....	227
7.5.4.	ItemEntryDialog.xaml.cs .....	229
7.6.	GC Service Client (SimpleGCServiceClient) .....	231
7.6.1.	Program.cs .....	231
7.7.	SampleAddIn (distributed with GC) .....	233
7.7.1.	SimpleLineNode.cs .....	235
7.7.2.	CalculatorNode.cs .....	240
7.7.3.	CalculatorNodeViewContent.xaml .....	247
7.7.4.	CalculatorNodeViewContent.xaml.cs .....	249
7.7.5.	ScriptFunctions.cs .....	258
7.7.6.	Initializer.cs .....	261

# 1. Getting Started

# 1.1. Creating a GC add-in project

GC's API lets you create your own GC node types that can do anything any of GC's built-in node types can do, plus many other things of your own invention. (Maybe not quite limited only by your imagination, but that's a general idea.) Furthermore, you can create new global script functions for the user to use in their own script, that works directly with the underlying nuts and bolts of GC and/or MicroStation.

You will use Visual Studio and the C# programming language to create a .NET assembly (DLL file) that can be loaded into GC as an "add-in". Here is the process:

1. Install Visual Studio, if you don't already have it. The free Community Edition is sufficient. The latest version (2022 at this writing) can be downloaded from here: <https://visualstudio.microsoft.com/vs/community/>

When you install Visual Studio, choose the workload ".NET desktop development".

2. Learn Visual Studio and C# programming (!).

◆ Furthermore, if you want to write any utility-type nodes, which each have a custom appearance and behavior in GC's graph, learn WPF (Windows Presentation Foundation, a Microsoft technology for developing user interfaces).

It's well beyond the scope of this GC API documentation to teach you these things or even recommend specific sources of information. However, Visual Studio, C#, and WPF are extremely popular, so you should have no trouble finding many tutorials on the internet and in books.

Your journey should include becoming familiar with Visual Studio's diagnostic facilities: Setting breakpoints, single stepping through your code, the Locals window, etc.

Take the time to create, and experiment with, small apps that have nothing to do with GC.

3. Get a Visual Studio project template for a GC add-in. There are two options (A and B), listed below. I recommend trying both options since they each have their own advantage.
  - Option A: Install the sample add-in project that's provided with GC. You can then use that as a starting point for your own node types and/or script functions.

The advantage of this option is that you'll start off having two fully-functional (and fully-commented) node classes, [one element-based and one utility](#), to serve as a reference for writing your own node classes. There is also a fully defined script function, showing how that's done.

The sample add-in project is provided in the form of a zip file named "GCSampleAddIn.zip". In a typical installation, it can be found here:

<C:\Program Files\Bentley\OpenBuildings CONNECT Edition\OpenBuildingsDesigner\GenerativeComponents\SampleSolution\>

To "install" the sample add-in project, extract the contents of that zip file to a local folder. For example, create a new folder named "GCSampleAddIn" under your standard Documents folder.

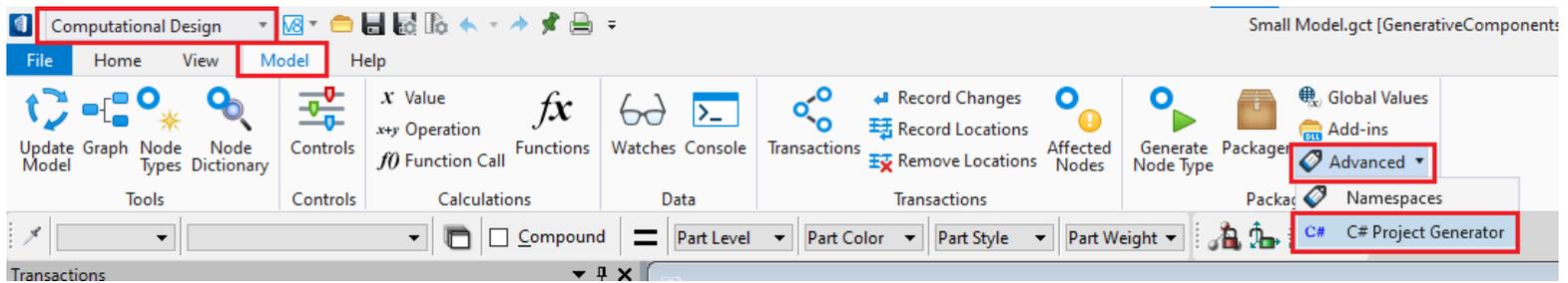
Within that extracted folder, read and follow the instructions in the file, "Please Read Me First.txt".

◆ That file describes a couple of environment variables you'll to define on your computer.

- Option B: Create a new add-in project using GC's built-in C# project generator.

The advantage of this option is that the project will be built to your custom specifications, including the names of your new node class(es), the names and parameters of your technique methods, etc. (Don't worry if you don't get it right the first time; after GC generates the project, you'll use Visual Studio to change it however you want.) Also, unlike option A, you won't need to define any new environment variables.

From within GenerativeComponents, launch the command, "C# Project Generator":



Then follow the prompts:

**C# Project Generator**

### Identify the name and location of your new C# project

This tool is intended for users who are proficient in Microsoft's Visual Studio development environment, including the .NET Framework, the C# programming language, and (optionally) Microsoft's WPF framework for writing custom user interfaces.

This tool will generate a new Visual Studio C# project based on your specifications. It will generate skeletal classes for new node types and/or functions that can be called from GCScript.

After that project has been generated, you can close OpenBuildings Designer, open that project (.csproj file) in Visual Studio, and "fill in the blanks" to implement the specific functionality you desire.

It's your responsibility to obtain, install, and learn Microsoft's Visual Studio. The free Community Edition is sufficient; search the internet for "visual studio community". The internet also provides lots of resources to help you learn Visual Studio, the .NET Framework, the C# language, and (optionally) WPF.

Project name

Location (parent folder)  
 ...

- Close GC — that is, close its host application, such as Open Buildings Designer.
1. Launch Visual Studio 2022 and open the project's project file (.csproj).
  2. Regardless of which option (A or B) you used to get the project, the project should be immediately buildable and test runnable.

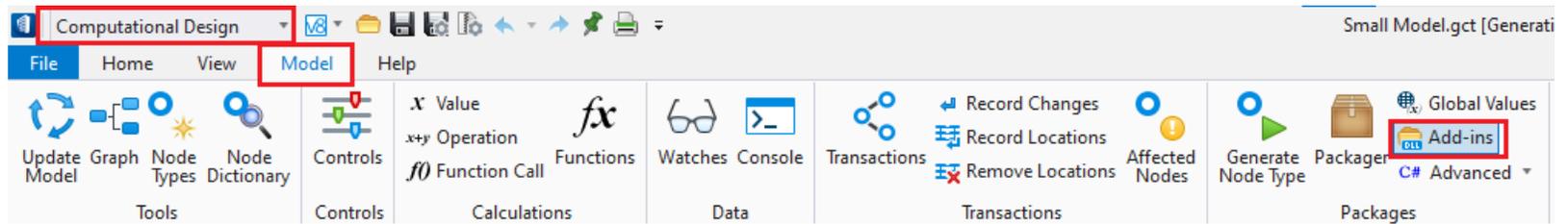
(From within Visual Studi:) Build the project, then start it ("Start Debugging"). After a short pause, it should launch GC's host application (e.g., Open Buildings Designer).

3. **Important:** When GC's host application starts, you should choose to open a design file or transaction file that's expendable — that is, it doesn't matter if it gets ruined.

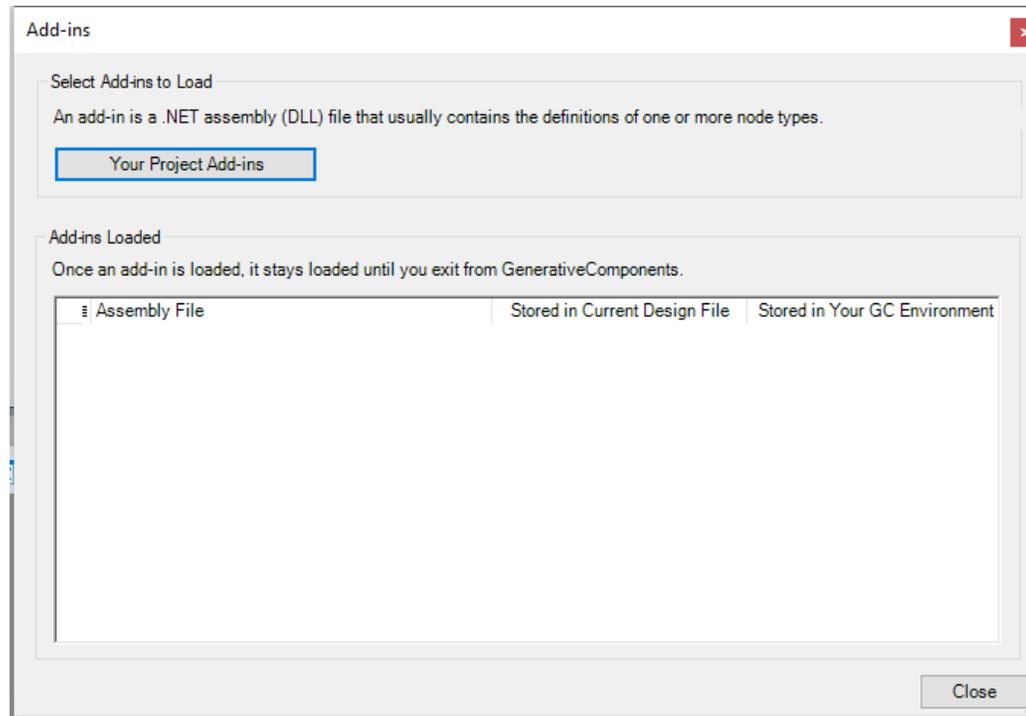
When you're test-running a Visual Studio project -- particularly after you start adding your own code -- it's possible it will behave so badly it will corrupt the active design or transaction file.

4. This step is something you should need to do only the first time you start your project.

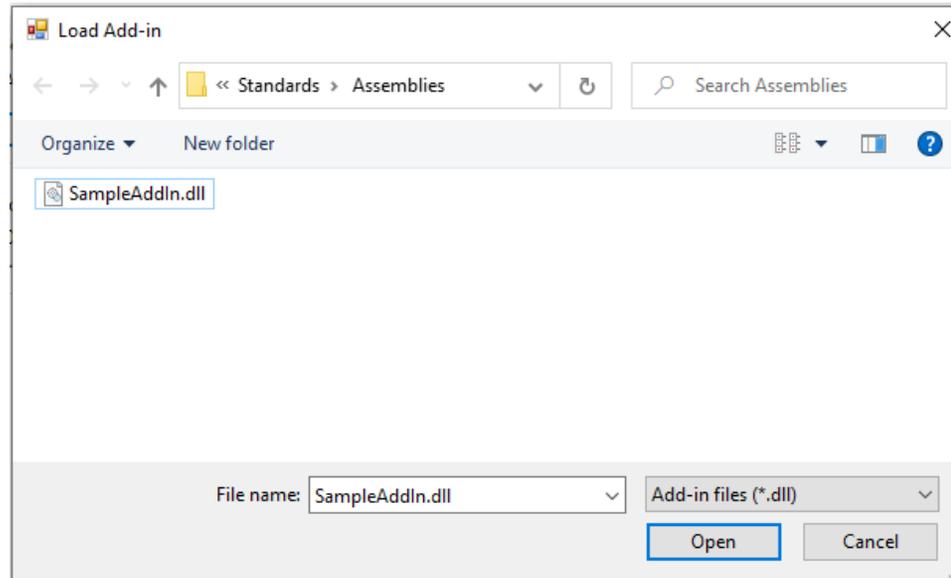
From within GC, launch the command, "Add-ins".



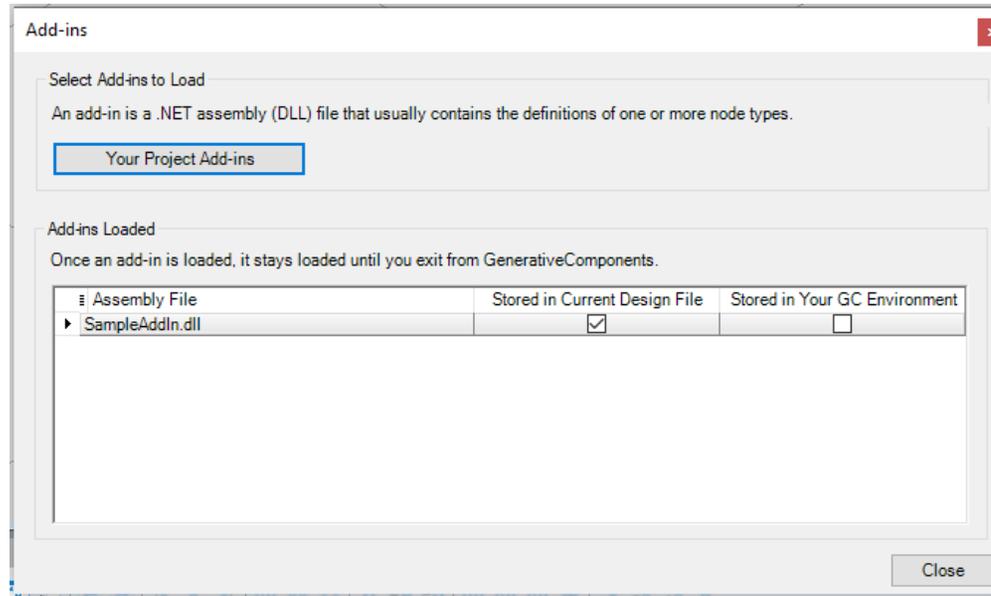
Then follow the prompts:



When you click the button, "Your Project Add-ins", you should see the assembly file (DLL file) that Visual Studio just built from your project. Open it.



And GC will load it:



You can see that, by default, your add-in is stored in your current design file. As long as you always start your GC (OBD) session by opening that same design file, your add-in will be available within GC.

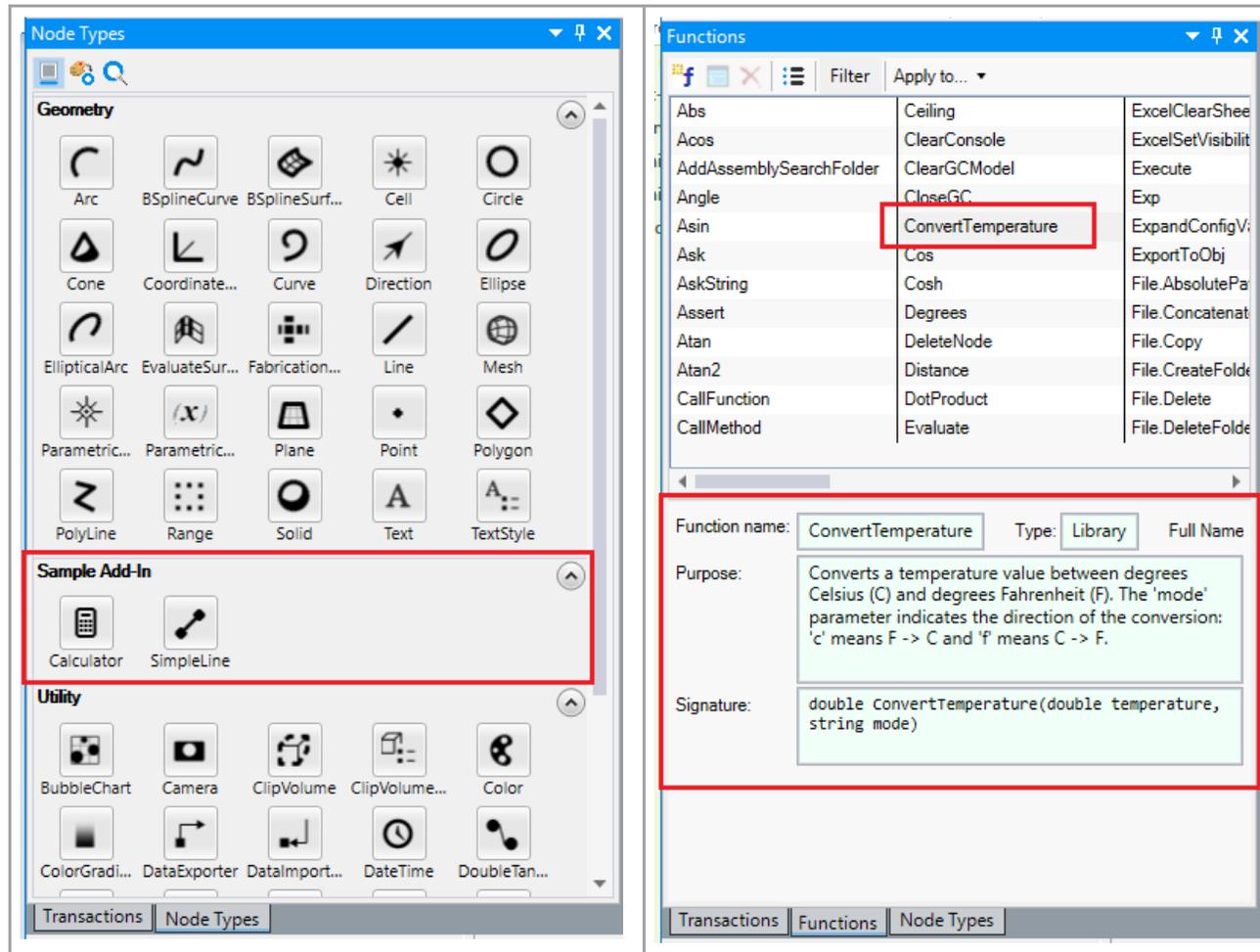
If you start your GC (OBD) session by opening a different design file, or a transaction (.gct) file, you will need to manually re-load your add-in by following step 8 from the beginning.

Alternatively, you can check the box "Stored in Your GC Environment", which means your add-in will always be loaded into GC regardless of which design file or transaction file was opened initially.



(If you later decide you want to stop loading your add-in, simply launch the "Add-ins" command again, and uncheck that check box.)

5. Assuming all has gone well so far, you should now see that your new node type(s) and/or script function(s) are integrated into GC, just like any other node types and script functions.



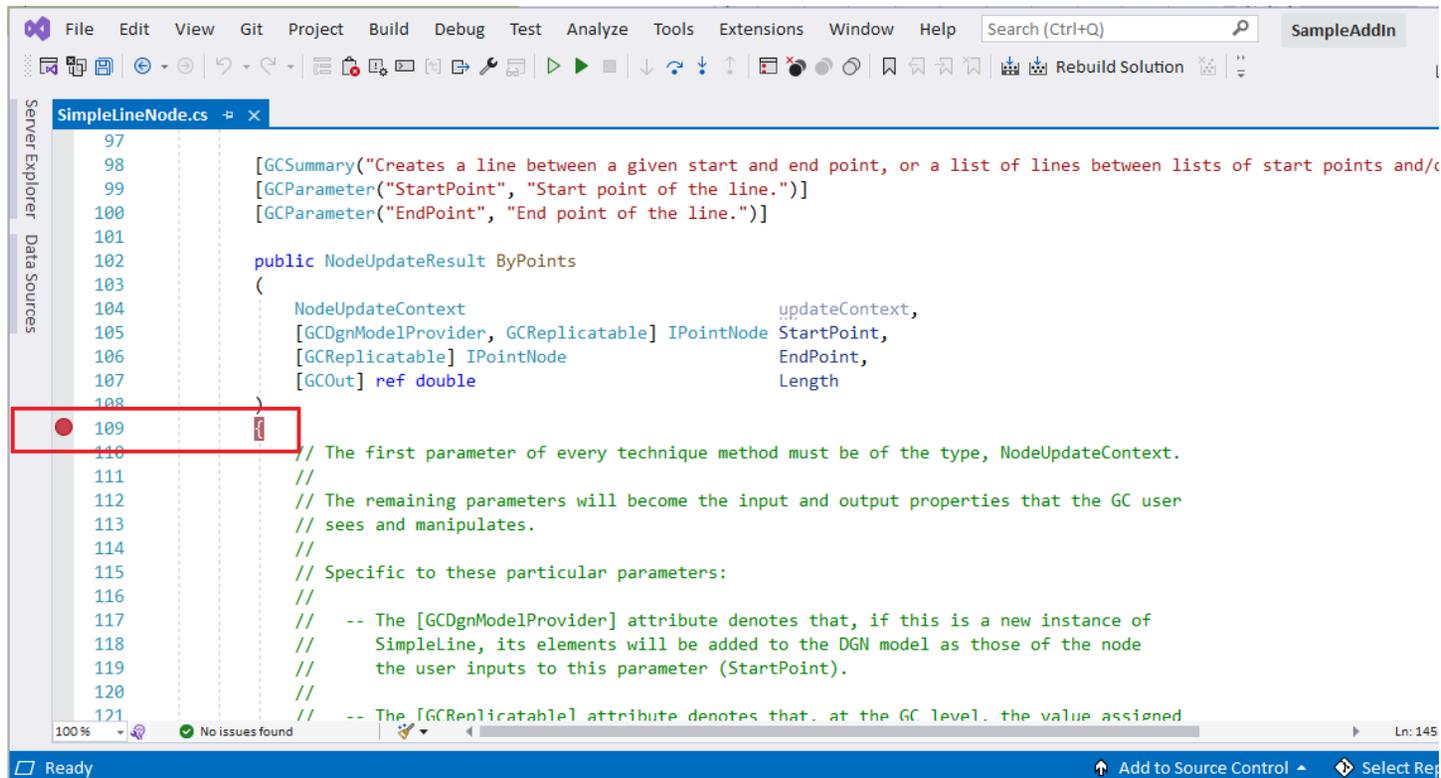
(The nodes and functions shown above are defined in GC's sample add-in, following options A in step 3.)

You can now exercise your new node(s) and/or function(s) to determine whether they're working as expected.

6. When you're finished exercising, close GC — that is, close its host application — to return to your Visual Studio project.
7. As you introduce your own code into your project, you'll discover that most of the time, it doesn't work, and you'll need to figure out why. This is completely normal: Even professional programmers often spend more time testing, diagnosing, and fixing their code than they spend originally writing it!

Using the GC sample add-in as an example, let's suppose that, when you try to add a SimpleLine node to your model, something doesn't work right: The node produces unexpected results, no results, or it even triggers an error condition that "crashes" GC (makes it stop working completely)!

This is where your expert code diagnostic skills come into play. Within your Visual Studio project, you can set a breakpoint at the top of SimpleLine's one-and-only technique method, ByStartPoints ...



The screenshot shows the Visual Studio IDE with the file SimpleLineNode.cs open. A red circle and vertical bar indicate a breakpoint is set on line 109. The code is as follows:

```
97  
98 [GCSummary("Creates a line between a given start and end point, or a list of lines between lists of start points and/  
99 [GCParameter("StartPoint", "Start point of the line.")]  
100 [GCParameter("EndPoint", "End point of the line.")]  
101  
102 public NodeUpdateResult ByPoints  
103 (  
104     NodeUpdateContext updateContext,  
105     [GCDgnModelProvider, GCReplicable] IPointNode StartPoint,  
106     [GCReplicable] IPointNode EndPoint,  
107     [GCOut] ref double Length  
108 )  
109  
110 // The first parameter of every technique method must be of the type, NodeUpdateContext.  
111 //  
112 // The remaining parameters will become the input and output properties that the GC user  
113 // sees and manipulates.  
114 //  
115 // Specific to these particular parameters:  
116 //  
117 // -- The [GCDgnModelProvider] attribute denotes that, if this is a new instance of  
118 // SimpleLine, its elements will be added to the DGN model as those of the node  
119 // the user inputs to this parameter (StartPoint).  
120 //  
121 // -- The [GCReplicable] attribute denotes that, at the GC level, the value assigned
```

... then re-start the project, which will re-launch GC's host application. Then, do whatever you did before to test your node.

When your breakpoint is hit, Visual Studio will jump to the foreground, and you can single-step through your method, examining the values of variables, etc., until you discover what the problem is.

This is the iterative process of software development: Test the code, diagnose the code, and revise the code, over and over again, until, finally, it's working the way you want.

## 1.2. The two node-type architectures

GC supports two broad categories of nodes: element-based nodes and utility nodes. Their capabilities and implementations are different almost from the ground up.

- **Element-based nodes**

- Derived, either directly or indirectly, from the base class `ElementBasedNode`.
- Designed for creating and managing MicroStation elements.
- Replicable.
- Consistent UI: All element-based nodes have the same look and feel in GC's graph.
- An element-based-node class includes lots of GC attributes, which drive the generation of the corresponding [shadow class](#).

[Here](#) is an example of a C# class that defines an element-based node.

- **Utility nodes**

- Derived, either directly or indirectly, from the base class `UtilityNode`.
- Designed for operations that affect the GC model but have no graphical representation.
- Not replicable.
- Customizable UI: A utility node can — and almost always does — have a custom UI in GC's graph. That UI must be implemented in WPF (Windows Presentation Foundation), a Microsoft technology provided in Visual Studio.
- A utility-node class doesn't have a corresponding, generated "shadow" class. Rather, the utility-node class, itself, directly implements all of its own functionality. Therefore, there are no GC attributes; everything is spelled out explicitly in the code.

[Here](#) is an example of a C# class that defines a utility node.

# 1.3. Script Almighty

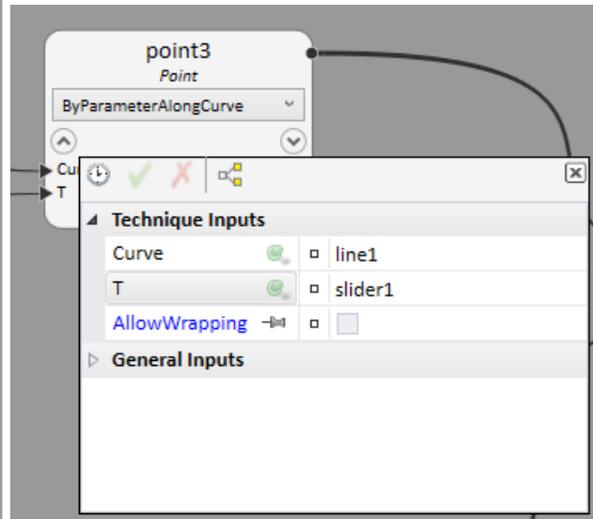
"Script" refers to any script code the user composes — either directly or indirectly, and either knowingly or unknowingly — while using GC. This includes not just script functions, but any input expression given to any node.

Even nodes that are connected purely by wires are working with the script since every wire carries the script.

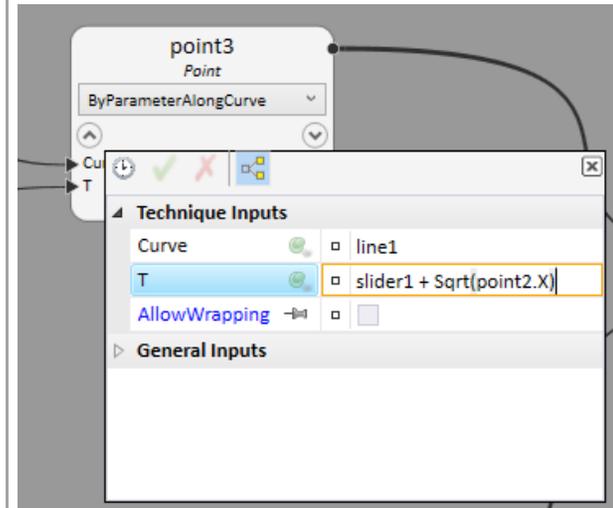
A user might look at this node and think, "There's no script involved here... just a couple of wires coming in".

A user might look at this node and think, "There's no script involved here... just a couple of wires coming in".

But when the user looks at what's being carried by those wires, they do see the script: 'line1' and 'slider1', respectively.



...As evidenced by the fact that, if the user wants to, they can manually extend that script that's being carried by the wire. (This doesn't change what's being carried by the wire -- that's still 'slider1' -- but it changes what ends up being given to the node.)



So, users are constantly working with the script, even if they don't perceive it. Script is the bloodstream of GC.

Therefore, when you create a new node type or almost anything else that's intended to work with GC, you're primarily defining how that node will behave when it's being controlled by script.

Manipulating nodes directly from C# isn't easy, since nodes aren't designed to be manipulated that way! If you're curious, [here](#)'s how to do it.

## 2.Nodes (General)

## 2.1. NodeUpdateResult

Every technique method, regardless of whether it's on an element-based node or a utility node, returns an instance of `NodeUpdateResult`. Then, GC examines that result to determine whether the node has been updated successfully, and, if not, to produce an appropriate error message for the user.

When you write your custom node classes, it's your job to return an appropriate instance of `NodeUpdateResult` from each of your technique methods, based on what you want to tell the user about the successfulness, or lack thereof, of that technique.

- **Using the predefined instances of NodeUpdateResult**

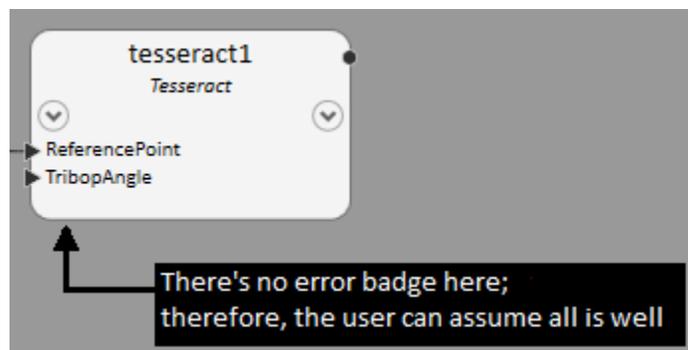
GC provides useful instances of `NodeUpdateResult` as static properties on the `NodeUpdateResult` class, itself.

- `NodeUpdateResult.Success`

This is probably the most commonly used instance of `NodeUpdateResult`:

```
return NodeUpdateResult.Success; // Indicate that this technique method has done whatever
                                // it's supposed to do, without any problems.
```

That result manifests to the user as the lack of an error badge (indicator):

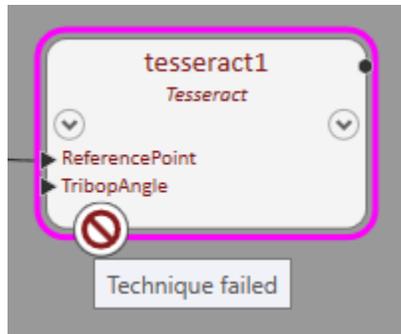


- `NodeUpdateResult.TechniqueGenericFailure`

This is a good "placeholder" result to indicate a failure when you're initially writing and testing your technique methods. However, you should plan on eventually replacing it with a more informative instance of `NodeUpdateResult`. Example:

```
if (TribopAngle <= 0)
    return NodeUpdateResult.TechniqueGenericFailure; // Quick-and-dirty indication that this
                                                    // technique method has encountered a problem,
                                                    // without providing any information regarding
                                                    // the nature of that problem.
:
```

That result manifests as a generic error message on the node:



- **Controlling the error message**

In addition to the predefined `NodeUpdateResult` instances described above, the class `NodeUpdateResult` provides some member classes that, themselves, are derived from `NodeUpdateResult`. The member classes give you a way to control the error message that manifests to the user.

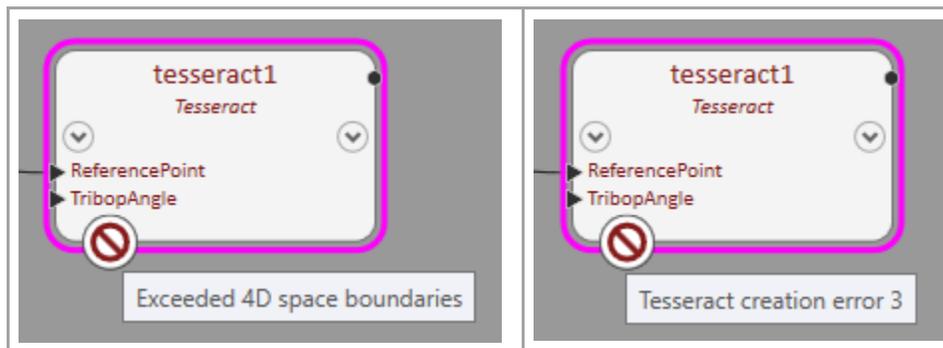
Since these member classes don't have predefined instances, each must be instantiated as you need it, using C#'s standard 'new' operator.

A commonly used class is `TechniqueFailureMessage`, which is a member class of `NodeUpdateResult`.

- `new NodeUpdateResult.TechniqueFailureMessage(Ls message)`

This lets you define the specific error message that will manifest to the user. For example:

```
int errorCode = SomeProcessThatCreatesATesseract(out tesseract);
if (errorCode != 0)
{
    string errorMessage;
    switch (errorCode)
    {
        case 2:    errorMessage = "Exceeded 4D space boundaries"; break;
        case 5:    errorMessage = "4D space has collapsed"; break;
        default:   errorMessage = $"Tesseract creation error {errorCode}"; break;
    }
    return new NodeUpdateResult.TechniqueFailureMessage(Ls.Literal(errorMessage));
}
:
```

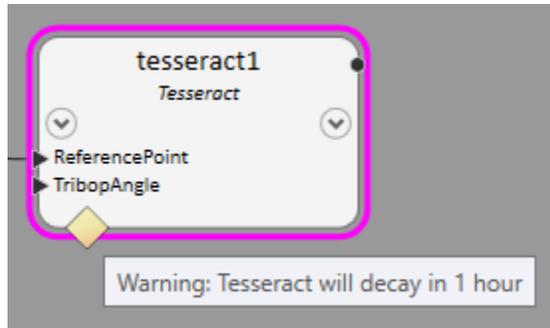


◆ That constructor, `NodeUpdateResult.TechniqueFailureMessage`, takes a parameter of type 'Ls', which means "localized string". Generally, you won't need to care about localization because your node types will support only one human language, such as English. Therefore, wherever an argument of type 'Ls' is expected, you can call the static method "Ls.Literal", which simply converts a normal C# string to an instance of Ls.

- `new NodeUpdateResult.SuccessButWithWarning(Ls message)`

GC treats this as a "success" result, exactly as if you return `NodeUpdateResult.Success`. However, additionally, you can specify a message that manifests to the user as a warning (only). For example:

```
if (tesract.IsUnstable)
    return new NodeUpdateResult.SuccessButWithWarning(Ls.Literal("Tesseract will decay in 1 hour"));
```



- **Handling errors in the user's input values**

GC, itself, performs some basic error checking of the user's input values; for example, it ensures those values are of the correct type expected by the technique method.

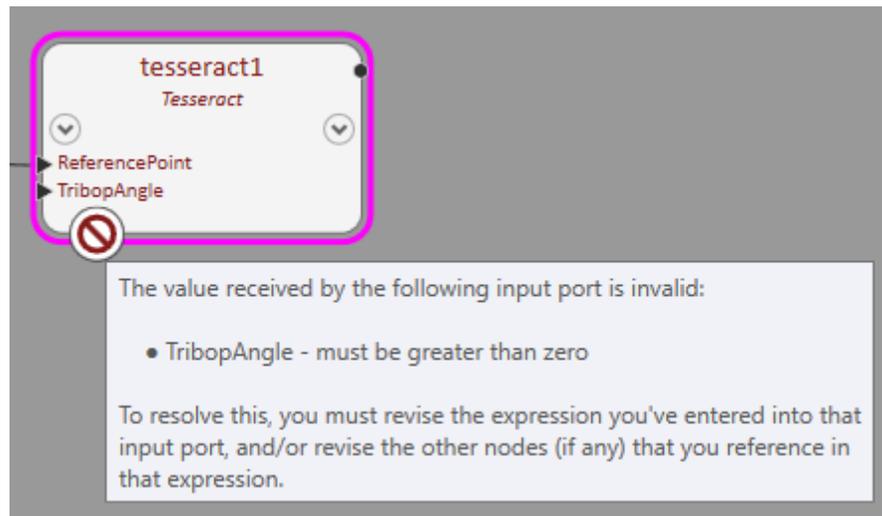
However, GC cannot know the particular conditions expected by any specific technique. For example, a technique may require that a certain input be non-null, or that a numeric input falls within a certain range.

We highly recommend that, at the beginning of each of your technique methods, you test the validity of each of the user's inputs, that is, each of those parameter values (arguments) given to your method. (Exception: Ignore the 'updateContext' parameter.)

- `new NodeUpdateResult.TechniqueInvalidArguments(string nameOfInvalidArgument, Ls reasonForInvalidness)`

Example:

```
[Technique]
public NodeUpdateResult ByTribopAngle
(
    NodeUpdateContext          updateContext,
    [GCDgnModelProvider, GCReplicable] IPointNode ReferencePoint,
    double                      TribopAngle
)
{
    if (ReferencePoint is null)
        return new NodeUpdateResult.TechniqueInvalidArguments("ReferencePoint",
                                                                Ls.Literal("may not be null"));
    if (TribopAngle <= 0)
        return new NodeUpdateResult.TechniqueInvalidArguments("TribopAngle",
                                                                Ls.Literal("must be greater than zero"));
    :
}
```



## Handling multiple input errors at once

Looking at the preceding code example, we see that:

- If the user gives an invalid ReferencePoint (that is, if it's null), the technique method returns immediately with an appropriate error condition.
- If the user gives an invalid TribopAngle (that is, if it's  $\leq 0$ ), the technique method returns immediately with an appropriate error condition.

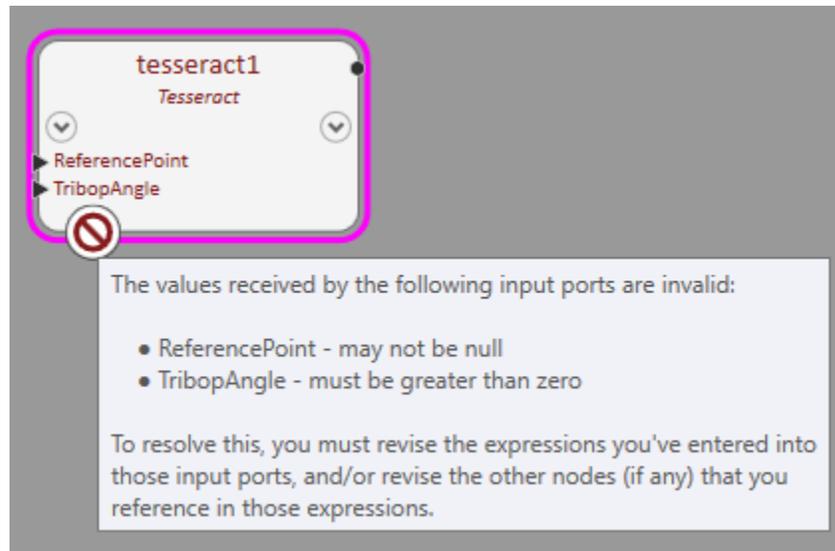
This is fine, and many technique methods work this way. However, putting ourselves in the user's shoes, it may be preferable to describe all the invalid inputs at once, so the user can see the whole picture before they start fixing their inputs. Here's how to do that:

- new `NodeUpdateResult.TechniqueInvalidArguments(IEnumerable<GCPair<string, Ls>> namesAndReasonsOfInvalidArguments)`

This overload of the constructor lets you pass as the list of string/message pairs. For example:

```
[GCTechnique]
public NodeUpdateResult ByTribopAngle
(
    NodeUpdateContext                updateContext,
    [GCDgnModelProvider, GCReplicable] IPointNode ReferencePoint,
    double                            TribopAngle
)
{
    List<GCPair<string, Ls>> argumentMessagePairs = new List<GCPair<string, Ls>>();
    if (ReferencePoint is null)
        argumentMessagePairs.Add(GCPair.Create("ReferencePoint", Ls.Literal("may not be null")));
    if (TribopAngle <= 0)
        argumentMessagePairs.Add(GCPair.Create("TribopAngle",
                                                Ls.Literal("must be greater than zero")));
    if (argumentMessagePairs.Count > 0)
        return new NodeUpdateResult.TechniqueInvalidArguments(argumentMessagePairs);
    :
}
```

Now the user will see all of their mistakes at once:

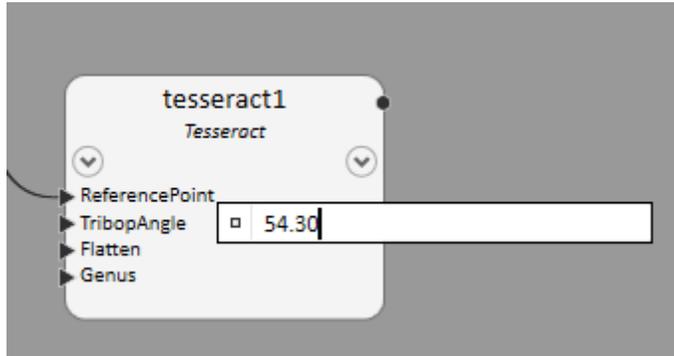


**Update:** The next release of GC will support C# value tuples as an alternative to GCPairs. When that happens, the preceding code can be (somewhat) simplified to:

```
[GCTechnique]
public NodeUpdateResult ByTribopAngle
(
    NodeUpdateContext          updateContext,
    [GCDgnModelProvider, GCReplicable] IPointNode ReferencePoint,
    double                      TribopAngle
)
{
    List<(string, Ls)> argumentMessagePairs = new List<(string, Ls)>();
    if (ReferencePoint is null)
        argumentMessagePairs.Add(("ReferencePoint", Ls.Literal("may not be null")));
    if (TribopAngle <= 0)
        argumentMessagePairs.Add(("TribopAngle", Ls.Literal("must be greater than zero")));
    if (argumentMessagePairs.Count > 0)
        return new NodeUpdateResult.TechniqueInvalidArguments(argumentMessagePairs);
    :
}
```

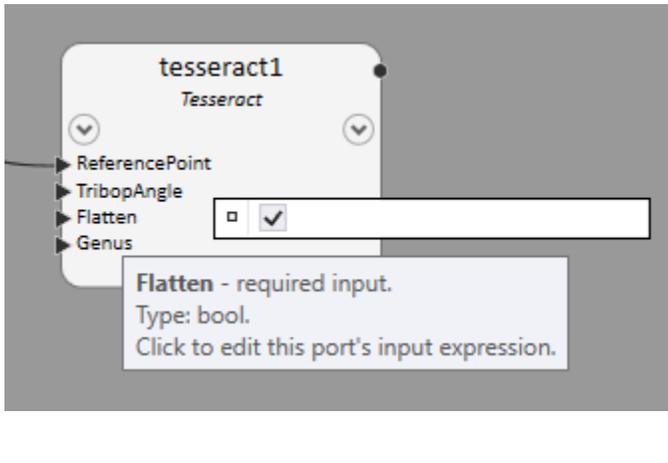
## 2.2. Customizing the pop-up property editor

The pop-up property editor is that thing the user sees when they hover over a node's input. They can click within it to manually edit that script:

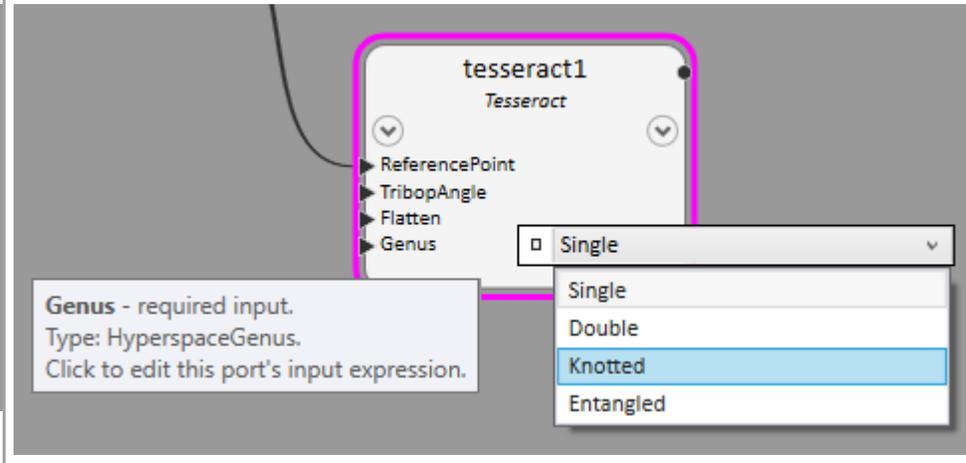


Some customizations are already provided by GC:

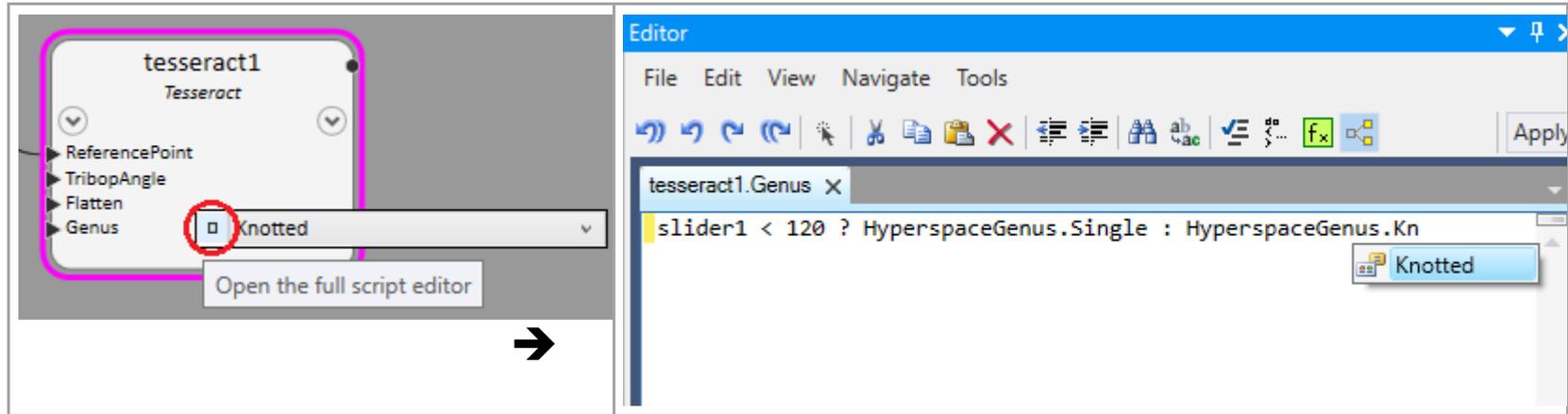
If the type of the input property is 'bool', GC presents it as a checkbox.



If the type of the input property is an enum type, GC presents it as a drop-down list (also known as a combobox).



Regardless of how it's customized, the user always has the option to work directly with the underlying script, in case they want to use an expression beyond merely 'true', 'false', or an enum value selected from the list:



But, what if you want to customize the appearance and behavior of a property's input beyond these automatic presentations? The subpages explain how.

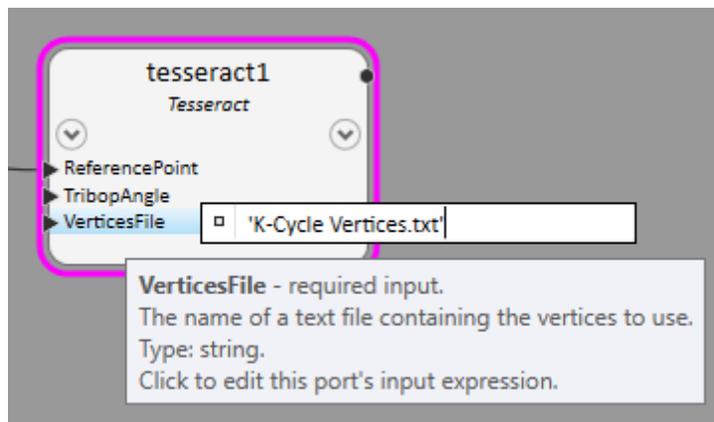
- [GCFileBrowser attribute](#)
- [Other custom editor behaviors](#)

## 2.2.1. File browser

This is applicable when an input property is a string that your technique will process as a file name. For example, an input property named VerticesFile:

```
[GCTechnique]
[GCPParameter("VerticesFile", "The name of a text file containing the vertices to use.")]
public NodeUpdateResult ByTribopAngle
(
    NodeUpdateContext updateContext,
    [GCDgnModelProvider, GCReplicatable] IPointNode ReferencePoint,
    double TribopAngle,
    string VerticesFile
)
{
    :
```

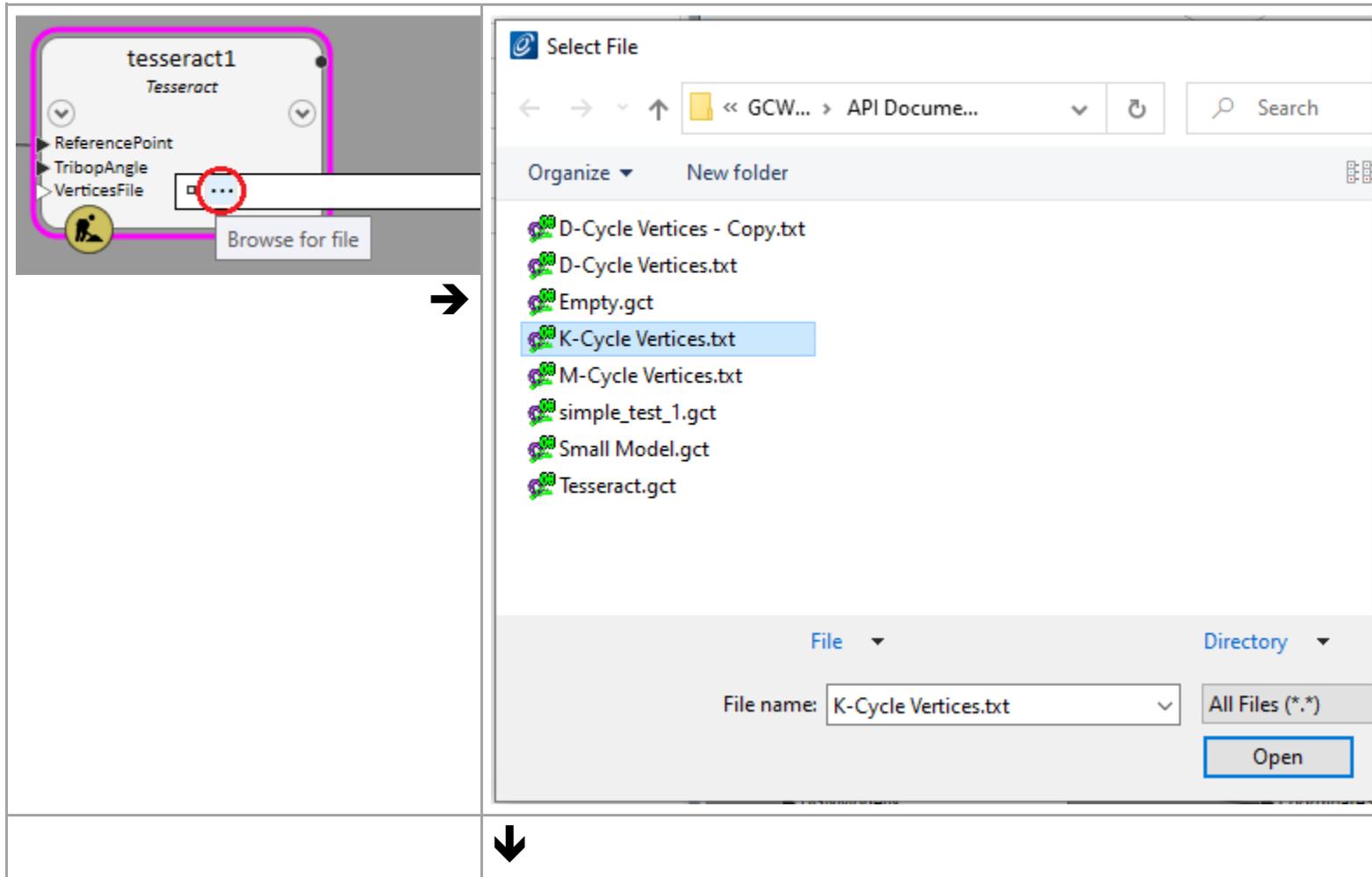
Without a customized property editor, it's presented as simply a string-type property. It's the user's responsibility to manually enter the file name they want. (It's a string, so it must be quoted, like any script string.)

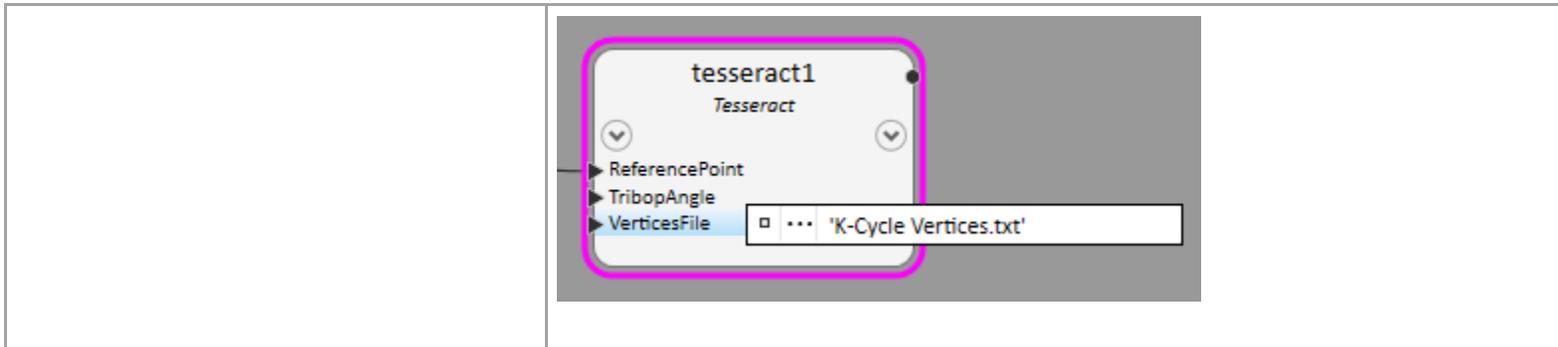


To introduce a proper file browser, we'll apply the attribute, [GCFileBrowser], to our VerticesFile input:

```
[GCFileBrowser] string VerticesFile
```

Merely doing that provides the user with a passable file browser:





- **Configuring the file browser**

Furthermore, that attribute, [GCFileBrowser], offers some optional parameters we can use to configure the appearance and/or behavior of the browser. Here's the complete definition of GCFileBrowserAttribute's constructor:

```
public GCFileBrowserAttribute
(
    FileBrowserMode mode           = FileBrowserMode.InputFile,
    string          buttonToolTip  = default,
    string          dialogTitle    = default,
    string          relativeFolderPath = null,
    string          fileFilterDescription = default,
    string          fileFilterMask   = null
)
```

As you can see, all parameters are optional (because they all have default values). That's why you can get away with simply saying:

```
[GCFileBrowser] string VerticesFile
```

...which is equivalent to:

```
[GCFileBrowser()] string VerticesFile // empty parameter list (using all default values)
```

As you may know, whenever you call a method (including a constructor method, including a constructor method on an attribute), you can specify the value of any parameter by prefacing that value with the name of the parameter plus a colon (:). For example:

```
[GCFileBrowser(buttonToolTip:"Browse for an X-Cycle vertices file"),
  dialogTitle:"Select X-Cycle file for Tesseract VerticesFile input")]
string VerticesFile
```

This powerful feature of C# lets us pick and choose which options we want, in any order, without worrying about the options we don't care about (which will therefore have their default values). For the GCFileBrowser attribute, those options are:

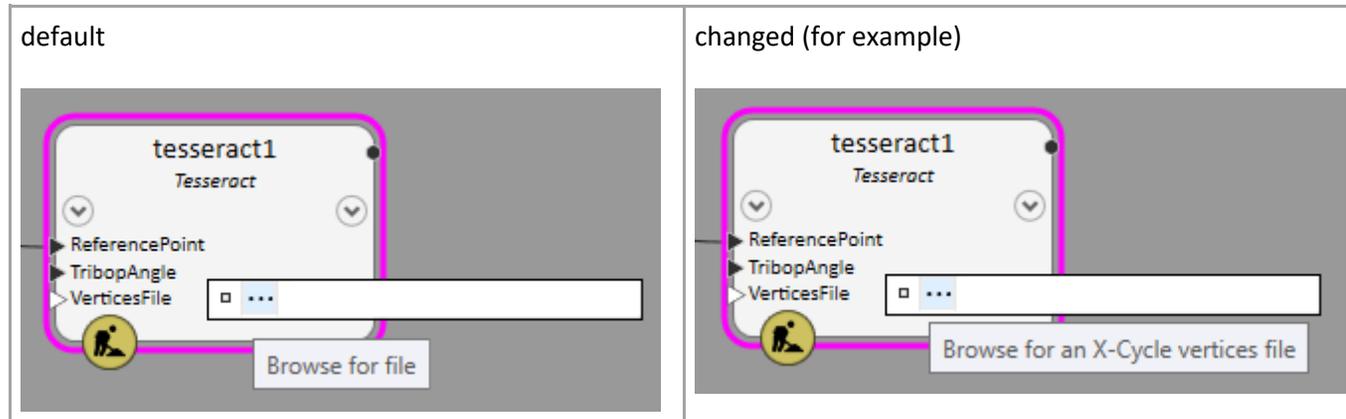
- mode (of the type FileBrowserMode) lets you specify the general behavior of the browser, as described here:

```
public enum FileBrowserMode
{
    InputFile,    // The user selects an existing file. This is the default.

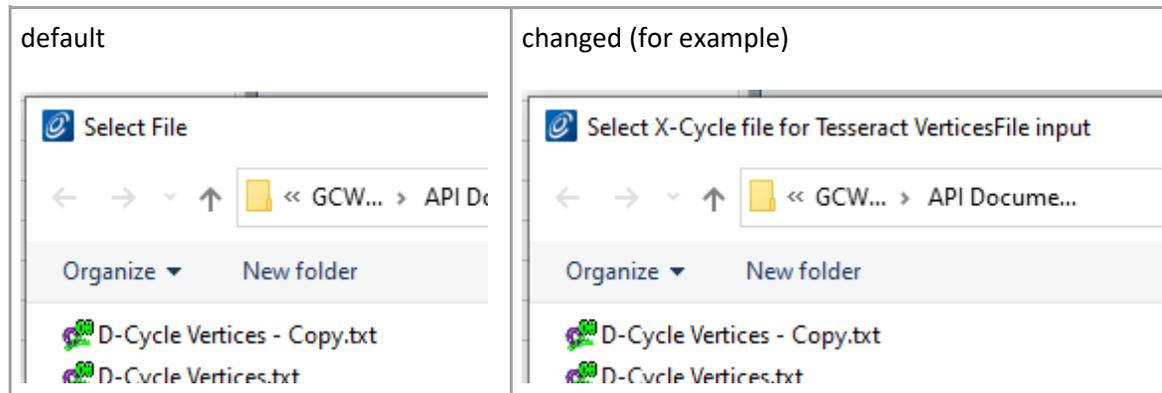
    OutputFile,  // The user selects an existing file or types the name of a new file.
                // If the former, the user is prompted to confirm overwriting that
                // existing file.

    FolderOnly,  // The user selects an existing folder rather than a file. The user
                // is given the opportunity to create new folders spontaneously.
}
}
```

- `buttonToolTip` (string) lets you change the tooltip of the open-browser button.



- `dialogTitle` (string) lets you change the title of the browser dialog.



- `relativeFolderPath` (string) lets you change the initial folder that's shown in the browser dialog.

Note that this folder is specified relative to the "home" folder, that is, the folder that contains the current top-level design file or GC transaction file. If you want to specify an absolute path, you can do so by specifying a path that's "rooted", as described [here](#) (internet link to Microsoft documentation).

Examples:

```
[GCFileBrowser(..., relativeFolderPath:"Vertices", ...)]
    // A subfolder named "Vertices" within the current home folder.

[GCFileBrowser(..., relativeFolderPath:@"..\Vertices"), ...]
    // A subfolder named "Vertices" within the parent folder of the current home
    // folder.

// Above and below, note the use of the '@' string prefix, which defines a C# "verbatim"
// string, which means (among other things) we don't need to replace each backslash (\)
// with a double backslash (\\) as we would need to otherwise.

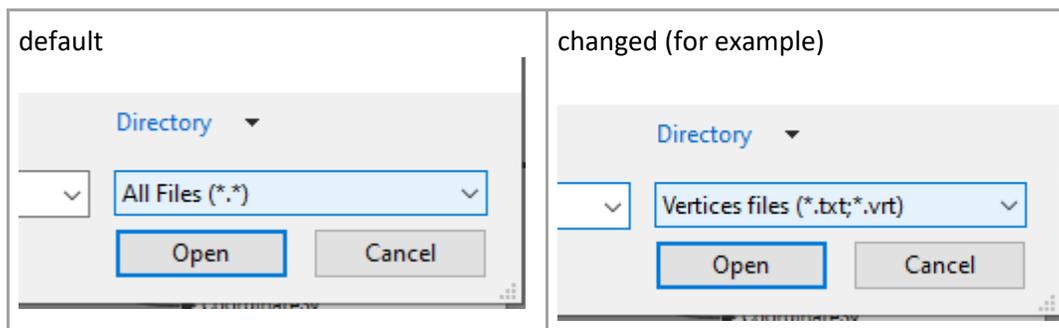
[GCFileBrowser(..., relativeFolderPath:@"D:\Research Results", ...)]
    // A folder named "D:\Research Results", unaffected by the current home folder.
```

- fileFilterDescription (string) and fileFilterMask (string)

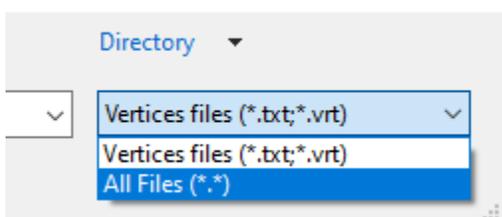
These two parameters are almost always used together.

- `fileFilterDescription` defines a description of the kinds of files you expect the user to choose; for example, "Vertices files".
- `fileFilterMask` defines a file mask for those files, typically in the form "\*.extension"; for example, "\*.txt". You can specify multiple masks by separating them with commas or semicolons; for example, "\*.txt,\*.vrt".

```
[GCFileBrowser(..., fileFilterDescription:"Vertices files", fileFilterMask:"*.txt;*.vrt")]
```



As with any well-behaved file-browser dialog, "All files" is always offered as an option:



## 2.2.2. Other custom editor behaviors (overview)

To add a file browser to any of your node's inputs, you can use the [GCFileBrowser attribute](#).

For other customization tasks, you'll need to create a customized editor. Here's how to do that.

- **"EECC"**

You'll be working with a GC class named `ExpressionEditorCustomConfiguration` as well as a GC attribute named `GCExpressionEditorCustomConfigurationAttribute`. Since those names are mouthfuls, we conventionally abbreviate them to "EECC" and "EECCAttr", respectively.

For any of your project's source files that will work with an `ExpressionEditorCustomConfiguration`, you should add the following 'using' statement and alias-name statements at the top. (The latter goes under your 'namespace' statement.)

```
:  
using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;  
  
namespace MyAddIn  
{  
    using EECC = ExpressionEditorCustomConfiguration;  
    using EECCAttr = GCExpressionEditorCustomConfigurationAttribute;  
  
    :  
}
```

(This requires that you also have a 'using' statement for `Bentley.GenerativeComponents.ElementBasedNodes`, but we assume you already have that.)

Now you can use those abbreviated names in your code.

- **Write a method that returns an EECC, and associate it with the appropriate technique parameter**
- Anywhere within your node class that contains the technique method parameter you want to customize, add a method having this general form:

```
static EECC GetEECCForPropertyName() // e.g., GetEECCForTribopAngle
{
    :
    return new EECC(...);
}
```

- Then use EECCAttr to associate your method with that technique method parameter:

```
[Technique]
public NodeUpdateResult ByTribopAngle
(
    NodeUpdateContext updateContext,
    :
    [EECCAttr(nameof(GetEECCForTribopAngle))] double TribopAngle,
    :
)
```

Notice how we're using C#'s 'nameof' operator to protect ourselves from accidentally misspelling the name of the method. That line is functionally equivalent to:

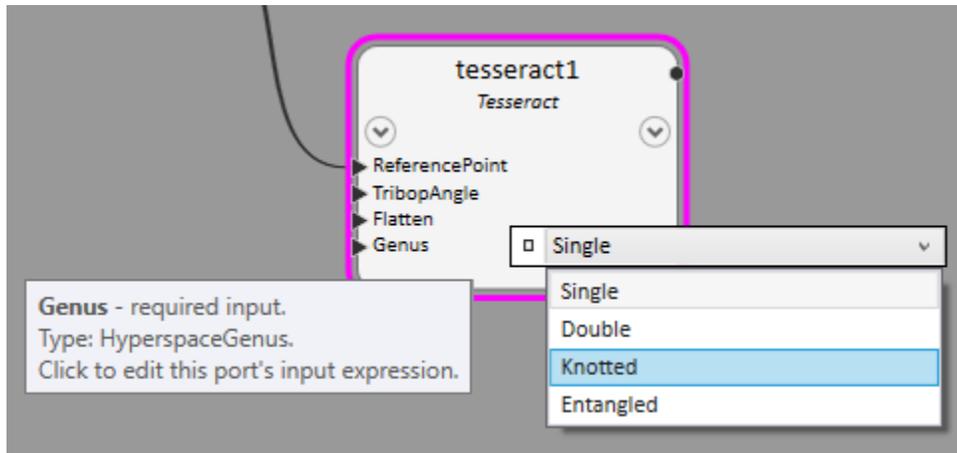
```
[EECCAttr("GetEECCForTribopAngle")] double TribopAngle,
```

And now you understand it all... no further explanation is needed! Good luck! 😊

Oh, wait.. did we neglect to explain what goes on in the "GetEECC" method? Let's do that in the following pages...

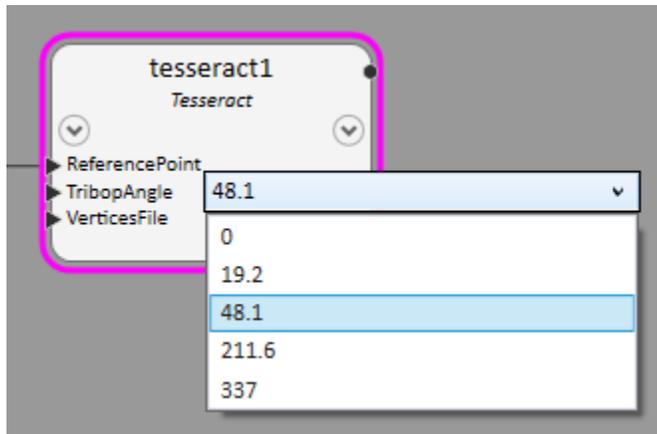
## 2.2.3. Custom lists

As we noted [previously](#), GC intrinsically customizes each enum-type property such that it manifests to the user as a drop-down list of values (combobox):



But there may be situations in which you want to present a drop-drop list of things other than enum values. For example, a list of hard-coded strings, or values that aren't even known until runtime.

Let's continue our example from [the previous page](#). Let's suppose that a tesseract's tribop angle (whatever that is!) should be chosen from a specific set of values. The property TribopAngle isn't an enum type, it's a double. Nonetheless, we'd like to produce a dropdown list of the recommended options:



Here's the code that produces that:

```
1 :
2 using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;
3
4 namespace MyAddIn
5 {
6     using EECC = ExpressionEditorCustomConfiguration;
7     using EECCAttr = GCExpressionEditorCustomConfigurationAttribute;
8
9     [GCNamespace("User"), GCNodeTypePaletteCategory("My Add-In")]
10    public class TesseractNode: ElementBasedNode
11    {
12        static EECC GetEECCForTribopAngle()
13        {
14            return new EECC(getScriptChoices);
15
16            IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
17            {
18                TesseractNode node = (TesseractNode) parentExpressionEditor.MeaningOfThis;
19                foreach (double angle in node.RecommendedTribopAngles())
20                    yield return new ScriptChoice(angle.ToString());
21            }
22        }
23    }
```

```

23
24     IEnumerable<double> RecommendedTribopAngles()
25     {
26         // This method simulates determining the angle choices at runtime, based on the
27         // current state of this node or on something else in the environment.
28         //
29         // (For this example, we're simply hard-coding the choices.)
30
31         return new double[] {0.0, 19.2, 48.1, 211.6, 337.0};
32     }
33
34     :
35
36     [GCTechnique]
37     public NodeUpdateResult ByTribopAngle
38     (
39         NodeUpdateContext updateContext,
40         [GCDgnModelProvider, GCReplicatable] IPointNode ReferencePoint,
41         [EECCAttr(nameof(GetEECCForTribopAngle))] double TribopAngle,
42         string VerticesFile
43     )
44     {
45         :
46         return NodeUpdateResult.Success;
47     }
48
49     :
50
51     } // class TesseractNode
52 } // namespace MyAddIn

```

- In line 14, you can see we're returning a new EECC, as required by the enclosing method, GetEECCForTribopAngle.

EECC has multiple constructors. In this case, we're calling this one:

```

public EECC(GetScriptChoicesDelegate methodToGetScriptChoices,
            ScriptChoiceOptions scriptChoiceOptions=ScriptChoiceOptions.None)

```

That looks rather arcane, doesn't it? What's a GetScriptChoicesDelegate? What are ScriptChoiceOptions?

For now, let's ignore the latter. Here's the definition of GetScriptChoicesDelegate:

```
public delegate IEnumerable<ScriptChoice> GetScriptChoicesDelegate(ExpressionEditor parentExpressionEditor);
```

So, the EEE constructor's first argument must be a method whose signature matches that of the delegate.

- Line 16 defines the aforementioned method (defined as a C# local function). You can see that the signature of 'getScriptChoices' matches that of the delegate: It takes an ExpressionEditor as its parameter and returns an IEnumerable<ScriptChoice> as its result.
- In line 18, we're getting the current instance of our node, that is, the instance upon which the user has invoked the expression editor for the TribopAngle input.
- On lines 19 and 20, we're calling a method on our node instance to get the values that are valid in the current circumstance. Then, since, the return type of 'getScriptChoices' is an IEnumerable<ScriptChoice> rather than IEnumerable<double>, we generate an appropriate ScriptChoice for each value.

Also in line 20, we're creating a new ScriptChoice by calling this constructor:

```
public ScriptChoice(string displayString, string scriptTextIfDifferentFromDisplayString=null,
                   ScriptChoiceSpecialActionDelegate actionWhenSelected=null,
                   ScriptingLanguage language=default)
```

Hmm, several optional parameters there. Let's just look at the simplest case that we use in line 20: We're passing a string as the first argument (i.e., the value of the first parameter).

Since that value must be a string, we're calling the ToString() method on each angle.

### But what's a ScriptChoice?

When you're customizing a GC expression editor drop-down, the values you specify must be instances of ScriptChoice, not instances of strings. A ScriptChoice gives us more control over the appearance and behavior of each item than a simple string would. (That's not demonstrated by this particular example, but you'll see it in the following pages.)

[Letting the user override the provided choices](#)

[Controlling what's displayed for each item](#)

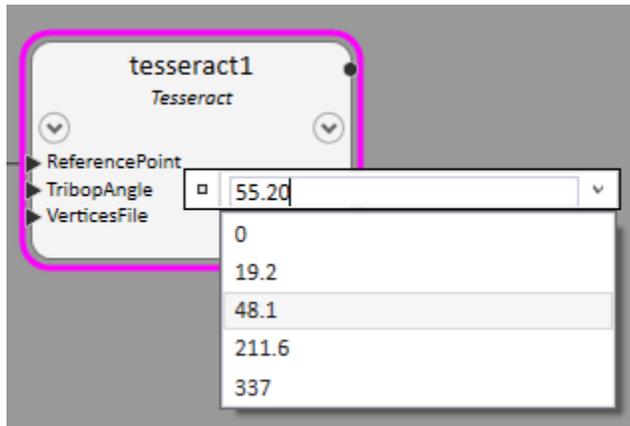
[Special considerations for string data](#)

[Grouping](#)

### 2.2.3.1. Letting the user override the provided choices

```
return new EECC(getScriptChoices, ScriptChoiceOptions.UserMayOverrideChoices);
```

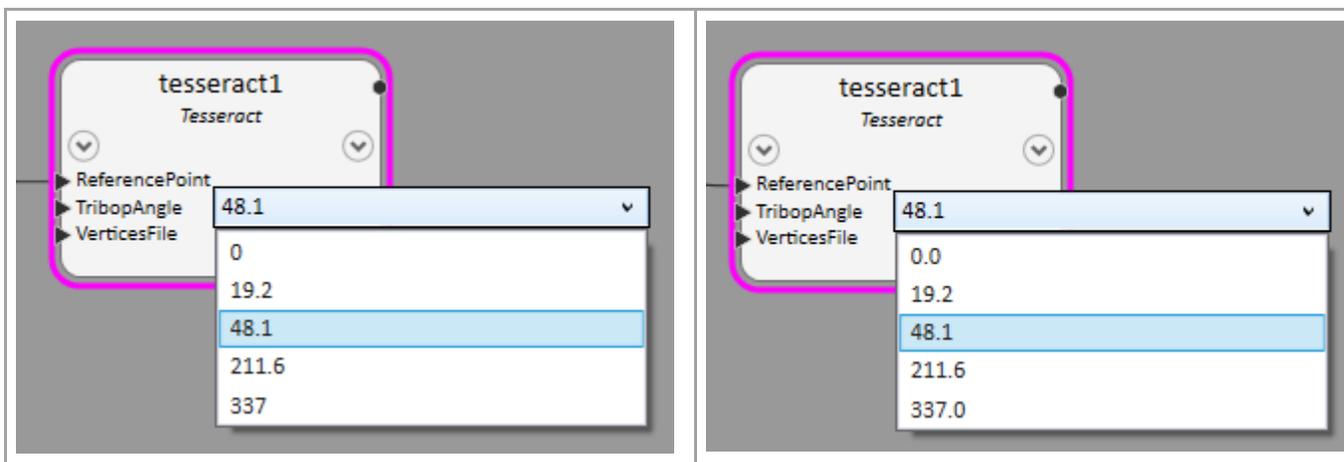
And now the user can type their own value directly into the edit box, ignoring all the suggested options.



## 2.2.3.2. Controlling what's displayed for each item

- **Formatting the item's string value**

Looking again at TribopAngle's drop-down list, you may think it's "sloppy" in that some numbers include decimals and some don't, as shown on the left, below. You may prefer that they're all consistent, as shown on the right.



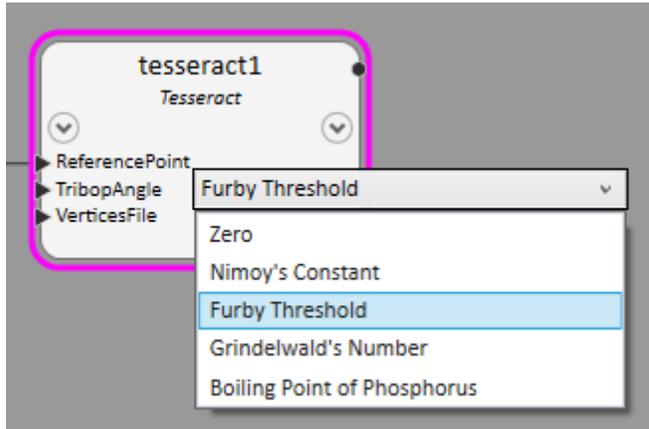
As you may know, when you call .NET's ToString() method, you can pass a format string, such as "F1". It's beyond the scope of this documentation to describe .NET's format strings; on the Microsoft site, that topic begins [here](#), with [here](#) describing the format strings specific to numeric types.

To achieve the result above, we simply add the format string "F1" -- which means, "format as a number having one digit after the decimal point" -- to [our ToString call on each numeric value](#):

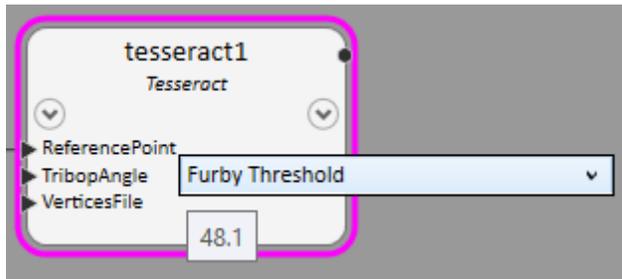
```
yield return new ScriptChoice(angle.ToString("F1"));
```

- **Displaying something entirely different from the underlying value**

Since those TribopAngle values seem arbitrary (well, because they are), let's suppose each one has a corresponding descriptive name, such as "Nimoy's Constant" or "Furby Threshold". In that case, we'd like to display our values like this:



It's important to realize we're not changing the value being assigned to the property, only the presentation of that value. For example, if the user chooses Furby Threshold, which is associated with the number 48.1, the node technique method will receive 48.1 as its input. That method doesn't know or care, about what the user sees in the UI.



Here's the code that accomplishes that:

```
1 static EECC GetEECCForTribopAngle()
2 {
3     return new EECC(getScriptChoices);
4
5     IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
6     {
7         TesseractNode node = (TesseractNode) parentExpressionEditor.MeaningOfThis;
8         foreach (var tuple in node.RecommendedTribopAngles())
9         {
10            double angle      = tuple.Item1;
11            string description = tuple.Item2;
12
13            yield return new ScriptChoice(description, angle.ToString());
14        }
15    }
16 }
17
18 IEnumerable<(double, string)> RecommendedTribopAngles()
19 {
20     // This method simulates determining the angles and their associated descriptions,
21     // at runtime, based on the current state of this node or of something else in the
22     // environment.
23     //
24     // This method uses a C# construct called a "tuple" (or "value tuple") to return
25     // each angle / description pair.
26     //
27     // (For this example, we're simply hard-coding the results.)
28
29     return new (double, string)[] {( 0.0, "Zero"),
30                                     ( 19.2, "Nimoy's Constant"),
31                                     ( 48.1, "Furby Threshold"),
32                                     (211.6, "Grindelwald's Number"),
33                                     (337.0, "Boiling Point of Phosphorus")};
34 }
```

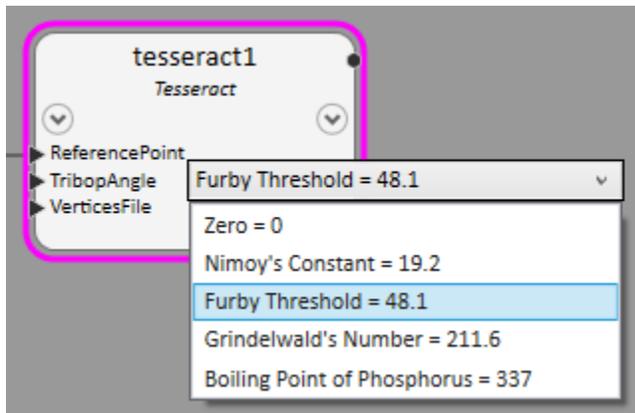
At line 13, we're calling the same overload of ScriptChoice's constructor we've been calling before:

```
public ScriptChoice(string displayString, string scriptTextIfDifferentFromDisplayString=null,
                   ScriptChoiceSpecialActionDelegate actionWhenSelected=null,
                   ScriptingLanguage language=default)
```

...but now we're making use of the second argument.

- So, in this case, the first argument to that constructor is whatever we want to display to the user. This value is only for presentation purposes; it will never be seen, nor otherwise processed, by the result of GC.
- The second argument is the script text that will be processed by GC, to determine the value to be assigned to the property. Happily, in this case, simply converting a C# number to a string is sufficient for GC script, as well. (That's not true with all types; see the page [Special considerations for string data.](#))
- **Another example**

Suppose we want to show the user not just the name of each number, but also the number, itself. Like this:



This is easy when we realize that each display string can be anything we want; it doesn't need to correlate with anything else in GC. For example, instead of this:

```
yield return new ScriptChoice(description, angle.ToString());
```

...we could say this:

```
string displayString = $"{description} = {angle}";  
yield return new ScriptChoice(displayString, angle.ToString());
```

This demonstrates a C# feature called string interpolation, which can be very powerful. It's described on Microsoft's site [here](#).

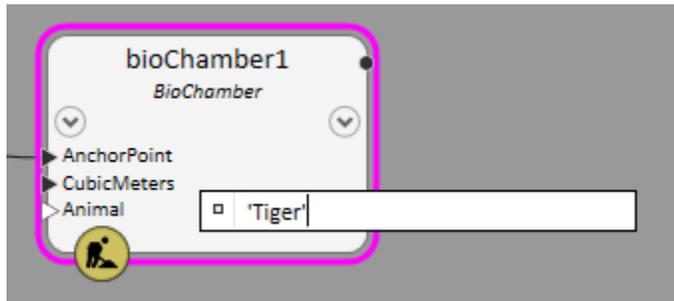
Of course, we're not limited to just string interpolation. We might say:

```
string displayString;  
: // Many lines of code, possibly involving 'if' statements and whatnot,  
: // that formulate the string we want to display to the user and then  
: // assign that value to 'displayString'.  
yield return new ScriptChoice(displayString, angle.ToString());
```

...which allows each display string to be as informative as possible.

### 2.2.3.3. Special considerations for string data

Introducing the BioChamber node! Of course, as everyone knows, a bio-chamber is designed to support a particular kind of animal.



The user can type in the name of any animal, but that assumes they know what the valid options are. (Bio-chambers are still being designed, and only certain animals are supported, so far.) So, let's provide the valid options as a drop-down list:

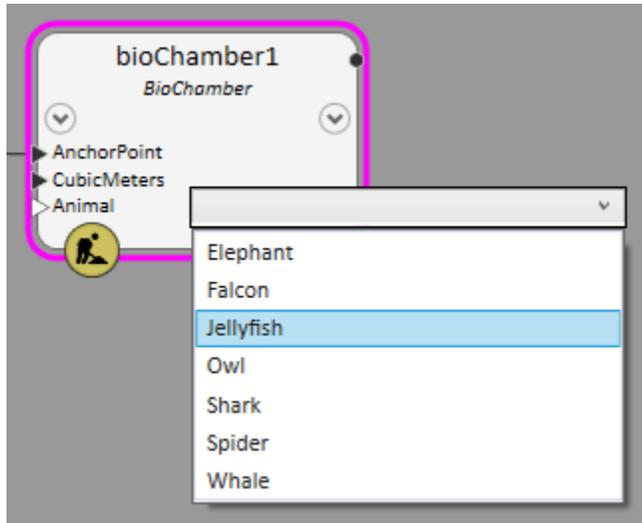
```

1 public class BioChamberNode: ElementBasedNode
2 {
3     static EECC GetEECCForAnimal()
4     {
5         return new EECC(getScriptChoices);
6
7         IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
8         {
9             BioChamberNode node = (BioChamberNode) parentExpressionEditor.MeaningOfThis;
10            foreach (string animal in node.Animals())
11                yield return new ScriptChoice(animal);
12        }
13    }
14
15    IEnumerable<string> Animals()
16    {
17        // Simulate determining the animals at runtime, based on the current state of this node.
18        // (For this example, we're simply hard-coding the result.)
19
20        return new string[] {"Elephant", "Falcon", "Jellyfish", "Owl", "Shark", "Spider", "Whale"};
21    }
22
23    [GCDefaultTechnique]
24    public NodeUpdateResult Default
25    (
26        NodeUpdateContext                updateContext,
27        [GCDgnModelProvider] IPointNode   AnchorPoint,
28        double                            CubicMeters,
29        [EECCAttr(nameof(GetEECCForAnimal))] string Animal
30    )
31    {
32        :

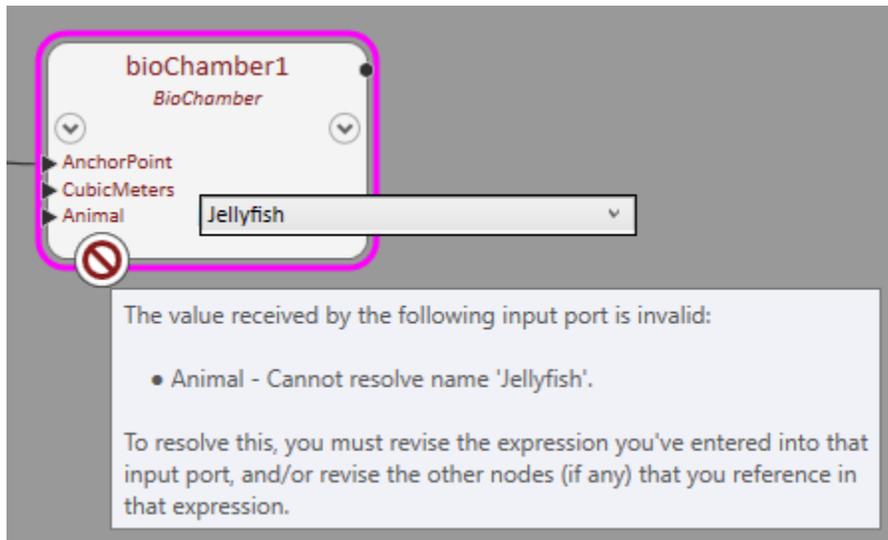
```

- In line 11, note that we don't need to call ToString() on the animalType (like we had to do with the angle in the [previous example](#)) because animalType is already a string. (But we're not out of the woods, yet...)

Now let's try it. It works like a dream!



Well, except that -- oops! -- when the user selects something, they'll see an error message like this:

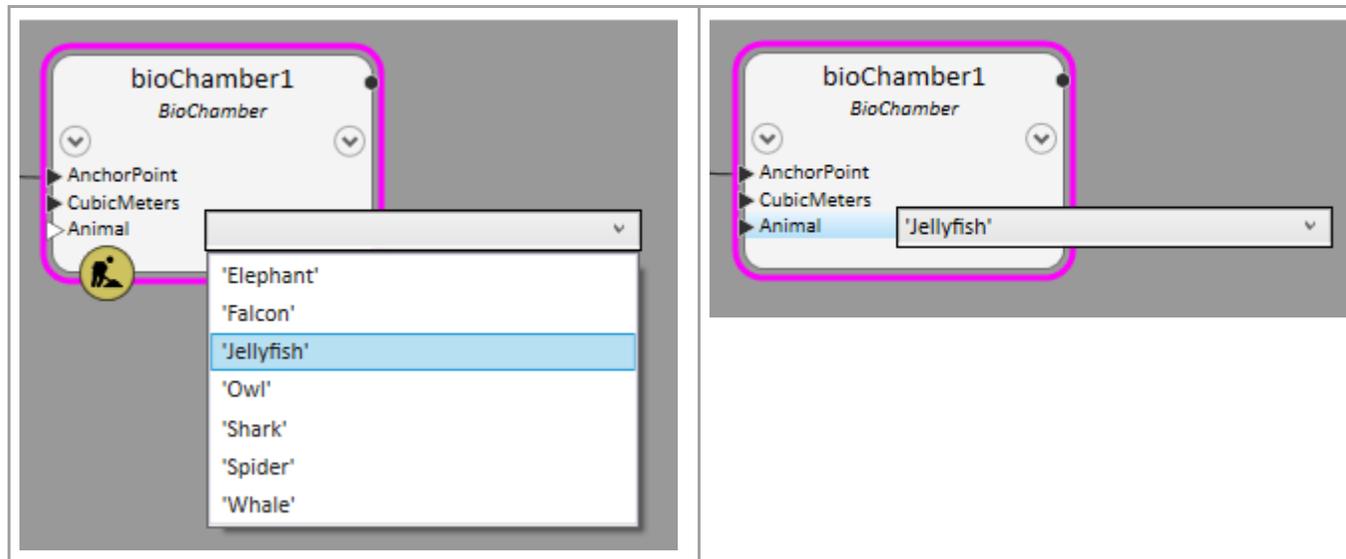


Oh, that's right: Anything that's entered in an expression editor -- whether via a drop-down list or any other means -- must be valuable as a legitimate script expression. Although a bare name like 'baseCS' would likely corollate with something that's known to GC, the bare name 'Jellyfish' does not; nothing in GC's world is named 'Jellyfish'.

So, we want to quote the name, so that the GC script sees it as a string literal rather than the name of something. That's easily accomplished by calling the extension method, `string.ToQuotedScriptText()`.

```
11 yield return new ScriptChoice(animal.ToQuotedScriptText());
```

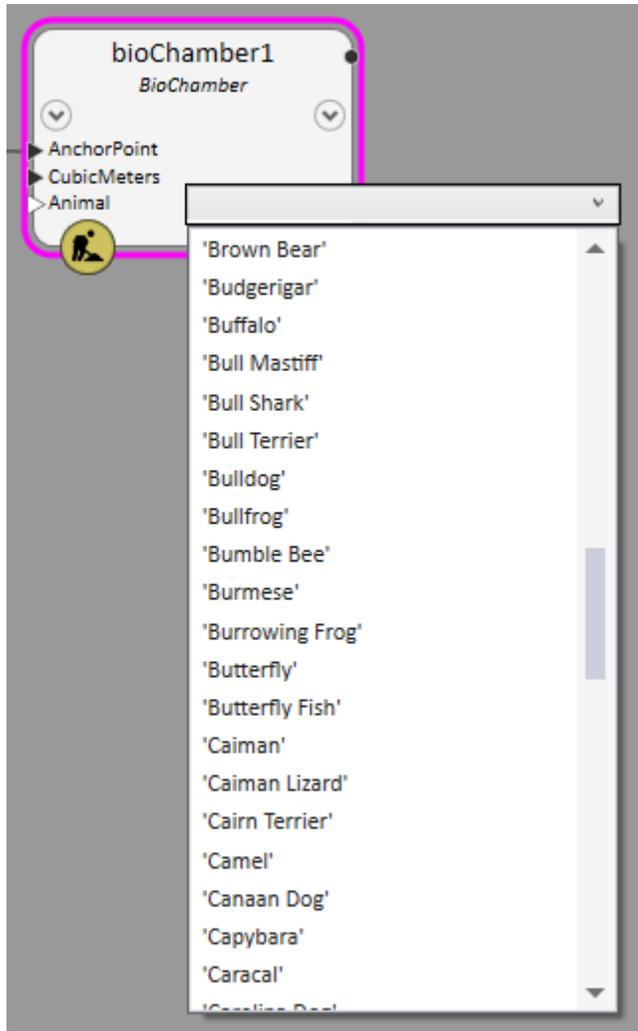
And now all the options are quoted:



This makes GC happy because it's being given a quoted string, which it understands to be an arbitrary string literal, not the name of something it's supposed to know about.

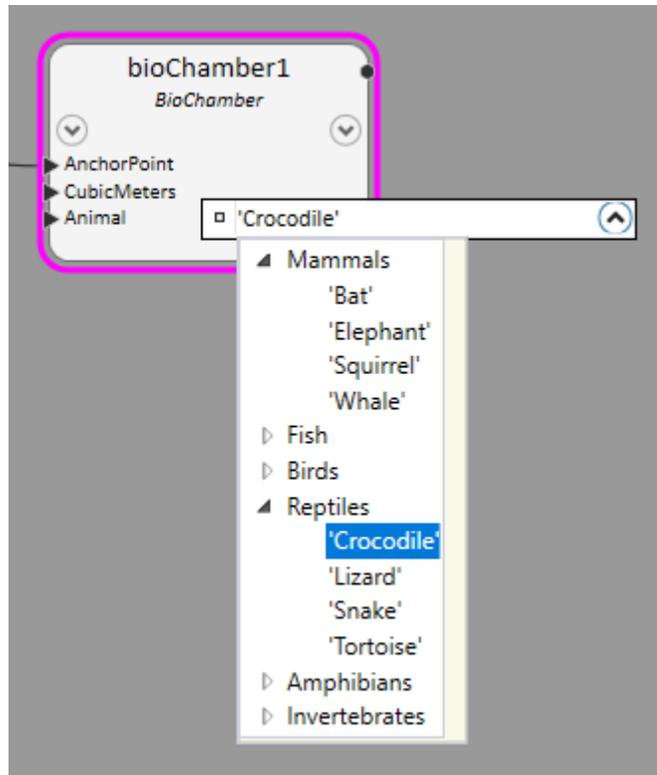
### 2.2.3.4. Grouping

As our bio-chambers continue to develop, we find we're able to support more and more kinds of animals. Eventually, our list might grow so long that it starts to become a bit overwhelming:



In a case like this, we might wish to apply grouping to our list of choices. For example, we may wish to group our animals based on their categories: Mammal, fish, bird, reptile, amphibian, and invertebrate.

What we'll get is something like this:



Here's an example of how to do that:

```
1 public class BioChamberNode: ElementBasedNode
2 {
3     static EECC GetEECCForAnimal()
4     {
5         return new EECC(getScriptChoices);
6
7         IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
8         {
9             BioChamberNode node = (BioChamberNode) parentExpressionEditor.MeaningOfThis;
10
11             ScriptChoiceList result = node.GetGroupedAnimals();
12
13             return result;
14         }
15     }
16
17     ScriptChoiceList GetGroupedAnimals()
18     {
19         // Simulate determining the animals at runtime, based on the current state of this node.
20         // (For this example, we're simply hard-coding the result.)
21
22         string[] mammals      = {"Bat", "Elephant", "Squirrel", "Whale"};
23         string[] fish         = {"Fish", "Eel", "Minnow", "Salmon", "Shark"};
24         string[] birds        = {"Chicken", "Falcon", "Owl", "Penguin"};
25         string[] reptiles     = {"Crocodile", "Lizard", "Snake", "Tortoise"};
26         string[] amphibians   = {"Frog", "Newt", "Salamander", "Toad"};
27         string[] invertebrates = {"Earthworm", "Insect", "Jellyfish", "Spider"};
28
29         ScriptChoiceList groupChoices = new ScriptChoiceList();
30
31         addGroupChoice("Mammals",      mammals);
32         addGroupChoice("Fish",         fish);
33         addGroupChoice("Birds",        birds);
34         addGroupChoice("Reptiles",     reptiles);
35         addGroupChoice("Amphibians",   amphibians);
36         addGroupChoice("Invertebrates", invertebrates);
37
38         return groupChoices;
39
40         void addGroupChoice(string groupName, string[] animals)
```

```

41     {
42         ScriptChoice groupChoice = groupChoices.Add(groupName);
43         foreach (string animal in animals)
44             groupChoice.AddSubChoice(animal.ToQuotedScriptText());
45     }
46 }
47
48 [GCDefaultTechnique]
49 public NodeUpdateResult Default
50 (
51     NodeUpdateContext                updateContext,
52     [GCDgnModelProvider] IPointNode  AnchorPoint,
53     double                            CubicMeters,
54     [EECCAttr(nameof(GetEECCForAnimal))] string Animal
55 )
56 {
57     :

```

Here we do things a little differently: The method `GetGroupedAnimals` (starting on line 17) doesn't merely determine what the animals and groups are; it programmatically builds the entire structure of choices that will be presented to the user. (The variable that holds that structure is named 'groupChoices'.)

The essence of that procedure is this:

```
40     void addGroupChoice(string groupName, string[] animals)
41     {
42         ScriptChoice groupChoice = groupChoices.Add(groupName);
43         foreach (string animal in animals)
44             groupChoice.AddSubChoice(animal.ToQuotedScriptText());
45     }
```

- At line 42, we call `groupChoices.Add(groupName)`, which adds a new `ScriptChoice` having the indicated display name (`groupName`), and returns that `ScriptChoice` so we can work further with it.
- In lines 43 and 44, we're adding subchoices for our animals that belong in the current group. As usual, we must call the extension method `string.ToQuotedScriptText()`, otherwise GC will try to interpret each animal's name as a known object in GC's environment.

Although it's not shown in this example, we could use the same approach to add further subchoices to each of our subchoices, to achieve multiple levels of grouping!

## 2.2.4. Custom expression editors, part 1

GC lets you create your custom expression editor that displays and behaves however you want it to.

◆ This topic assumes you're already familiar with writing WPF (Windows Presentation Foundation) user controls. It's far beyond the scope of this GC API documentation to teach that Microsoft technology.

A custom expression editor is a WPF user control that's derived from a GC class named Editor (which, in turn, is derived from EditorBase, which, in turn, is derived from WPF's UserControl).

As an example, we'll create a custom expression editor that lets the user enter and edit the BioChamber node's "DailySnacks" value in a friendly way.

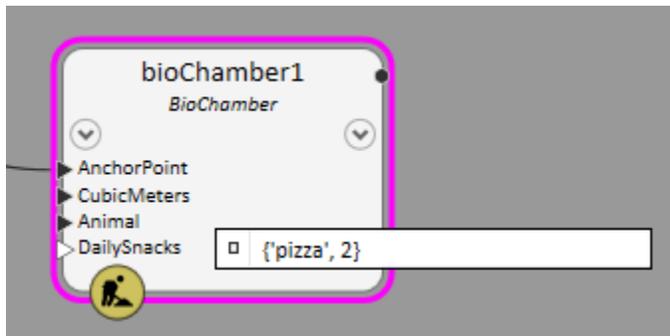
### 2.2.4.1. A new property on the BioChamber

We've been informed that a bio-chamber can record the daily snacks that should be given to the animals within. There may be any number of snacks given per day, but they must all be of the same type.

So, our new DailySnacks property will have two components: The name of a snack and a count.

We could break that into two separate properties -- DailySnackName and DailySnackCount -- but we want to be able to allow for additional snack components in the future (such as the cost of each snack). It's just tidier if we can have a single property take in those tightly-related data items.

Let's define a new property, DailySnacks, as a property that takes a list of two items, a snack name and snack count:



*The animals housed in this BioChamber get two pizzas per day... and that's just their snack!*

In the node implementation, that property looks like this:

```
public class BioChamberNode: ElementBasedNode
{
    :
    [GCDefaultTechnique]
    public NodeUpdateResult Default
    (
        NodeUpdateContext updateContext,
        :
        IGCObject[] DailySnacks
    )
    {
        :
        if (DailySnacks != null && DailySnacks.Length >= 2)
        {
            string name = DailySnacks[0].Unbox<string>();
            int count = DailySnacks[1].Unbox<int>();

            : // Do whatever we do with those values.
        }

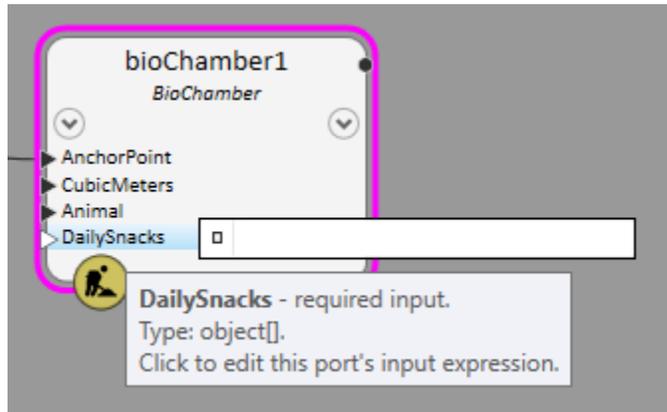
        return NodeUpdateResult.Success;
    }
}
```

This property demonstrates some things we haven't discussed before:

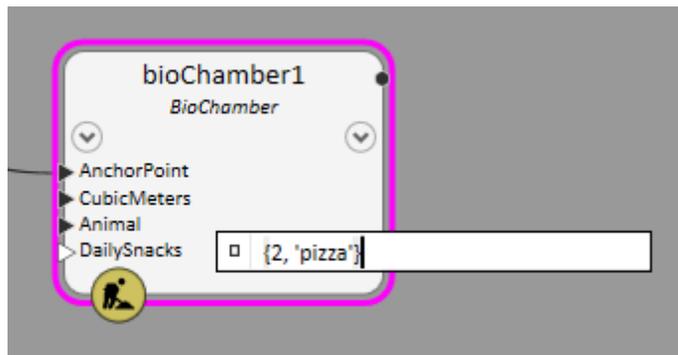
- When you want a node input to be able to accept a list containing an arbitrary number of items having arbitrary types, define that parameter's type as `IGCObject[]`.
- When processing each item in the given list, use the `Unbox<T>` method to get the C# value represented by that item.
- **Problems with this property**

The property 'DailySnacks' is useable as it stands, but it has problems from a user's perspective.

- The user does not get any prompts to indicate what they're supposed to enter. They can see the property's type is "object[]", which means it's a list, but what is the nature of that list?



- Even if the user understands what that list is for, nothing protects them from making mistakes, such as switching the positions of the snack name and count:



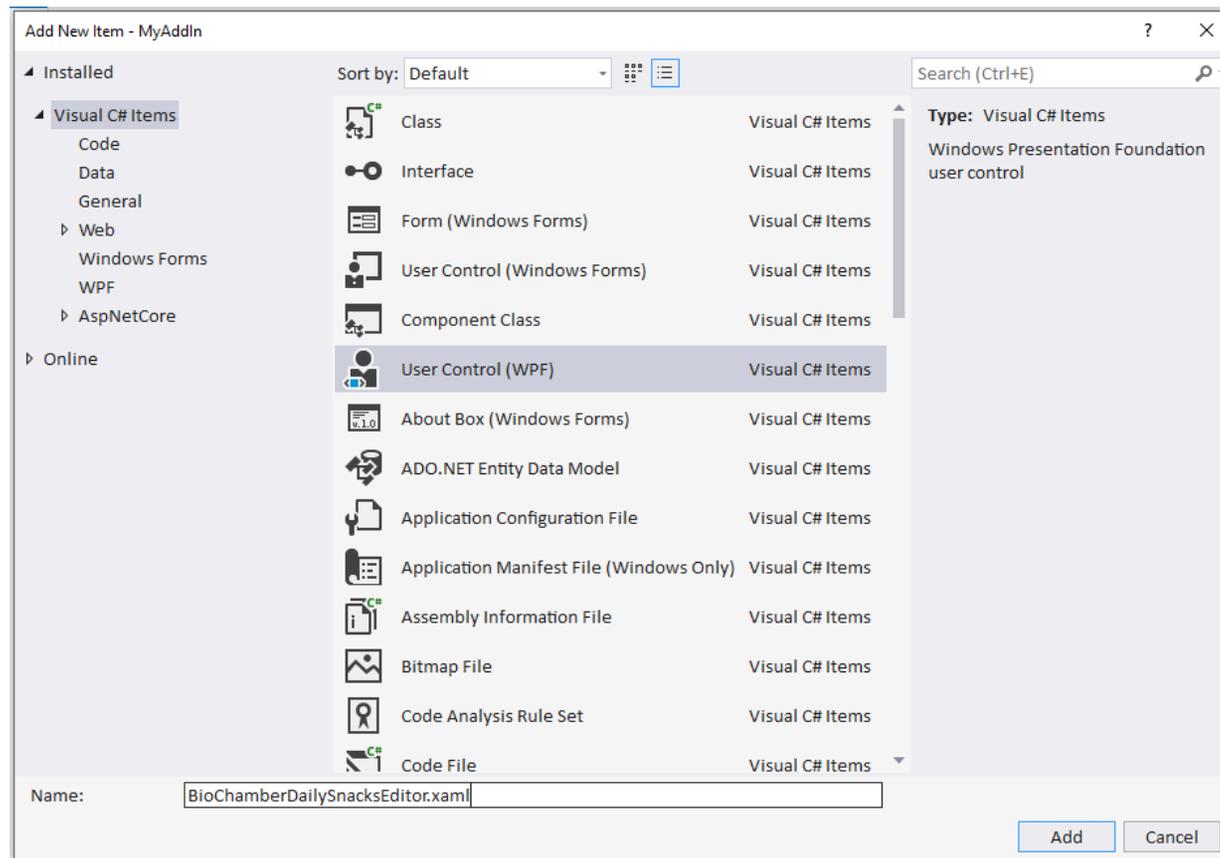
*Oops: The list should be {'pizza', 2}, not {2, 'pizza'}.*

Starting on the next page, we'll solve these problems by creating a custom expression editor that's just for entering daily snacks.

## 2.2.4.2. Getting started with a custom expression editor

- **Start by creating an empty custom expression editor**
- Create a WPF UserControl as usual. Name it after the node type and property it will be used for.

For our example, we'll name it "BioChamberDailySnacksEditor".



- Whenever Visual Studio creates every new WPF user control, it derives that class directly from UserControl.

However, in our case, we want our control to derive from the GC class 'Editor', so we'll manually make these changes to our new designer file (BioChamberDailySnacksEditor.xaml) and its associated the code file (BioChamberDailySnacksEditor.xaml.cs), respectively:

```
<gcse:Editor x:Class="MyAddIn.BioChamberDailySnacksEditor"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:local="clr-namespace:MyAddIn"
  xmlns:gcse="urn:GC.ScriptEditor"
  mc:Ignorable="d"
  d:DesignHeight="450" d:DesignWidth="800">
  <Grid>

  </Grid>
</gcse:Editor>
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;

namespace MyAddIn
{
  /// <summary>
  /// Interaction logic for BioChamberDailySnacksEditor.xaml
  /// </summary>
  public partial class BioChamberDailySnacksEditor : Editor
```

```
{
    public BioChamberDailySnacksEditor()
    {
        InitializeComponent();
    }
}

// Minimal method overrides we need to implement so our custom editor
// will be displayed, at least. We'll revisit these methods in the future.

protected override bool IsExpressionScriptRepresentableByThisEditor(Script script)
{
    return true;
}

protected override Script ExpressionScriptAsRepresentedByThisEditor()
{
    return default;
}

protected override UIElement OnActivatedWhileSelected()
{
    return default;
}

protected override void RefreshFromChangedExpressionScript(Script newScript, Script oldScript)
{
}
}
```

1. Back in our node class, let's add the additional code that tells GC to use our custom expression editor instead of one of its standard expression editors. For the most part, this code should be familiar.

```
public class BioChamberNode: ElementBasedNode
{
    :
    static EECC GetEECCForDailySnacks()
    {
        return new EECC(typeof(BioChamberDailySnacksEditor));
    }
    :
    [GCDefaultTechnique]
    public NodeUpdateResult Default
    (
        NodeUpdateContext updateContext,
        :
        [EECCAttr(nameof(GetEECCForDailySnacks)) IGCOBJECT[] DailySnacks
    )
    {
        :
        if (DailySnacks != null && DailySnacks.Length >= 2)
        {
            string name = DailySnacks[0].Unbox<string>();
            int count = DailySnacks[1].Unbox<int>();

            : // Do whatever we do with those values.
        }
    }
}
```

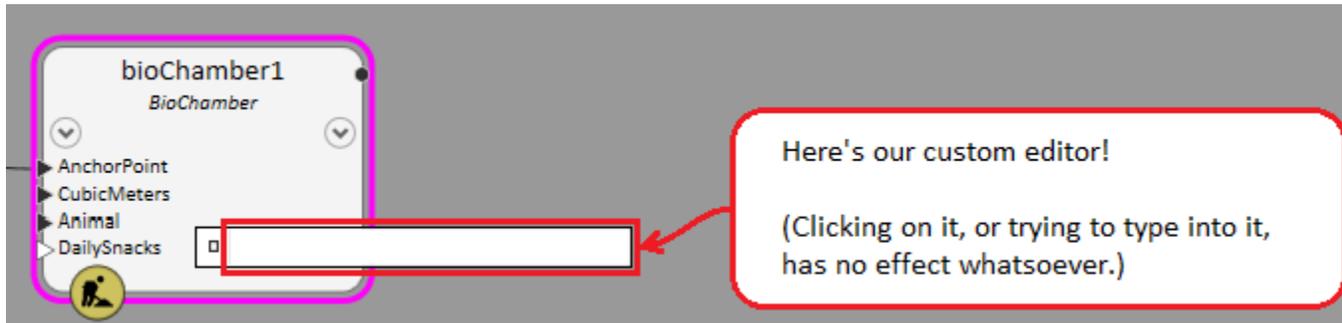
One new thing is the EECC constructor we're calling:

```
public ExpressionEditorCustomConfiguration(Type typeOfUserDefinedEditorOrExpandedContent)
```

Also, you may wonder why we defined the DailySnacks property as type "IGCOBJECT[]". That will become clearer as we flesh out this example.

- Trying out our new custom expression editor

At this point, enough pieces are in place that we can test-run our add-in project and see what our custom editor looks like! Here we go!



Hmm, not a very auspicious beginning. But, we're just getting started...

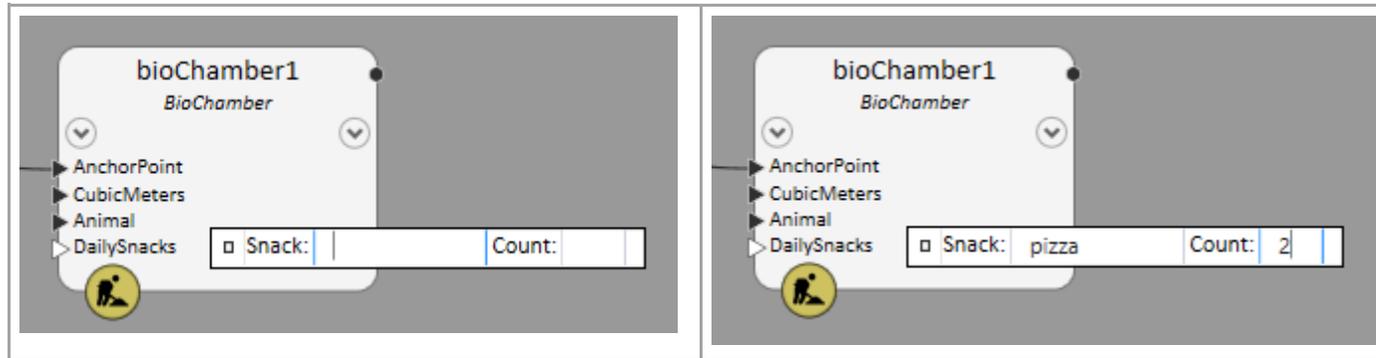
### 2.2.4.3. Making our custom expression editor useful

Let's add some controls to our custom editor...

```
<gcse:Editor x:Class="MyAddIn.BioChamberDailySnacksEditor"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:local="clr-namespace:MyAddIn"
  xmlns:gcse="urn:GC.ScriptEditor"
  mc:Ignorable="d"
  d:DesignHeight="450" d:DesignWidth="800">
  <StackPanel Orientation="Horizontal" >
    <TextBlock Text="Snack:" />
    <TextBox x:Name="_nameTextBox" Width="80" Margin="2,-4,2,0" FontSize="10" />
    <TextBlock Text="Count:" />
    <TextBox x:Name="_countTextBox" Width="30" Margin="2,-4,2,0" FontSize="10" />
  </StackPanel>
</gcse:Editor>
```

(I determined the Width, Margin, and FontSize values empirically.)

And give it another test run:



Aside from some minor alignment issues, this isn't bad. The user can see what they're supposed to do and enter those values accordingly.

◆ If you try this example (in the role of the user), you'll discover you can't use the Tab key to switch between the Snack and Count fields. The reason is that GC commandeers the Tab key for its nefarious purposes (specifically, for switching between inputs on the node). So, the user must click into whichever field (Snack or Count) they want to enter.

However, this custom editor is still just a "mockup" -- it doesn't connect with anything in the node. We'll do that next...

#### 2.2.4.4. Wiring up our custom editor to our node

Now we'll make use of those extra methods in our custom editor's code file, `BioChamberDailySnacksEditor.xml.cs`. Taking a more detailed look at those...

```
public partial class BioChamberDailySnacksEditor : Editor
{
    :

    protected override bool IsExpressionScriptRepresentableByThisEditor(Script script)
    {
        // We'll just always return true. In some implementations, it makes sense
        // to pre-examine the script to ensure it's something we can deal with.
        // But, in this case, we'll handle any invalid script when we actually try
        // to interpret the script, in the method RefreshFromChangedExpressionScript,
        // below.

        return true;
    }

    protected override Script ExpressionScriptAsRepresentedByThisEditor()
    {
        // Here we take the data components represented by this custom editor (i.e.,
        // a snack name and a snack count) and convert them to the script expression
        // that our node expects (i.e., a list: {'name', count}).

        return default;
    }

    protected override UIElement OnActivatedWhileSelected()
    {
        // This method determines which (if any) of our custom editor's child
        // controls should get the focus whenever this editor becomes active, and
        // returns that child control (or default, which is the same as null).

        return default;
    }

    protected override void RefreshFromChangedExpressionScript(Script newScript, Script oldScript)
    {
        // Here we take the script representation of this property's value, i.e. {'name', count},
```

```
    // and use it to populate the controls of this custom editor.
  }
}
```

So, the most significant methods (for this example) are:

```
protected override Script ExpressionScriptAsRepresentedByThisEditor()
{
    // Here we take the data components represented by this custom editor (i.e.,
    // a snack name and a snack count) and convert them to the script expression
    // that our node expects (i.e., a list: {'name', count}).

    return default;
}

protected override void RefreshFromChangedExpressionScript(Script newScript, Script oldScript)
{
    // Here we take the script representation of this property's value, i.e. {'name', count},
    // and use it to populate the controls of this custom editor.
}
```

- **ExpressionScriptAsRepresentedByThisEditor**

Remember we defined two TextBox controls [in our custom editor](#):

```
<TextBox x:Name="_nameTextBox" ... />
:
<TextBox x:Name="_countTextBox" ... />
```

So, when this method is called, we can expect the users' values to be held, as strings, in the properties `_nameTextBox.Text` and `_countTextBox.Text`.

Now we "only" need to write some C# code that generates the appropriate script code. This should do it:

```
protected override Script ExpressionScriptAsRepresentedByThisEditor()
{
```

```

// Here we take the data components represented by this custom editor (i.e.,
// a snack name and a snack count) and convert them to the script expression
// that our node expects (i.e., a list: {'name', count}).

string name = _nameTextBox.Text; // Get the name value.

int.TryParse(_countTextBox.Text, out int count); // Get the count value
// (or 0 if it's invalid).

StringBuilder builder = new StringBuilder(); // Build the script, e.g.:

builder.Append('{'); // {
builder.Append(name.ToQuotedScriptText()); // {'pizza'
builder.Append(", "); // {'pizza',
builder.Append(count); // {'pizza', 2
builder.Append('}'); // {'pizza', 2}

string scriptText = builder.ToString();

return scriptText.ToScript(); // Our return value must be of the type Script.
}

```

- **RefreshFromChangedExpressionScript**

This is the reverse operation: Convert a given script expression into its component parts, which we can then assign to the properties `_nameTextBox.Text` and `_countTextBox.Text`.

This code will do the trick:

```
protected override void RefreshFromChangedExpressionScript(Script newScript, Script oldScript)
{
    // Here we take the script representation of this property's value, i.e. {'name', count},
    // and use it to populate the controls of this custom editor.

    string name = ""; // Establish the default values we'll use
    int count = 0;    // in case the given 'newScript' is invalid.

    if (TryGetValueOfScript(out GCList value, newScript)) // GCList is a C# structure that
                                                         // corresponds to a script list.
    {
        // A GCList holds a collection of IGCOBjects, where each IGCOBject holds the C#
        // representation of a script object. To retrieve the underlying C# object from
        // each IGCOBject instance, we must "unbox" the instance.
        //
        // Actually, in this case, we're going to TRY to unbox it: The method 'TryUnbox'
        // returns false if the IGCOBject cannot be converted to a C# object of the
        // desired type.

        if (value.Count > 0 && value[0].TryUnbox(out string n))
            name = n;
        if (value.Count > 1 && value[1].TryUnbox(out int c))
            count = c;
    }

    _nameTextBox.Text = name; // Finally, we assign the values
    _countTextBox.Text = count.ToString(); // to their corresponding controls.
}
```

## 2.2.5. Custom expression editors, part 2

Here we'll make use of a custom, standalone dialog to enter our specialized data.

### 2.2.5.1. Revising the BioChamber property

Our manager has announced they want us to start tracking the unit cost of each snack item. So, we're going to change the script representation of each DailySnacks value from:

```
{'pizza', 2}
```

to:

```
{'pizza', 2, 7.95}
```

where the new third item indicates the cost of each pizza.

The first thing we'll do is update the definition of that technique parameter to accommodate that third item in the given list:

```
public class BioChamberNode: ElementBasedNode
{
    :

    [GCDefaultTechnique]
    public NodeUpdateResult Default
    (
        NodeUpdateContext                updateContext,
        [GCDgnModelProvider] IPointNode   AnchorPoint,
        double                            CubicMeters,
        [EECCAttr(nameof(GetEECCForAnimal))] string Animal,
        [EECCAttr(nameof(GetEECCForDailySnacks))] IGCObject[] DailySnacks
    )
    {
        // Process the other inputs... Then:

        if (DailySnacks != null && DailySnacks.Length >= 2)
        {
            string name    = DailySnacks[0].Unbox<string>();
            int    count    = DailySnacks[1].Unbox<int>();
            double unitCost = DailySnacks.Length > 2 ? DailySnacks[2].Unbox<double>() : 0.0;

            // Do whatever we do with those values.
        }

        return NodeUpdateResult.Success;
    }
}
```

Note that, in our new statement, we're using a conditional test so that we extract the third item only if it's present. This allows our 'DailySnacks' property to be backward-compatible with any values that were entered previously (which don't have that third item).

- **While we're here...**

While we're working in BioChamberNode.cs, anyway, let's add this additional method at the bottom of the class. This method's job is to return a valid list of snacks based on the currently-selected animal.

Later in this tutorial, we'll call this method to provide valid snack choices to our custom dialog.

```
public class BioChamberNode: ElementBasedNode
{
    :

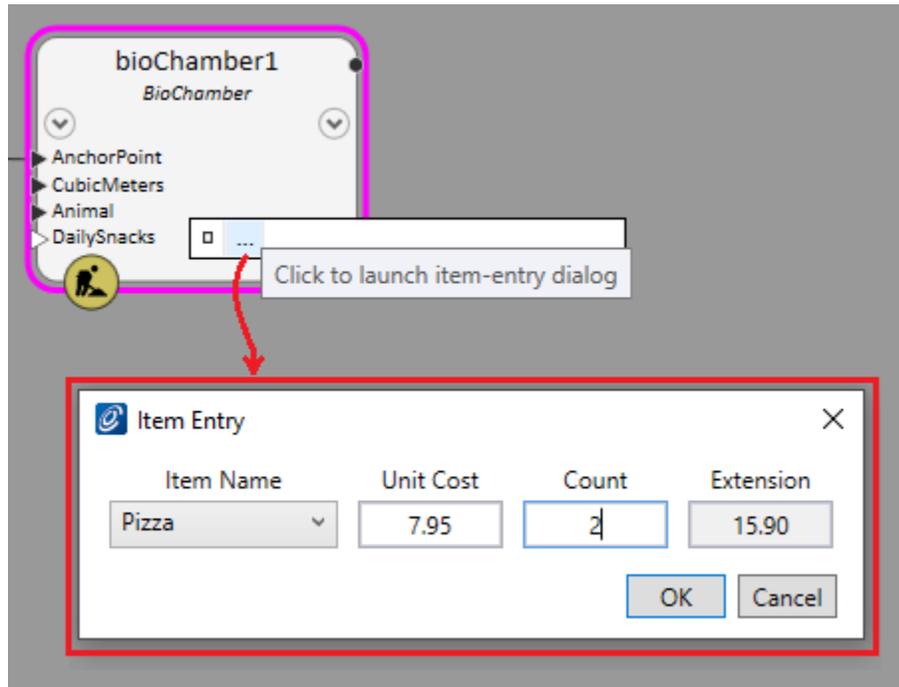
    internal IEnumerable<string> DailySnackChoices()
    {
        string animal = GetPropertyValue("Animal", "");

        // Maybe in the future we'll introduce a switch statement here, e.g.:
        //
        //     switch(animal)
        //     {
        //         case "Bat":      // Return a collection of snacks that are appropriate for bats.
        //         case "Chicken": // Return a collection of snacks that are appropriate for chickens.
        //         ...             // Etc.
        //     }
        //
        // But, for now, we'll just hard-code some silly snacks that all the animals can enjoy.

        return new string[] {"Chips", "Pizza", "Popcorn", "Toast"};
    }
}
```

## 2.2.5.2. The custom dialog

Let's peek ahead and take a look at the custom dialog we'll present to the user.



- **Who creates that dialog?**

Perhaps surprisingly, that's not relevant to this lesson!

- It could be a dialog you created, yourself, using WPF or some other UI technology (such as Windows Forms).
- It could be a dialog that originated from somewhere else in your organization.
- It could be a dialog obtained from a third party.

In any case, your GC add-in will treat that dialog as simply a "black box" without knowing or caring about its inner mechanisms.

Having said that, whichever dialog you choose must fulfill the following qualifications.

- Ideally, you should be able to populate that dialog's relevant field values (which will be edited by the user) from C#.
- You must be able to launch that dialog from C#, preferably as a modal dialog. ("Modal" means the user can't do anything else in GC or its host application until they close the dialog.)
- After you've launched that dialog, and after the user has filled in their data and closed the dialog...
- You must be able to determine the dialog's "close status" from C# -- e.g., whether the user clicked "OK" versus "Cancel".
- You must be able to read the dialog's relevant field values (which were filled in by the user) into corresponding C# variables.
- **In our case...**

We "stumbled across" a dialog that provides a generic way for the user to choose an item from a list and enter its unit cost and its count. The dialog then displays the extended cost (= unitCost \* count). We won't preserve the extended cost anywhere, since we can recalculate that value on the fly at any time.

To assign items and launch that dialog from C#, we do this:

### 2.2.5.3. Modifying our custom editor

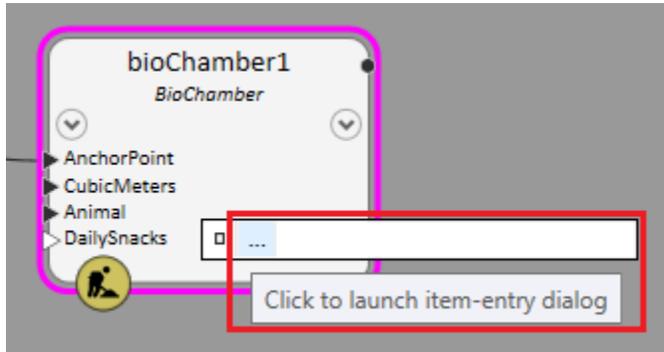
We must renovate our custom expression editor to serve its new purpose of launching a custom dialog.

- **BioChamberDailySnacksEditor.xaml**

Here's the revised version of that file, in its entirety:

```
<gcse:Editor x:Class="MyAddIn.BioChamberDailySnacksEditor"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:local="clr-namespace:MyAddIn"
  xmlns:gcse="urn:GC.ScriptEditor"
  mc:Ignorable="d"
  d:DesignHeight="450" d:DesignWidth="800">
  <DockPanel>
    <Button x:Name="_launchItemEntryDialogButton" DockPanel.Dock="Left" Width="18" Height="18"
      Click="_launchItemEntryDialogButton_Click"
      BorderBrush="LightGray" Style="{StaticResource {x:Static ToolBar.ButtonStyleKey}}"
      Content="..." Padding="0,-4,0,4" ToolTip="Click to launch item-entry dialog"
      />
    <TextBlock x:Name="_presentationTextBlock" Margin="2,0,0,0"/>
  </DockPanel>
</gcse:Editor>
```

You can see we're replaced the previous StackPanel with a DockPanel that contains a Button and a TextBlock. (The appearance specifications were determined empirically.) Our dialog now looks like this:



The Button is docked on the left and the TextBlock, which we named `_presentationTextBlock`, fills the remainder of the space.

- **BioChamberDailySnacksEditor.xaml.cs**

Here's the revised version of [that](#) file, in its entirety:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;

namespace MyAddIn
{
```

```

public partial class BioChamberDailySnacksEditor: Editor
{
    string _name;        // For ease of implementation, we maintain local copies
    int _count;         // of the three components encapsulated within our node
    double _unitCost;   // property, BioChamber.DailySnacks.

    public BioChamberDailySnacksEditor()
    {
        InitializeComponent();
    }

    protected override bool IsExpressionScriptRepresentableByThisEditor(Script script)
    {
        // We'll just always return true. In some implementations, it makes sense
        // to pre-examine the script to ensure it's something we can deal with.
        // But, in this case, we'll handle any invalid script when we actually try
        // to interpret the script, in the method RefreshFromChangedExpressionScript,
        // below.

        return true;
    }

    protected override Script ExpressionScriptAsRepresentedByThisEditor()
    {
        // Here we take the data components represented by this custom editor (i.e.,
        // snack name, snack count, and snack unit cost) and convert them to the script
        // expression that our node expects (i.e., a list: {'name', count, unitCost}).

        StringBuilder builder = new StringBuilder(); // Build the script, e.g.:

        builder.Append('{'); // {
        builder.Append(_name.ToQuotedScriptText()); // {'pizza'
        builder.Append(", "); // {'pizza',
        builder.Append(_count); // {'pizza', 2
        builder.Append(", "); // {'pizza', 2,
        builder.Append(_unitCost); // {'pizza', 2, 7.95
        builder.Append('}'); // {'pizza', 2, 7.95}

        string scriptText = builder.ToString();

        return scriptText.ToScript();
    }
}

```

```

protected override UIElement OnActivatedWhileSelected()
{
    // This method determines which (if any) of our custom editor's child
    // controls should get the focus whenever this editor becomes active, and
    // returns that child control (or default, which is the same as null).

    return _launchItemEntryDialogButton; // We want our launch button to get
                                        // the focus each time.
}

protected override void RefreshFromChangedExpressionScript(Script newScript, Script oldScript)
{
    // Here we take the script representation of this property's value, i.e.,
    // {'name', count, unitCost}, and use it to populate the local fields of
    // of this custom editor.

    _name      = ""; // Establish the default values we'll use
    _count     = 0;  // in case the given 'newScript' is invalid.
    _unitCost  = 0.0;

    if (TryGetValueOfScript(out GCList value, newScript)) // GCList is a C# structure that
                                                         // corresponds to a script list.
    {
        // A GCList holds a collection of IGCOBjects, where each IGCOBject holds the C#
        // representation of a script object. To retrieve the underlying C# object from
        // each IGCOBject instance, we must "unbox" the instance.
        //
        // Actually, in this case, we're going to TRY to unbox it: The method 'TryUnbox'
        // returns false if the IGCOBject cannot be converted to a C# object of the
        // desired type.

        if (value.Count > 0 && value[0].TryUnbox(out string s))
            _name = s;
        if (value.Count > 1 && value[1].TryUnbox(out int n))
            _count = n;
        if (value.Count > 2 && value[2].TryUnbox(out double d))
            _unitCost = d;
    }

    // Following is where we make an attractive display for the user.

    if (_name.Length == 0)
        _presentationTextBlock.Text = "";
}

```

```

        else
            _presentationTextBlock.Text = $"{_count} {_name}(s) @ {_unitCost} each";
    }

private void _launchItemEntryDialogButton_Click(object sender, RoutedEventArgs e)
{
    // Now we're going to create, populate, launch, and process the results from, our
    // custom dialog, which we've "obtained from somewhere". This is just an example:
    // The way you work with your own custom dialog may be very different.

    // Create an instance of the custom dialog.

    ItemEntryDialog dialog = new ItemEntryDialog();

    // Before we show the custom dialog to the user, we'll pre-populate its field values.
    //
    // First we need to query the current BioChamber node to get the choices of daily
    // snacks. The following statement may look weird, but it's the standard way of
    // getting the current node instance from within a custom expression editor.

    BioChamberNode bioChamber = (BioChamberNode) ParentExpressionEditorMeaningOfThis;

    // Now we assign the custom dialog's field values from our values that are based
    // on the underlying node property (BioChamber.DailySnacks). This particular custom
    // dialog gives us nice C# properties that we can simply assign values to.

    dialog.ItemChoices = bioChamber.DailySnackChoices();

    dialog.ItemName    = _name;
    dialog.Count       = _count;
    dialog.UnitCost    = _unitCost;

    // Now we can show the dialog to the user (modally), and wait for them to fill or
    // edit the fields before they close the dialog.

    bool? b = dialog.ShowDialog();

    if (b.GetValueOrDefault()) // i.e., if the user clicked OK rather than Cancel.
    {
        // Retrieve the field values from the dialog...

        _name    = dialog.ItemName;
    }
}

```

```

        _count    = dialog.Count;
        _unitCost = dialog.UnitCost;

        // ...then notify GC that this expression has changed. In turn, GC will call
        // the method ExpressionScriptAsRepresentedByThisEditor(), allowing us to
        // formulate the actual script expression that will be passed to our node.

        CommitChangesMadeToExpressionScript();
    }
}
}
}
}

```

(It's intended that the code plus its comments are self-explanatory.)

One interesting place is this passage:

```

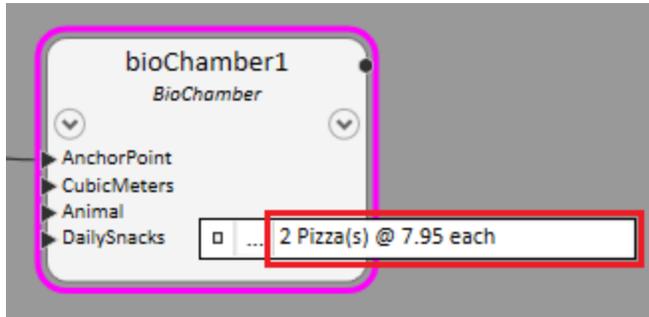
protected override void RefreshFromChangedExpressionScript(Script newScript, Script oldScript)
{
    :

    // Following is where we make an attractive display for the user.

    if (_name.Length == 0)
        _presentationTextBlock.Text = "";
    else
        _presentationTextBlock.Text = $"{_count} {_name}(s) @ {_unitCost} each";
}

```

Recall that most of the space of our custom expression is taken up by a TextBlock named `_presentationTextBlock`. Here, we're using that space to let the user see the property's value without having to launch the dialog:

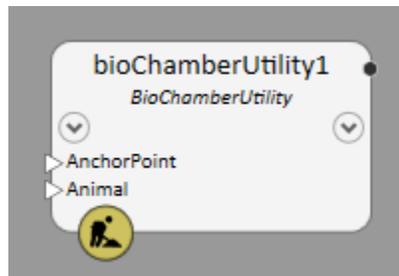


(Of course, the user will still need to launch the dialog if they want to edit those values.)

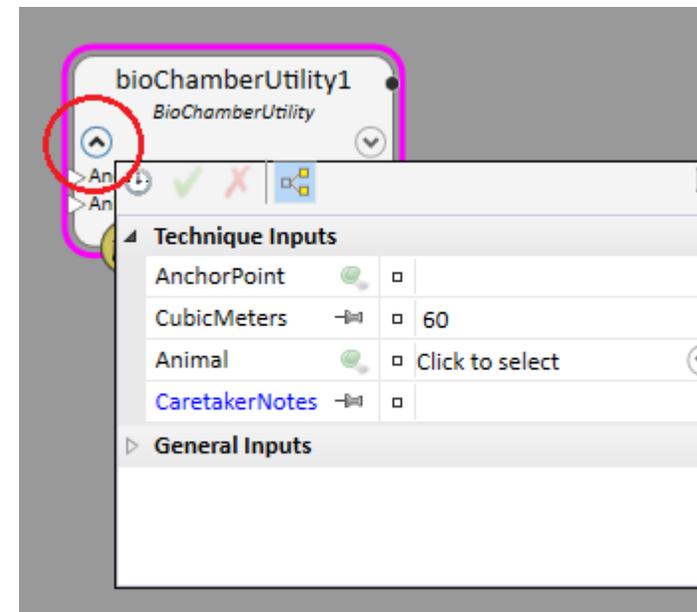
## 2.3. Node port pinning

When the user drops a new node onto GC's graph or changes the active technique of an existing node, certain of the inputs -- which manifest as input "ports" on the node image -- are already pinned.

1. In this example, the user has just dropped a BioChamberUtility node onto the graph. Initially, only the input ports 'AnchorPoint' and 'Animal' are pinned.

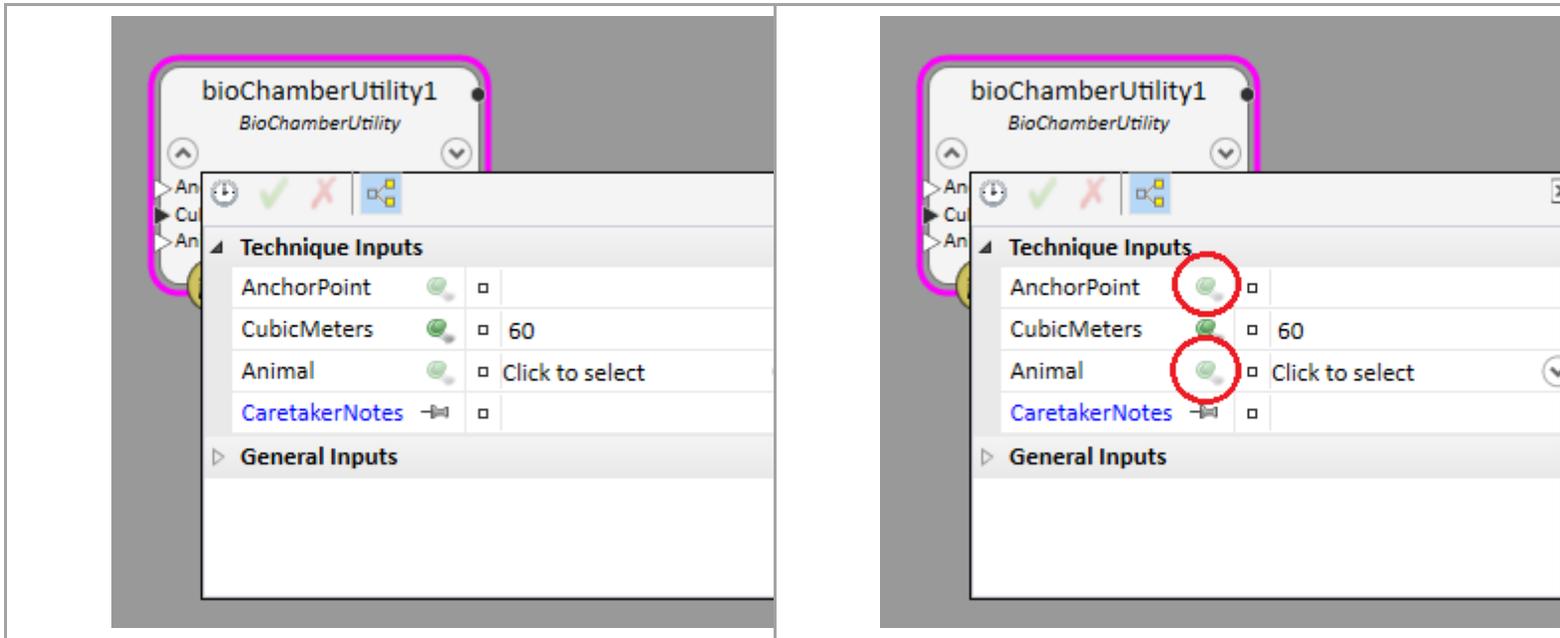


2. The user can reveal all the possible inputs by opening the "input ports" dropdown. Here we see that the technique's other two inputs, 'CubicMeters' and 'CaretakerNotes', are initially unpinned.



3. Of course, the user can manually pin any input(s) they wish.

4. Exception: The user can't change the pin state of required inputs that have no initial values. Those inputs are always pinned.



- **Which ports are initially pinned?**

By default, all required inputs that don't have initial values are initially pinned, while required inputs having initial values as well as optional inputs are initially unpinned:

	Is required input	Is optional input
<b>Has an initial value</b>	unpinned	unpinned
<b>Doesn't have an initial value</b>	pinned	unpinned

- **Overriding the default behavior**

When you write a new node type in C#, you can override the default behavior. Specifically, you can specify that a particular technique input should be initially pinned when it otherwise wouldn't be.

- For an element-based node, you add the attribute [GCInitiallyPinned] to that particular technique parameter. For example:

```
[GCInitialValue(60), GCInitiallyPinned] double CubicMeters,
```

- For a utility node, you call the method SetIsInitiallyPinned on the object that's returned from your call to technique.AddParameter for that particular parameter. For example:

```
technique.AddParameter(environment, nameof(CubicMeters), typeof(double), "60",  
    Ls.Literal("Interior volume of this bio-chamber."))  
    .SetIsInitiallyPinned(PropertyDirections.Input)
```

(A general discussion about defining technique parameters can be found [here](#) for element-based nodes and [here](#) for utility nodes.)

## 3. Element-Based Nodes

## 3.1. CopyTransform / MirrorCopy techniques

Question: I've created a custom element-based node, and, in general, it seems to work correctly. However, if the user selects either of the techniques `CopyTransformGeometricContents` or `MirrorCopyGeometricContentsAboutPlane`, it doesn't work: The technique fails, or nothing seems to happen. How can I make my custom node work correctly with those two techniques?

(For brevity, for the rest of this page I'll refer to both of the techniques "`CopyTransformGeometricContents`" and "`MirrorCopyGeometricContentsAboutPlane`" as the "Copy" techniques.)

Here's how to revise a custom node class to work with both of those Copy techniques:

1. If it doesn't already, your custom node class must derive from GC's class, `GeometricNode`, either directly or indirectly. (It's not sufficient to derive merely from GC's `ElementBasedNode` class.)

`GeometricNode` (which is, itself, derived directly from `ElementBasedNode`) provides support for certain geometric operations, including both of those Copy technique methods.

2. Add the following method to your class. This is the "workhorse" method that's called by both of those Copy technique methods.

```
public override NodeUpdateResult TransformContents(NodeUpdateContext updateContext,
                                                  ElementBasedNode nodeToCopy,
                                                  DTransform3d transformToUse,
                                                  bool reverseDirection)
{
    // You'll eventually add more code here...

    return NodeUpdateResult.Success;
}
```

Although this method isn't a technique method, itself (because it doesn't have either of the attributes `[Technique]` or `[DefaultTechnique]`), you implement it as though it is a technical method. That is, you'll use the given parameter values to create or revise an `Element` object, call `SetElement` to store that object on your node instance, then return an appropriate `NodeUpdateResult`.

The parameters are:

NodeUpdateContext updateContext	Generally, you can ignore this value. Its value is the same as that which was given to the calling Copy technique method.
ElementBasedNode nodeToCopy	This is the value of the 'NodeToCopy' or 'NodeToMirrorCopy' input the user gave to the calling Copy technique.
DTransform3d transformToUse	You'll apply this transform to the relevant geometry of 'nodeToCopy' to get the geometry you'll need to create or update your node's element.
bool reverseDirection	Generally, you can ignore this value; it's relevant only under rare circumstances. This is the value of the 'ReverseDirection' input, if any, the user gave to the calling Copy technique.
NodeUpdateResult (result)	Whatever value you return from your method will be value returned by the calling Copy technique method.

The rest of that method's implementation is up to you. Again, treat it just as you would any technique method.

Here's an example of [the SimpleLine class](#) that's part of GC's sample solution (which is installed with GC):

```
public override NodeUpdateResult TransformContents(NodeUpdateContext updateContext,
                                                ElementBasedNode nodeToCopy,
                                                DTransform3d transformToUse,
                                                bool reverseDirection)
{
    if (!(nodeToCopy is SimpleLineNode lineToCopy))
        return NodeUpdateResult.TechniqueGenericFailure; // This shouldn't ever happen; the caller should
                                                         // have already checked for this condition.

    IPointNode startPointToCopy = nodeToCopy.GetProperty<IPointNode>("StartPoint");
    IPointNode endPointToCopy = nodeToCopy.GetProperty<IPointNode>("EndPoint");

    DPoint3d startPtToCopy = startPointToCopy.GetDPoint3d();
    DPoint3d endPtToCopy = endPointToCopy .GetDPoint3d();

    DPoint3d startPt = transformToUse * startPtToCopy;
    DPoint3d endPt = transformToUse * endPtToCopy;
}
```

```

// The following code is the same as that which appears in SimpleLineNode's technique method, ByPoints.

DPoint3d uorStartPt = NativeDgnTools.FromMUToUOR(startPt);
DPoint3d uorEndPt   = NativeDgnTools.FromMUToUOR(endPt);

LineElement lineElement = new LineElement(GCDgnModel().DgnModel(), TemplateElement(),
                                          new DSegment3d(uorStartPt, uorEndPt));

SetElement(lineElement);

return NodeUpdateResult.Success;
}

```

As noted, about half of that method's code is copied from SimpleLine's technique method, ByPoints. You could take advantage of that situation by extracting that common code into its method, then revising both ByPoints and TransformContents to call that new method. (This is a common programming technique known as refactoring.) Thereafter, if you decide to revise that code for any reason, you'll need to revise it in only one place, not two.

That refactoring (in SimpleLineNode.cs) would look something like this:

```

public NodeUpdateResult ByPoints
(
    NodeUpdateContext          updateContext,
    [GCDgnModelProvider, GCReplicatable] IPointNode StartPoint,
    [GCReplicatable] IPointNode EndPoint
)
{
    :

    DPoint3d startPt = StartPoint.GetDPoint3d();
    DPoint3d endPt   = EndPoint.GetDPoint3d();

    CreateLineElement(startPt, endPt);

    return NodeUpdateResult.Success;
}

public override NodeUpdateResult TransformContents(NodeUpdateContext updateContext,
                                                    ElementBasedNode nodeToCopy,
                                                    DTransform3d transformToUse,

```

```

        bool reverseDirection)
{
    :

    IPointNode startPointToCopy = nodeToCopy.GetProperty<IPointNode>("StartPoint");
    IPointNode endPointToCopy = nodeToCopy.GetProperty<IPointNode>("EndPoint");

    DPoint3d startPtToCopy = startPointToCopy.GetDPoint3d();
    DPoint3d endPtToCopy = endPointToCopy.GetDPoint3d();

    DPoint3d startPt = transformToUse * startPtToCopy;
    DPoint3d endPt = transformToUse * endPtToCopy;

    CreateLineElement(startPt, endPt);

    return NodeUpdateResult.Success;
}

void CreateLineElement(DPoint3d startPt, DPoint3d endPt)
{
    // A subroutine of both ByPoints and TransformContents.
    // (This code was originally in the technique method, ByPoints.)

    DPoint3d uorStartPt = NativeDgnTools.FromMUToUOR(startPt);
    DPoint3d uorEndPt = NativeDgnTools.FromMUToUOR(endPt);
    LineElement lineElement = new LineElement(GCDgnModel().DgnModel(), TemplateElement(),
        new DSegment3d(uorStartPt, uorEndPt));
    SetElement(lineElement);
}

```

## 3.2. Getting/setting node properties

In your custom element-based node class, it's sometimes necessary to directly reference the node properties of that class (or another class).

For example, consider this technique method:

```
[GCTechnique] // or [GCDefaultTechnique]
public NodeUpdateResult ByPoints
(
    NodeUpdateContext updateContext,
    [GCDgnModelProvider, GCReplicatable] IPointNode StartPoint, // input property
    [GCReplicatable] IPointNode EndPoint, // input property
    [GCOut] ref double Length, // output property
    [GCOut] ref double MidLength // output property
)
{
    :
}
```

Of course, you don't need any help accessing this node's node properties StartPoint, EndPoint, Length, and/or MidLength, since they're being given directly to, or being assigned directly within, this technique method, itself!

But, what if you want to access those properties from within another method, or even from another class?

```
void SomeOtherMethodWithinYourClass(...)
{
    :
    IPointNode startPoint = <somehow get the value of the node property, StartPoint>;
    :
    double length = someOtherNode.<somehow get the value of the other node's node property, Length>;
    :
}
```

There are several ways to do that, as described in the following subpages.

- [Way 1: Call GetPropertyValue / SetPropertyValue](#)
- [Way 2: C# class properties](#)
- [Way 3: Performance improvement](#)

### 3.2.1. Way 1: Call GetPropertyValue / SetPropertyValue

Your node class's base type, `ElementBasedNode`, provides the following methods, which you can call:

- **GetPropertyValue**
- `public T GetPropertyValue<T>(string propertyName, bool enablePropertyCalculatorIfThereIsOne=true);`

This method takes a property name and returns that property's value. For example:

```
IPointNode startPoint = GetPropertyValue<IPointNode>("StartPoint");  
:  
double length = someOtherNode.GetPropertyValue<double>("Length");
```

Note that, since this is a generic method, you must explicitly specify its resultant type. It's your responsibility to ensure that the type aligns with that property's actual declared type (in whichever technique methods declare that property as a parameter).

If the property's value has never been assigned (e.g. the technique method that assigns that property has never been called), this method returns `default(T)`, that is, the default value of the specified type. (For `IPointNode`, which is a reference type, that value would be `null`; for `double`, which is a value type, that value would be `0.0`.)

In the vast majority of cases, you can omit the optional parameter `'enablePropertyCalculatorIfThereIsOne'`, letting it have its default value of `'true'`. Firstly, most properties don't have associated calculators, anyway. Secondly, rarely, you would ever not want to get a property's most current value (as the result of its latest calculation). Property calculators are discussed in more depth [here](#).

- `public T GetPropertyValue<T>(string propertyName, T defaultNativeValue,`

```
bool enablePropertyCalculatorIfThereIsOne=true);
```

This method behaves precisely the same as the preceding method, except that it takes an explicit default result that is returned if the property has never been assigned. For example:

```
double length = someOtherNode.GetPropertyvalue("Length", 100.0);
```

Note that you don't need to explicitly define the generic type in this case, since the C# compiler can determine that based on the type of your given default value. (Again, it's your responsibility to ensure that the type aligns with that property's actual declared type.)

- **SetPropertyValue**

- ```
public void SetPropertyValue<T>(string propertyName, T value,  
                               DependeeReevaluationScope effectOnParentNode  
                               =DependeeReevaluationScope.Dependents,  
                               bool forciblySetInputExpression=false);
```

This method is less commonly used since usually the user provides the inputs to your node. But, it can occasionally be useful to set a property's value programmatically. Here's how to do that, for example:

```
IPointNode someContrivedStartPoint = <whatever>;  
SetPropertyValue("StartPoint", someContrivedStartPoint);  
:  
someOtherNode.SetPropertyValue("Length", 0.0);
```

Note that you don't need to explicitly define the generic type in this case, since the C# compiler can determine that based on the type of your given default value. (Again, it's your responsibility to ensure that the type aligns with that property's actual declared type.)

You should omit the optional parameters 'effectOnParentNode' and 'forciblySetInputExpression', letting them have their default values. Those parameters are for internal use within GC, itself.

### 3.2.2. Way 2: C# class properties

This is a variant of [Way 1](#). If you find yourself getting and/or setting the same node properties multiple times, it may be helpful to implement "helper" C# properties in your class. These simply wrap the calls to `GetProperty` or `SetProperty` in a more convenient form. For example:

```
public IPointNode StartPoint
{
    get => GetPropertyValue<IPointNode>("StartPoint");
    set => SetProperty("StartPoint", value); // This 'set' accessor is optional.
}

public double Length
{
    get => GetPropertyValue<double>("Length");
    set => SetProperty("Length", value); // This 'set' accessor is optional.
}
```

As noted, the 'set' accessors are optional and should be used only if you intend to assign values programmatically to the corresponding node properties.

Now the preceding examples become:

```
IPointNode startPoint = StartPoint;
:
double length = Length;

StartPoint = <whatever>;
:
Length = 0.0;
```

### 3.2.3. Way 3: Performance improvement

Internally, node properties are stored in dictionaries so that GC can find any property very quickly, given its name.

However, that lookup time is still greater than zero, and you may wish to optimize your custom node's performance by retaining the result of each node property's lookup-by-name. In other words, no matter how many times you access a particular node property, its name lookup will occur only once. (This is the technique used in GC's node classes.)

- **Getting and managing the shadow property**

The result of a script-property lookup-by-name is something called a "shadow property"; it's an instance of a GC-defined class named `ShadowProperty`. For example:

```
ShadowProperty startPointShadowProperty = Shadow.ShadowProperty("StartPoint");
```

The preceding code references a C# property named `Shadow`, which is defined on the base class, `ElementBasedNode`. (For more information, see the documentation page, [Shadow classes](#).) That `Shadow` class, in turn, includes a method named `ShadowProperty`, which takes a property name and returns the corresponding shadow property.

The idea is this: For each of your node class's node properties that want to access the value, you'll get the corresponding shadow property only once, then reuse that shadow property for all subsequent value accesses.

The code will look like this, for example:

```
ShadowProperty _startPointShadowProperty; // Field whose initial value is null, by default.

ShadowProperty StartPointShadowProperty // C# property that looks up the shadow property only once.
{
    get
    {
        if (_startPointShadowProperty is null)
            _startPointShadowProperty = Shadow.ShadowProperty("StartPoint");
        return _startPointShadowProperty;
    }
}
```

You may wonder why you don't just initialize the field directly, e.g.:

```
ShadowProperty _startPointShadowProperty = Shadow.ShadowProperty("StartPoint"); // Don't do this!
```

The reason has to do with the timing of initialization. In some cases, when your class is initialized (which includes field values being assigned), the shadow properties aren't present, yet. So, instead, we defer that initial lookup until the first time the shadow property is needed.

- **How to use the shadow property**

The base class, `ElementBasedNode`, provides the following methods.

As you can see, these methods mimic the methods described in the previous section, except that their first parameters are shadow properties rather than property names.

- `public T GetPropertyValue<T>(ShadowPropertyBase shadowProperty, bool enablePropertyCalculatorIfThereIsOne=true);`

(`ShadowPropertyBase` is the base class of `ShadowProperty`.)

This method takes a shadow property and returns that property's value. For example:

```
IPointNode startPoint = GetPropertyValue<IPointNode>(StartPointShadowProperty);
```

The following text is copied from the "property name" version of this method, in the previous section:

Note that, since this is a generic method, you must explicitly specify its resultant type. It's your responsibility to ensure that the type aligns with that property's actual declared type (in whichever technique methods declare that property as a parameter).

If the property's value has never been assigned (e.g. the technique method that assigns that property has never been called), this method returns `default(T)`, that is, the default value of the specified type. (For `IPointNode`, which is a reference type, that value would be null; for `double`, which is a value type, that value would be 0.0.)

In the vast majority of cases, you can omit the optional parameter 'enablePropertyCalculatorIfThereIsOne', letting it have its default value of 'true'. Firstly, most properties don't have associated calculators, anyway. Secondly, rarely, you would ever not want to get a property's most current value (as the result of its latest calculation). Property calculators are discussed in more depth [here](#).

- `public T GetPropertyValue<T>(ShadowPropertyBase shadowProperty, T defaultNativeValue, bool enablePropertyCalculatorIfThereIsOne=true);`

(ShadowPropertyBase is the base class of ShadowProperty.)

This method behaves precisely the same as the preceding method, except that it takes an explicit default result that is returned if the property has never been assigned. For example:

```
double length = GetPropertyValue(LengthShadowProperty, 100.0);
```

The following text is copied from the "property name" version of this method, in the previous section:

Note that you don't need to explicitly define the generic type in this case, since the C# compiler can figure that out based on the type of your given default value. (Again, it's your responsibility to ensure that the type aligns with that property's actual declared type.)

- `public void SetPropertyValue<T>(ShadowPropertyBase shadowProperty, T value);`

(ShadowPropertyBase is the base class of ShadowProperty.)

This method is less commonly used, since the common way of assigning a property is through the user's inputs to your node. But, it can sometimes be useful to set a property's value programmatically. For example:

```
SetPropertyValue(LengthShadowProperty, 0.0);
```

The following text is copied from the "property name" version of this method, in the previous section:

Note that you don't need to explicitly define the generic type in this case, since the C# compiler can figure that out based on the type of your given default value. (Again, it's your responsibility to ensure that the type aligns with that property's actual declared type.)

- **Tying it all together**

Now you're ready to define a node property the way that GC does it within its node types. It's best explained by a full example:

```
// ===== StartPoint node property =====
ShadowProperty _startPointShadowProperty; // Assigned on first use

ShadowProperty StartPointShadowProperty
{
    get
    {
        if (_startPointShadowProperty is null)
            _startPointShadowProperty = Shadow.ShadowProperty("StartPoint");
        return _startPointShadowProperty;
    }
}

public IPointNode StartPoint
{
    get => GetPropertyValue<IPointNode>(StartPointShadowProperty);
    set => SetPropertyValue(StartPointShadowProperty, value); // This 'set' accessor is optional.
}

// ===== Length node property =====
ShadowProperty _lengthShadowProperty; // Assigned on first use.

ShadowProperty LengthShadowProperty
{
    get
    {
        if (_lengthShadowProperty is null)
            _lengthShadowProperty = Shadow.ShadowProperty("Length");
        return _lengthShadowProperty;
    }
}
```

```
public double Length
{
    get => GetPropertyValue<double>(LengthShadowProperty);
    set => SetPropertyValue(LengthShadowProperty, value); // This 'set' accessor is optional.
}
```

### 3.3. Output-only properties

In most cases, a node's output-only node properties are like its input properties in that they're based on the current user-selected technique. For example, GC's Plane node has an output property, `ShearAngle`, that's available only when the user has selected the technique, `ByThreePoints`.

You define that kind of technique-specific output property by adding a corresponding parameter to your technique method. For example, consider this technique method:

```
[GCTechnique] // or [GCDefaultTechnique]
public NodeUpdateResult ByPoints
(
    NodeUpdateContext updateContext,
    [GCDgnModelProvider, GCReplicatable] IPointNode StartPoint,
    [GCReplicatable] IPointNode EndPoint
)
{
    // Do whatever this technique does.

    return NodeUpdateResult.Success;
}
```

In a technique method -- that is, a method having either attribute `[GCTechnique]` or `[GCDefaultTechnique]` -- any parameters after the first one will become properties on any runtime instance of that node. (This happens because every node class has a corresponding [shadow class](#).)

By default, those properties will be input properties. In the method above, the parameters `StartPoint` and `EndPoint` will become input properties.

(The type of each input property doesn't matter. In this example, they both happen to be of the type `'IPointNode'`.)

To define a parameter that will become an output property, that parameter must include (1) the attribute `'[GCOut]'` and (2) the C# keyword `'ref'`. The following example shows the same method as above, but now with two output properties named `Length` and `HalfLength`.

```
[GCTechnique] // or [GCDefaultTechnique]
public NodeUpdateResult ByPoints
(
```

```
NodeUpdateContext updateContext,  
[GCDgnModelProvider, GCReplicatable] IPointNode StartPoint,  
[GCReplicatable] IPointNode EndPoint,  
[GCOut] ref double Length,  
[GCOut] ref double HalfLength  
)  
{  
    // Do whatever this technique does.  
  
    return NodeUpdateResult.Success;  
}
```

(The type of each output property doesn't matter. In this example, they both happen to be of the type 'double'.)

Next, within your technique method, itself, you'll need to calculate and assign the values to those parameters. For example:

```
)  
{  
    // Do whatever this technique does. Then:  
  
    Length = startPt.Distance(endPt);  
    HalfLength = Length / 2;  
  
    return NodeUpdateResult.Success;  
}
```

Strictly speaking, you don't need to assign values to those output parameters. If you don't, their values will be zero, null, or whatever corresponds to a "zero" value for their corresponding types.

## 3.4. Output-only properties that are always valid

The previous page, [Output-only properties](#), describes how to define a node's output-only properties that are valid when the user has selected a specific technique.

But, what if you want your node to have an output property that's always valid, regardless of which technique the user has selected? An example is GC's Circle node, whose output property, Radius, always provides a valid value regardless of the currently selected technique.

- [Way 1: Add that property to every technique](#)
- [Way 2: Property calculators](#)

-

### 3.4.1. Way 1: Add that property to every technique

That is, make sure every technique defines that property's corresponding parameter, as either an input or an output

This is probably the best way for the common case in which a node class has only a few technique methods. It doesn't require the significant overhead of a calculator method (which is described on the next page, [Way 2](#)).

You examine each of your node technique methods. If the method does not already have a parameter (input or output) that corresponds to the output property, add that parameter as an output parameter.

For example, suppose you want your node class to include an always-valid output property named `WidgetCount`, of the type `'int'`. Initially, your node class has these three technique methods:

|                                                                                                                                                           |                                                                                                                         |                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| <pre>[GCDefaultTechnique] public NodeUpdateTechnique Technique1 {     NodeUpdateContext updateContext,     :     int WidgetCount,     : } {     : }</pre> | <pre>[GCTechnique] public NodeUpdateTechnique Technique2 {     NodeUpdateContext updateContext,     : } {     : }</pre> | <pre>[GCTechnique] public NodeUpdateTechnique Technique3 {     NodeUpdateContext updateContext,     : } {     : }</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|

You can see that the output property you want, `WidgetCount`, is already accounted for in `Technique1`: It's one of the input parameters. (Of course, every node input property can also serve as an output property.)

However, Technique2 and Technique3 don't include any parameters corresponding to WidgetCount. So, if you want WidgetCount to be a valid output property in all circumstances, you'll need to add it as an output parameter to those techniques:

|                                                                                                                                                           |                                                                                                                                                                                                   |                                                                                                                                                                                                   |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>[GCDefaultTechnique] public NodeUpdateTechnique Technique1 {     NodeUpdateContext updateContext,     :     int WidgetCount,     : } {     : }</pre> | <pre>[GCTechnique] public NodeUpdateTechnique Technique2 {     NodeUpdateContext updateContext,     :     [GCOut] ref int WidgetCount } {     :     WidgetCount = &lt;whatever&gt;;     : }</pre> | <pre>[GCTechnique] public NodeUpdateTechnique Technique3 {     NodeUpdateContext updateContext,     :     [GCOut] ref int WidgetCount } {     :     WidgetCount = &lt;whatever&gt;;     : }</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

This achieves the goal.

Of course, each technique method must have its way of calculating the value of WidgetCount. That may involve reading the values of other properties on your node, as described [here](#).

## 3.4.2. Way 2: Property calculators

This way is significantly more complicated than [Way 1](#), but it's appropriate if/when:

- Your node class has lots of technique methods, such that it's impractical to add those output properties to everyone.
- You expect your node class will be getting more technique methods in the future, and you don't want the author of those methods (either yourself or someone else) to have to remember to add those output properties in each case.
- In each technique method, you're using the same calculations to assign values to those output parameters. For maintenance purposes, it would be better if your node class contained only one occurrence of that calculation code.
- You feel it's more elegant from a software design point of view: You'll be completely separating the functionality of output property calculation from the functionality of technique methods.

◆ The remainder of this page refers to calculated properties having names like `WidgetCount` and `TesseractAngle`. These are simply placeholder names for actually calculated properties you might create yourself.

Also included is a "real life" calculated property named `Length`. [Here](#) is the full listing of a class that implements `Length` as a calculated property, using the techniques described below.

- **Calculator methods**

For each always-valid output property, you'll need to implement a property calculator method, which:

- Is a standard C# class method.
- Has this signature:

```
public void <MethodName>(DependeeReevaluationScope effectOnParentNode)
```

The method name can be anything, though you'll probably want to choose something meaningful, like "WidgetCountPropertyCalculator".

- Has the attribute '[GCPropertyCalculator(...)]', which takes the name (as a string) of the output property to which that calculator method applies. For example:

```
[GCPropertyCalculator("WidgetCount")]
```

Or, if you've defined WidgetCount as a C# class property (as described [here](#)):

```
[GCPropertyCalculator(nameof(WidgetCount))]
```

Putting it all together, we have:

```
[GCPropertyCalculator("WidgetCount")]
public void WidgetCountPropertyCalculator(DependeeReevaluationScope effectOnParentNode)
{
    : // Based on the current state of this node, calculate the value to be assigned to this
    : // property, 'WidgetCount'. (There's more information below.)
    :
    : SetPropertyValue("WidgetCount", value, effectOnParentNode);
}
```

It's your job to figure out how to calculate the value you want, based on the current state of your node. This may involve reading the values of other properties (as described [here](#)) and/or the values of other members of your class.

Perhaps the most common way to calculate the value is to base it on the node's current element. That code might look something like this:

```
[GCPropertyCalculator("Length")]
public void LengthCalculator(DependeeReevaluationScope effectOnParentNode)
{
    // This method calculates the value of this node's output property, Length.
    //
    // In this example, we're assuming that this node manifests as a line element in MicroStation.
```

```

// In other words, we're assuming that each of this node's technique methods creates a MicroStation
// line element (by calling the appropriate methods in MicroStation's own .NET API), then assigns
// that element to this node by calling this node's method, SetElement (which is inherited from
// the base class, ElementBasedNode).

// In this calculator method, we'll start by calling Element() to get the alleged line element
// that was created in the latest technique method that was called on this node. Element() may
// return null if no technique method has been called, yet, or if the latest technique method
// failed for some reason.

if (Element() is LineElement lineElement)
{
    // The following code demonstrates one way to get the length of a MicroStation line element.

    CurveVector curveVector = lineElement.GetCurveVector();

    curveVector.GetStartEnd(out DPoint3d uorStartPt, out DPoint3d uorEndPt);

    DPoint3d startPt = NativeDgnTools.FromUORToMU(uorStartPt); // Must convert both points from the
    DPoint3d endPt   = NativeDgnTools.FromUORToMU(uorEndPt);   // MicroStation API's standard units
  // (UORs) to the GC API's standard
  // units (master units).

    double length = startPt.Distance(endPt);

    // Finally, assign that value to this node's output property, 'Length'.

    SetPropertyValue("Length", length, effectOnParentNode);
}
}

```

If multiple properties are interrelated, you can use the same calculator method to calculate and assign all their values together. To do so, simply add more property names to the 'GCPropertyCalculator' attribute:

```

[GCPropertyCalculator("WidgetCount", "AerodynamicFlow", "TesseractAngle")]
public void WidgetCountEtcPropertyCalculator(DependeeReevaluationScope effectOnParentNode)
{
    : // Calculate the values of all three of these interrelated properties...
    :
    SetPropertyValue("WidgetCount", widgetCountValue, effectOnParentNode);
    SetPropertyValue("AerodynamicFlow", aerodynamicFlowValue, effectOnParentNode);
}

```

```
        SetPropertyValue("TesseractAngle", tesseractAngleValue, effectOnParentNode);
    }
```

- **Declaring output properties that don't otherwise exist**

When GC generates the underlying [shadow class](#) of a node class, it determines the node's properties based on all the parameters passed to all of that node's technique methods.

However, what if you want your node to have an always-valid output property that doesn't appear in any technique's parameter list?

The solution is to define a special technique method that exists only to define properties that otherwise don't appear anywhere. It's defined the same as any other technique method, including having the attribute [GCTechnique] and the same signature as any technique method. However, it also has the attribute [GCUnselectable], which means the user can never select that technique, nor even see any evidence of it in GC's UI.

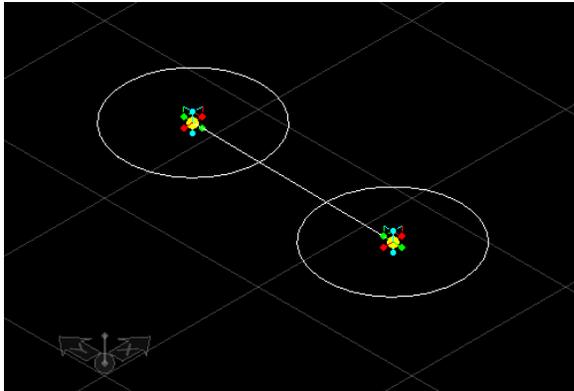
Here's an example that defines the properties WidgetCount and TesseractAngle, which we're presuming aren't given to any other technique methods in our class:

```
[GCTechnique, GCUnselectable]
public NodeUpdateResult DummyPropertyDefiningTechnique // This method name can be anything.
(
    NodeUpdateContext updateContext, // Required first parameter for all technique methods,
                                     // even this one.

    int WidgetCount, // Include all the properties that you want to be
    double TesseractAngle // valid for all techniques.
)
{
    return NodeUpdateResult.TechniqueGenericFailure; // This result can be anything, since this
   // method will never actually be called.
}
```

## 3.5. Constituent nodes

Constituent nodes provide a way to programmatically incorporate additional nodes into your node. For example, here is a Line node to which I've programmatically added a Circle node centered around each of the line's endpoints.



Those circles are added to the Line node as constituent nodes. Following is the general process. These steps all take place in the appropriate technique method of the node that will include the constituent nodes.

1. Start your technique method by letting it do whatever it normally would if constituent nodes were not involved.
2. Call this method, which prepares the node's constituent-nodes list to receive new values.

```
DeleteConstituentNodes(updateContext);
```

(The 'updateContext' is the same 'updateContext' that's being given to the current technique method.)

3. For each constituent node that you want to create and add to the current node:
  - a. Call the method 'CreateNodeThatWillBecomeAConstituentOfThis':

```
T constituentNode = CreateNodeThatWillBecomeAConstituentOfThis<T>();
```

where 'T' is the type of the constituent node. For example:

```
CircleNode circle = CreateNodeThatWillBecomeAConstituentOfThis<CircleNode>();
```

- b. Do whatever needs to be done to initialize that constituent node. Typically, that means calling the appropriate technique method on that node:

```
constituentNode.LatestUpdateResult = constituentNode.TechniqueMethod(updateContext, ...);
```

For example:

```
circle.LatestUpdateResult = circle.ByCenterRadius(updateContext, StartPoint, circleRadius, support, false);
```

As you can see, we assign the result of calling that technique method to the constituent node's property, LatestUpdateResult. This is important. Even if you initialize a constituent node in some other way (besides calling one of its technique methods), you should assign an explicit value to that property:

```
T constituentNode = CreateNodeThatWillBecomeAConstituentOfThis<T>()  
: // Initialize the constituent node in some way other than by calling a technique method.  
constituentNode.LatestUpdateResult = NodeUpdateResult.Success;
```

- c. Finally, call the method 'AddConstituentNode':

```
AddConstituentNode(constituentNode);
```

For example:

```
AddConstituentNode(circle);
```

[Here](#) is the full listing of the modified Line class that adds the two constituent nodes shown at the top of this page.

- **Indexible constituent nodes**

GC supports a way you can add constituent nodes such that they can be accessed by their index, similar to how replicated nodes are accessed.

But, there's a big rule you must follow: Just like replicated nodes, all of a node's indexible constituent nodes must be of the same type as that parent node.

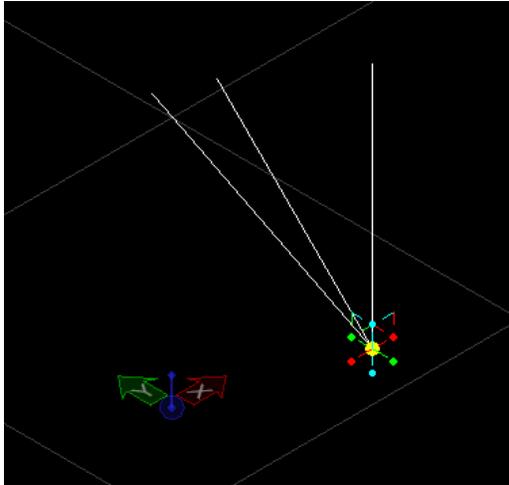
Beyond that, all you need to do is add this statement in your technique method, immediately after you clear the constituent nodes list:

```
DeleteConstituentNodes(updateContext);  
LetConstituentNodesBeDirectlyIndexible(); // New statement.
```

Here's an example that creates several indexible Line nodes within a parent Line node. This code would appear in one of the parent Line node's technique methods.

```
[GCTechnique]  
public NodeUpdateResult DemoIndexibleConstituents(NodeUpdateContext updateContext, IPointNode  
StartPoint)  
{  
    DPoint3d startPt = StartPoint.GetDPoint3d();  
    double x = startPt.X;  
    double y = startPt.Y;  
    double z = startPt.Z;  
  
    DeleteConstituentNodes(updateContext);  
    LetConstituentNodesBeDirectlyIndexible();  
  
    for (int i = 0; i < 3; ++i)  
    {  
        LineNode line = CreateNodeThatWillBecomeAConstituentOfThis<LineNode>();  
        DVector3d vector = DVector3d.FromXYZ(1, i + 1, 0);  
        vector.NormalizeInPlace();  
        line.LatestUpdateResult = line.ComputeFromStartDirectionLength(startPt, vector, 10);  
        AddConstituentNode(line);  
    }  
  
    return NodeUpdateResult.Success;  
}
```

The result of running that technique looks like this:



We can see that the indexible constituent nodes are treated just like replicated nodes. However, the technique method, itself, is fully responsible for managing that "replication".

## 3.6. Shadow classes

An element-based node class -- including any that you write, yourself -- doesn't include everything GC needs to create and manage nodes of that type. In particular, a node class does not include:

- Code that implements every technique-method parameter as a node property.
- Wrapper code such that the class's technique methods can be called from the script.
- Code that handles replication.
- Other support codes are required by GC's infrastructure.

If node-class authors had to write all that additional code, themselves, it would be an extremely tedious and error-prone process. However, fortunately, GC generates all that additional code automatically.

More precisely: Whenever GC starts, or whenever the user loads a new add-in, GC uses .NET reflection to examine all the included node classes. For each of those node classes, GC generates and compiles a corresponding "shadow" class that comprises all the additional code described above.

Thus, each node-class-and-shadow-class pair provides the complete definition of a node, from the standpoint of GC's infrastructure.

Incidentally: The purpose of the various [GC...] attributes that you use in your node class is to guide GC's shadow-class generator. In other words, each of those attributes affects the generated shadow code in some way.

It's only rarely that you'll ever need to access your node class's corresponding shadow class. If you ever do, it's provided via the property 'Shadow' on the base class, `ElementBasedNode`.

## 3.7. Technique attributes

GC uses attributes to define the characteristics of each technique on an element-based node. These attributes control how GC generates the node's implicit shadow class. For example:

```
using ...;
using ...;
:

namespace MyAddIn
{
    [GCNamespace("User")]
    [GCNodeTypePaletteCategory("My Add-In")]
    [GCNodeTypeIcon("Resources/BioChamberNode.png")]
    [GCSummary("A high-tech animal cage for long-term care.")]
    public class BioChamberNode: ElementBasedNode
    {
        :

        [GCDefaultTechnique]
        [GCSummary("Creates a bio-chamber from a given anchor point and other criteria.")]
        [GCParameter("AnchorPoint", "Anchor point of the bio-chamber.")]
        [GCParameter("CubicMeters", "Interior volume of this bio-chamber.")]
        [GCParameter("Animal", "The type of animal this bio-chamber is designed for.")]
        [GCParameter("CaretakerNotes", "A file that contains caretaking instructions.")]
        [GCParameter("IsAnimalCuddly", "Whether this type of animal is nice to cuddle with.")]
        public NodeUpdateResult Default
        (
            NodeUpdateContext                updateContext,
            [GCDgnModelProvider] IPointNode   AnchorPoint,
            [GCInitialValue(60)] double       CubicMeters,
            [EECCAttr(nameof(GetEECCForAnimal))] string   Animal,
            [GCOptional, GCInitiallyPinned,
             GCFileBrowser(fileFilterDescription:"Text files",
                           fileFilterMask:"*.txt")]
            string                             CaretakerNotes,
            [GCOut] ref bool                   IsAnimalCuddly
        )
        {
            // Do whatever this method does with the given inputs.
        }
    }
}
```

```

        IsAnimalCuddly = Animal == "Cat" || Animal == "Dog"; // Or however this method
   // calculates this value.

        return NodeUpdateResult.Success;
    }

    :

    } // class
} // namespace

```

(Note: The full class listing can be found [here](#) in the section, Code Samples.)

where:

- [GCNamespace("...")] specifies which scripting namespace that will contain this node type. "User" is the standard. If you omit this, your node's scripting namespace will be the same as its C# namespace; in this case, "MyAddIn".
- [GCNodeTypePaletteCategory("...")] specifies where this node type will appear within GC's Node Types panel; that is, which group it will belong to. You can make up any group name you want, though, of course, it should make sense to the user.
- [GCNodeTypeIcon("...")] specifies the custom image of this node type within the Node Types panel. It's a PNG file, 24 x 24 pixels, and is included in your Visual Studio project as a Resource (not Embedded Resource). The path given to GCNodeTypeIcon is the file's resource path relative to the location of your CSProj file, so that path uses forward slashes (/) instead of backslashes (\).
- [GCSummary("...")] provides the user with an overview description of this node type.
- [GCDefaultTechnique] or [GCTechnique] means the following C# method is to be treated as this node type's default technique or a non-default technique, respectively.
- [GCSummary("...")] provides a description of this technique.
- [GCParameter("...", "...")] describes one of this technique's inputs.

- [GCDgnModelProvider] indicates that this input will provide the DGN model (i.e., the MicroStation model, as opposed to the GC model) into which the graphics produced by this node will be written. In other words: Any graphics produced by this technique method will be put into the DGN model that this input is using.
- [GCReplicable] indicates that this input can be given a list of items, in which case this node instance will become replicated. (Replication is handled entirely by GC's infrastructure. Each node instance is responsible for handling only a single case.)
- [GCInitialValue(...)] defines the initial value of this input. (By default, node inputs are empty.)
- [EECCAttr(nameof(...))] defines a custom editor... or, more precisely, associates this input with a C# method that, when called, will produce and return a custom editor. This is all explained starting [here](#).
- [GCOptional] means this input is optional.
- [GCInitiallyPinned] means that, within GC's graph, this input will start out being "pinned" to the left side of the node, like most inputs are. By default, an optional input ([GCOptional]) or an input having a [GCInitialValue(...)] doesn't appear unless/until the user explicitly pins it.

More about node-port pinning can be found [here](#).

- [GCFileBrowser(...)] means this input will have a built-in file browser. Various configuration options are available in the form of optional parameters to this attribute's constructor.
- [GCOut] means this method parameter will serve as a technique-specific output rather than an input. The parameter, itself, must have the C# modifier 'ref' (not 'out').

## 4. Utility Nodes

## 4.1. Explicit framework

For an element-based node, GC, itself, provides most of the node's framework by generating a corresponding [shadow class](#). You sprinkle your element-based-node class with attributes, which govern the generation of that shadow class.

A utility node uses an entirely different architecture wherein GC doesn't generate anything. Your definition of a utility node class is its entire definition. Therefore, it's your responsibility to provide all the gritty details.

On the other hand, a utility node is conceptually simpler than an element-based node because it doesn't need to worry about things like replication and MicroStation element management.

[Here](#) is an example of a C# class that defines a utility node. It's intended to be reasonably self-explanatory, so I encourage you to study it.

## 4.2. Where are the technique attributes?

[This page](#) demonstrates how we use attributes to define the characteristics of a technique on an element-based node. For example:

```
[GCDefaultTechnique]
[GCSummary("Creates a bio-chamber from a given anchor point and other criteria.")]
[GCPParameter("AnchorPoint", "Anchor point of the bio-chamber.")]
[GCPParameter("CubicMeters", "Interior volume of this bio-chamber.")]
[GCPParameter("Animal", "The type of animal this bio-chamber is designed for.")]
[GCPParameter("CaretakerNotes", "A file that contains caretaking instructions.")]
[GCPParameter("IsAnimalCuddly", "Whether this type of animal is nice to cuddle with.")]
public NodeUpdateResult Default
(
    NodeUpdateContext                updateContext,
    [GCDgnModelProvider] IPointNode  AnchorPoint,
    [GCInitialValue(60)] double      CubicMeters,
    [EECCAttr(nameof(GetEECCForAnimal))] string  Animal,
    [GCOptional, GCInitiallyPinned,
     GCFileBrowser(fileFilterDescription:"Text files",
                    fileFilterMask:"*.txt")]
    string                            CaretakerNotes,
    [GCOut] ref bool                  IsAnimalCuddly
)
{
    // Do whatever this method does with the given inputs.

    IsAnimalCuddly = Animal == "Cat" || Animal == "Dog"; // Or however this method
   // calculates this value.

    return NodeUpdateResult.Success;
}
```

Now here's the same functionality as it would be expressed on a utility node:

```
static void AddAdditionalMembersToGCType(IGCEnvironment environment, GCType gcType,
   NativeNamespaceTranslator namespaceTranslator)
{
    UtilityNodeTechnique technique = gcType.AddDefaultNodeTechnique("Default", DefaultTechnique,
```

```

criteria."));
  Ls.Literal("Creates a bio-chamber from given
technique.AddParameter(environment, nameof(AnchorPoint), typeof(DPoint3d), "",
  Ls.Literal("Anchor point of this bio-chamber."));
technique.AddParameter(environment, nameof(CubicMeters), typeof(double), "60",
  Ls.Literal("Interior volume of this bio-chamber."))
  .SetIsInitiallyPinned(PropertyDirections.Input);
technique.AddParameter(environment, nameof(Animal), typeof(string), "",
  Ls.Literal("The type of animal this bio-chamber is designed for.))
  .SetGetEECC(nameof(Default), GetEECCForAnimal);
technique.AddParameter(environment, nameof(CaretakerNotes), typeof(string), "",
  Ls.Literal("A file that contains caretaking instructions."),
NodePortRole.TechniqueOptionalInput)
  .SetGetEECC(nameof(Default), GetEECCForCaretakerNodes));
technique.AddParameter(environment, nameof(IsAnimalCuddly), typeof(bool), "",
  Ls.Literal("Whether this type of animal is nice to cuddle with."),
NodePortRole.TechniqueOutputOnly);
}

```

(The section "Code Samples" provides the full listings of both of the preceding node-class files, [BioChamberElementBasedNode.cs](#) and [BioChamberUtilityNode.cs](#).)

where:

- `gcType.AddDefaultNodeTechnique` associates a static method on this class (e.g. `DefaultTechnique`) with a technique for this node (e.g., "Default"). (If we want to define a technique for this node that's not the default technique, we call `gcType.AddNodeTechnique` rather than `gcType.AddDefaultNodeTechnique`.)
- The technique's description (as it will be presented to the user) is specified in the form of `"Ls.Literal(...)"` passed to the call, `gcType.AddNodeTechnique` or `gcType.AddDefaultNodeTechnique`. ([Here](#) is a little more information about the type 'Ls'.)
- Each input and output to the technique is defined by the calling `technique.AddParameter(...)`. It takes these arguments:
- The value 'environment', which we simply pass along.

- The name of the input or output property. We can use C#'s "nameof(...)" operator because, elsewhere in this class, we've defined properties having those names. (See the full listing [here](#).)
- A string containing the initial value of this property in the form of a script expression. If you want the property to be empty (which is the usual case), pass "" (two consecutive quotation marks) — that is, an empty C# string.
- The property's description (as it will be presented to the user) is in the form of "Ls.Literal(...)".
- Optionally, a NodePortRole value governs whether the property is (for example) an optional or output property. If omitted, this value is treated as NodePortRole.TechniqueRequiredInput.
- Each call to technique.AddParameter(...) returns an object of the type UtilityNodeTechniqueParameterDefinition. Of course, we can then immediately call any of that object's methods, such as SetGetEECC or SetIsInitiallyPinned:

```
technique.AddParameter(...).SetGetEECC(...);  
technique.AddParameter(...).SetIsInitiallyPinned(...);
```

Furthermore, each of those "secondary" method calls, itself, returns that same instance of UtilityNodeTechniqueParameterDefinition, so those calls can be chained together:

```
technique.AddParameter(...).SetGetEECC(...).SetIsInitiallyPinned(...);
```

Here's what those "secondary" method calls do:

- SetGetEECC lets you establish a custom editor for that parameter, as we did with element-based nodes.

```
technique.AddParameter(...).SetGetEECC("Default", GetEECCForAnimal)
```

The first argument to SetGetEECC is the name of the technique method as it will be presented to the user. In this case, we're working on the default technique, which we've conveniently named 'Default'.

The second argument is the name of a static method that we've defined elsewhere in our node class. That method's signature and general form look like this:

```
static EECC GetEECC(INodeBase nodeBase) // Utility node version.
{
    return new EECC(...);
}
```

Note this is different from the "GetEECC" methods we used in element-based nodes, which look like this:

```
static EECC GetEECC() // Element-based node version.
{
    return new EECC(...);
}
```

In the utility-node version, the node instance we're working with is given directly as a parameter. Thus, we can slightly simplify our implementation. For example:

```
static EECC GetEECCForAnimal(INodeBase nodeBase) // Utility node version.
{
    return new EECC(getScriptChoices);

    IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
    {
        BioChamberUtilityNode bioChamber = (BioChamberUtilityNode) nodeBase;

        ScriptChoiceList result = bioChamber.GetGroupedAnimals();

        return result;
    }
}
```

Compared with the element-based-node version:

```
static EECC GetEECCForAnimal() // Element-based node version.
{
    return new EECC(getScriptChoices);
}
```

```

    IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
    {
        BioChamberElementBasedNode node = (BioChamberElementBasedNode)
parentExpressionEditor.MeaningOfThis;

        ScriptChoiceList result = node.GetGroupedAnimals();

        return result;
    }
}

```

- SetIsInitiallyPinned - Whether a technique input or output is "pinned" refers to its appearance on the node image within GC's graph view.

More about node-port pinning can be found [here](#).

- **What about a file browser?**

For element-based nodes, there is a single attribute, [[GCFileBrowser](#)], that can be applied to any technique method parameter such that the corresponding input will manifest to the user as a file browser.

Unfortunately, utility nodes don't (yet) have a single attribute or method call that establishes a file browser. However, it's (relatively) easy to establish one using another 'GetEECC' method.

Here is our definition of the parameter that will manifest to the user as a file browser:

```

technique.AddParameter(environment, nameof(CaretakerNotes), typeof(string), "",
    Ls.Literal("A file that contains caretaking instructions."),
    NodePortRole.TechniqueOptionalInput)
    .SetGetEECC(nameof(Default), GetEECCForCaretakerNodes));

```

And here's our definition (elsewhere in the same class, BioChamberUtilityNode) of the 'GetEECC' method:

```

static public EECC GetEECCForCaretakerNote(INodeBase nodeBase)
{

```

```
FileBrowserConfiguration configuration = FileBrowserConfiguration.Get(fileFilterDescription:"Text files",
  fileFilterMask:"*.txt");
return configuration.GetEECC(nodeBase);
}
```

GC's FileBrowserConfiguration class has a static 'Get' method that provides various configuration options in the form of optional parameters. Generally, those parameters are the same as those given to the [GCFileBrowser](#) attribute in an element-based node.

## 4.3. Custom UI

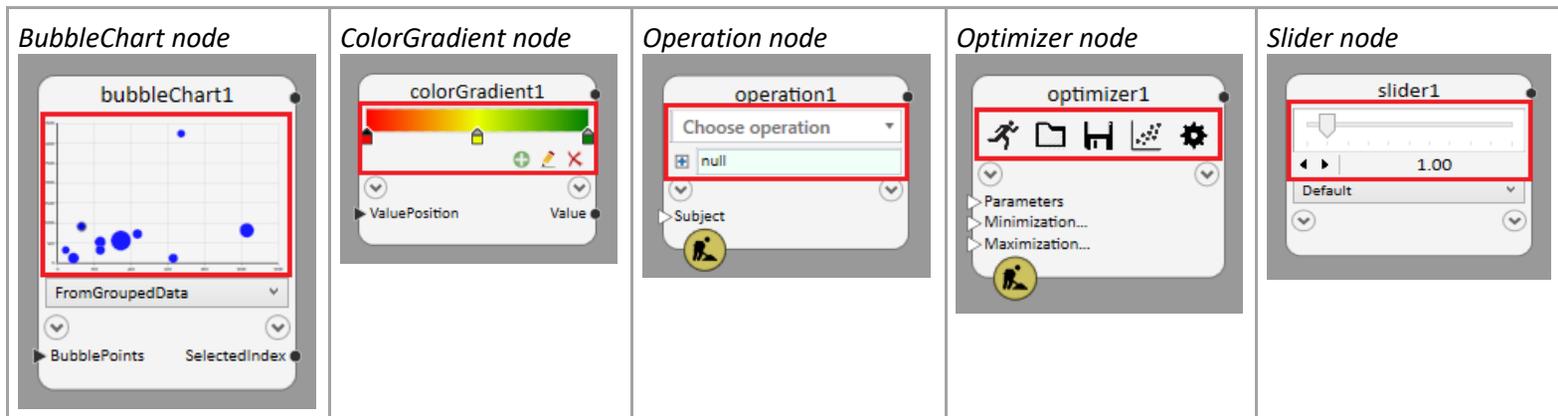
One of the most distinctive and valuable aspects of a utility node is that it can have a custom UI. In fact, in many cases, that custom UI is the primary motivation for the utility node to exist in the first place.

A utility node's custom UI must be written in WPF, so here we go again with this disclaimer:

◆ This topic assumes you're already familiar with writing WPF (Windows Presentation Foundation) user controls. It's far beyond the scope of this GC API documentation to teach that Microsoft technology.

- **Examples**

Following are examples of GC's built-in utility-node types. In each case, the red rectangle indicates the region occupied by the custom WPF UI.



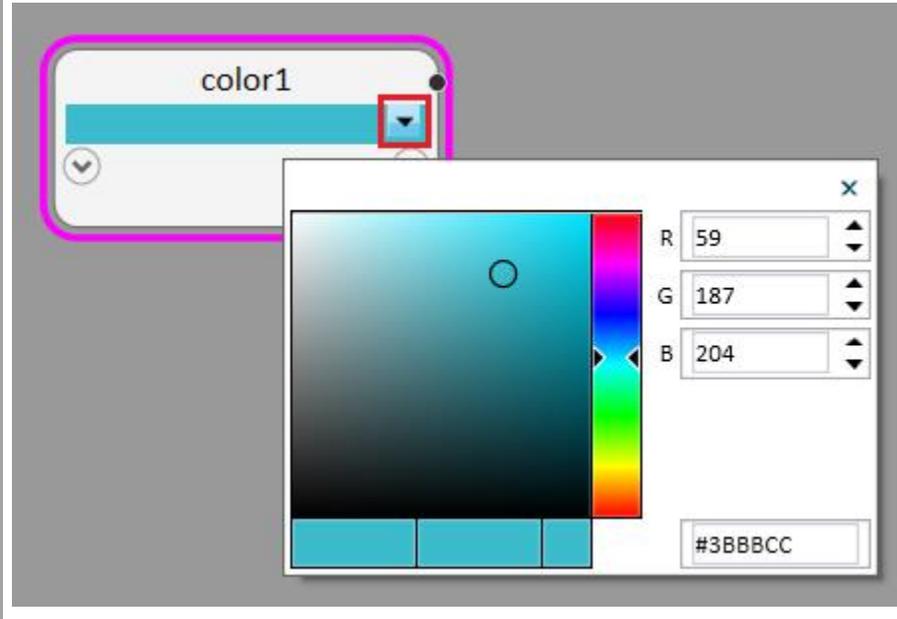
As with any WPF UserControl, a utility node's custom UI can include occurrences of popup windows, drop-downs, and third-party controls. Following are examples of custom UIs that include a popup, a dropdown, and a third-party control, respectively.

Optimizer node

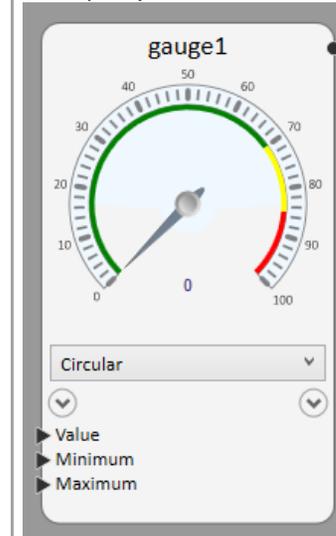
The image shows a software interface with an 'optimizer1' node and an open 'Optimization Settings' dialog box. The dialog box is divided into two sections: 'Genetic Algorithm' and 'Completion Criteria'. Each section contains several parameters with input fields and descriptive text. The 'Genetic Algorithm' section includes parameters like Random seed, Population size, and Crossover rate. The 'Completion Criteria' section includes Maximum generations, Maximum trials, and Non-improvement generations. At the bottom of the dialog are buttons for 'Restore Defaults', 'OK', and 'Cancel'.

| Parameter                    | Value | Description                  |
|------------------------------|-------|------------------------------|
| <b>Genetic Algorithm</b>     |       |                              |
| Random seed:                 | 5     | (Any positive integer.)      |
| Population size:             | 100   | (Typically, 50 to 500.)      |
| Elite population size:       | 10    | (Typically, 10 to 20.)       |
| Crossover points:            | 3     | (Typically, 2 to 10.)        |
| Mutation rate:               | 0.015 | (Typically, 0.001 to 0.030.) |
| Creeping rate:               | 0.001 | (Typically, 0.001 to 0.030.) |
| Creeping down rate:          | 0.5   | (Typically, 0.3 to 0.7.)     |
| Elite mate rate:             | 0.005 | (Typically, 0.001 to 0.020.) |
| Crossover rate:              | 0.95  | (Typically, 0.7 to 1.0.)     |
| Tournament fittest win rate: | 0.95  | (Typically, 0.9 to 1.0.)     |
| <b>Completion Criteria</b>   |       |                              |
| Maximum generations:         | 50    | (Typically, 500 to 2000.)    |
| Maximum trials:              | 5000  | (Typically, 10000 or more.)  |
| Non-improvement generations: | 5     | (Typically, 100 to 300.)     |

Color node



Gauge node. The gauge image is from Actipro™ WPF Controls, which is one of many third-party WPF control suites available online.



- **Getting started**

You start writing a utility node's custom UI by creating a WPF UserControl from within Visual Studio. You can name it anything you want, but we follow the naming convention "<NodeClassName>View". For example, for a node class named BioChamberNode, you would name your corresponding WPF UserControl "BioChamberNodeView".

Within your node class, itself, you revise (or add) this method:

```
public class BioChamberNode: UtilityNode
{
    :

    // The following property associates a WPF framework element with this node type. (Typically,
    // as is the case here, the framework element is a kind of WPF UserControl.) Subsequently,
    // whenever this node type is instantiated, an instance of the framework element will be
    // instantiated, as well, and appear on the face of the node within GC's graph.

    public override Type TypeOfCustomViewContent => typeof(BioChamberNodeView);

    :
}
```

- **Exchanging data between your node class and its corresponding custom UserControl**

When you write the custom control, you can assume that the corresponding view-model class will be an instance of the node type being represented. In other words, when GC displays an instance of that node type in its graph, it automatically assigns that node instance to the DataContext property of your custom UserControl.

◆ In the MVVM pattern, your node class serves as both the Model and the View Model, with your custom WPF UserControl serving as the View.

WPF's standard data-binding mechanism will work. Your node properties aren't dependency properties, but they don't need to be: The class `Node`, which is the base class of all GC node types, implements `INodePropertyChanged`, and it raises the `NotifyPropertyChanged` event whenever any node input value changes.

### "Cheating" the MVVM pattern

Since we know that our custom control's `DataContext` property will always be an instance of our node class (or null), we can use that information to implement some activities more efficiently than going through WPF's data-binding process.

```
public partial class BioChamberNodeView: UserControl
{
    public BioChamberNodeView()
    {
        InitializeComponent();
    }

    bool TryGetNode(out BioChamberNode node)
    {
        node = DataContext as BioChamberNode;
        return node != null;
    }

    :

    void ExampleMethod()
    {
        if (TryGetNode(out BioChamberNode node)
        {
            // Get/set property values and/or call methods directly on the node.
        }
    }
}
```

You may wonder why we implemented the "GetNode" method as a "Try" method, rather than simply:

```
BioChamberNode GetNode() => (BioChamberNode) DataContext; // Risky.
```

The reason is a matter of timing. Depending on when your custom control calls that method relative to its own startup and shutdown, the DataContext property may not have been initialized, yet, or it may have been cleared. Furthermore, under certain circumstances, the DataContext property will return the value of the DataContext in an outer-level control. (GC's Graph panel, like almost all the GC-specific panels and dialogs, is a WPF control.)

- **Design your WPF control to be resizable**

(This is good WPF practice regardless of the context.)

Remember that GC allows the user to resize any node in the graph at any time. In the case of a utility node, the WPF custom UI is resized along with the node.

Therefore, you should avoid hard coding the dimensions of your custom control, as well as the dimensions and/or locations of any of its sub-controls. Instead, use panels, margins, and alignments (as they're intended to be used) so that your control will continue to look attractive no matter what size or aspect ratio it has.

Of course, if you feel you have more material to display than would fit in a typically sized graph node, you can expose that additional material in the form of a popup or drop-down window.

## 5. General Operations

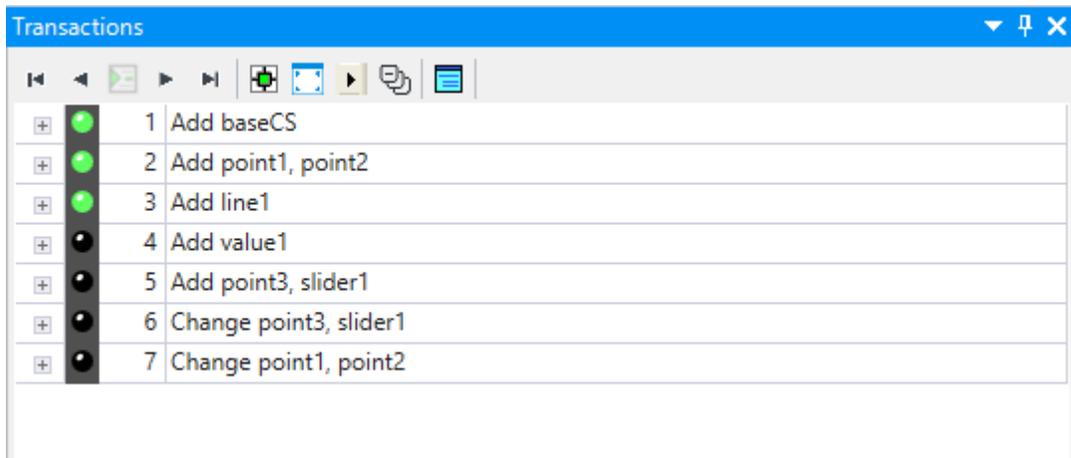
## 5.1. Creating and modifying nodes programmatically

Generally, whenever you want to programmatically create new nodes or update existing nodes, you do so by creating an in-memory transaction that describes what you want to do, then telling GC to perform that transaction.

- [GC transactions](#)
- [Creating and performing a transaction](#)

### 5.1.1. GC transactions

Transactions are those things the user works within the transaction player. The following image shows seven transactions, the first three of which have been played:



The user can view and edit any transaction in GC's Editor. Here are the Editor's presentations of transactions 2 and 4, respectively:

```
Transaction 2 x
transaction 2 stateChange 'Add point1, point2'
{
  gcModel
  {
    node User.Objects.point1 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByCartesianCoordinates';
      CoordinateSystem     = baseCS;
      XTranslation         = <free> 9.8;
      YTranslation         = <free> 45.5;
      ZTranslation         = <free> 0.0;
      GraphLocation        = {306.761, 41.584};
    }
    node User.Objects.point2 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByCartesianCoordinates';
      CoordinateSystem     = baseCS;
      XTranslation         = <free> 37.3;
      YTranslation         = <free> 15.6;
      ZTranslation         = <free> 0.0;
      GraphLocation        = {290.761, 254.888};
    }
  }
}

Transaction 4 x
transaction 4 stateChange 'Add value1'
{
  gcModel
  {
    node User.Objects.value1 Bentley.GC.NodeTypes.Value
    {
      Technique           = 'Default';
      Value               = line1.Length;
      GraphLocation        = {788.95, 83.127};
    }
  }
}
```

You can see these are both state-change ('stateChange') transactions. GC supports other types of transactions, but state-change is the type that's relevant to this topic.

As its name implies, a state-change transaction describes a change in the state of GC's model. It means, "when the user plays this transaction, these node-property assignments will be applied to the model." Note that...

- The transaction describes only the changes made directly to the indicated nodes. It doesn't describe the "ripple" effects those changes may have on other nodes in the model.
- The order in which the nodes are listed is irrelevant. For example, in transaction 2, the block describing 'line1' could appear before the blocks describing 'point1' and 'point2'.
- Comparing transaction 2 and transaction 7:

```
Transaction 2 x
transaction 2 stateChange 'Add point1, point2'
{
  gcModel
  {
    node User.Objects.point1 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByCartesianCoordinates';
      CoordinateSystem    = baseCS;
      XTranslation         = <free> 9.8;
      YTranslation        = <free> 45.5;
      ZTranslation        = <free> 0.0;
      GraphLocation       = {306.761, 41.584};
    }
    node User.Objects.point2 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByCartesianCoordinates';
      CoordinateSystem    = baseCS;
      XTranslation         = <free> 37.3;
      YTranslation        = <free> 15.6;
      ZTranslation        = <free> 0.0;
      GraphLocation       = {290.761, 254.888};
    }
  }
}

Transaction 7 x
transaction 7 stateChange 'Change point1, point2'
{
  gcModel
  {
    node User.Objects.point1 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByCartesianCoordinates';
      CoordinateSystem    = baseCS;
      XTranslation         = <free> 17.0;
      YTranslation        = <free> 50.0;
      ZTranslation        = <free> 0.0;
      GraphLocation       = {306.761, 41.584};
    }
    node User.Objects.point2 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByCartesianCoordinates';
      CoordinateSystem    = baseCS;
      XTranslation         = <free> 29.6;
      YTranslation        = <free> 12.2;
      ZTranslation        = <free> 0.0;
      GraphLocation       = {290.761, 254.888};
    }
  }
}
```

You can see that, aside from their titles and some of the numbers, these transactions are identical. And, yet they'll behave differently because they'll be played under different conditions (i.e., at different stages in the GC model):

- When transaction 2 is played, the nodes 'point1' and 'point2' won't exist, yet. So, this transaction will create both of those nodes from scratch and assign their initial property values.
- When transaction 7 is played, the nodes 'point1' and 'point2' will already exist. So, this transaction will "merely" assign new property values to those existing nodes.

In other words: A transaction, as it stands, doesn't need to know (or care) whether or not its nodes exist. That decision is deferred to when the transaction is played. At that time, GC will spontaneously create any nodes that don't already exist.

- **Transactions are data structures**

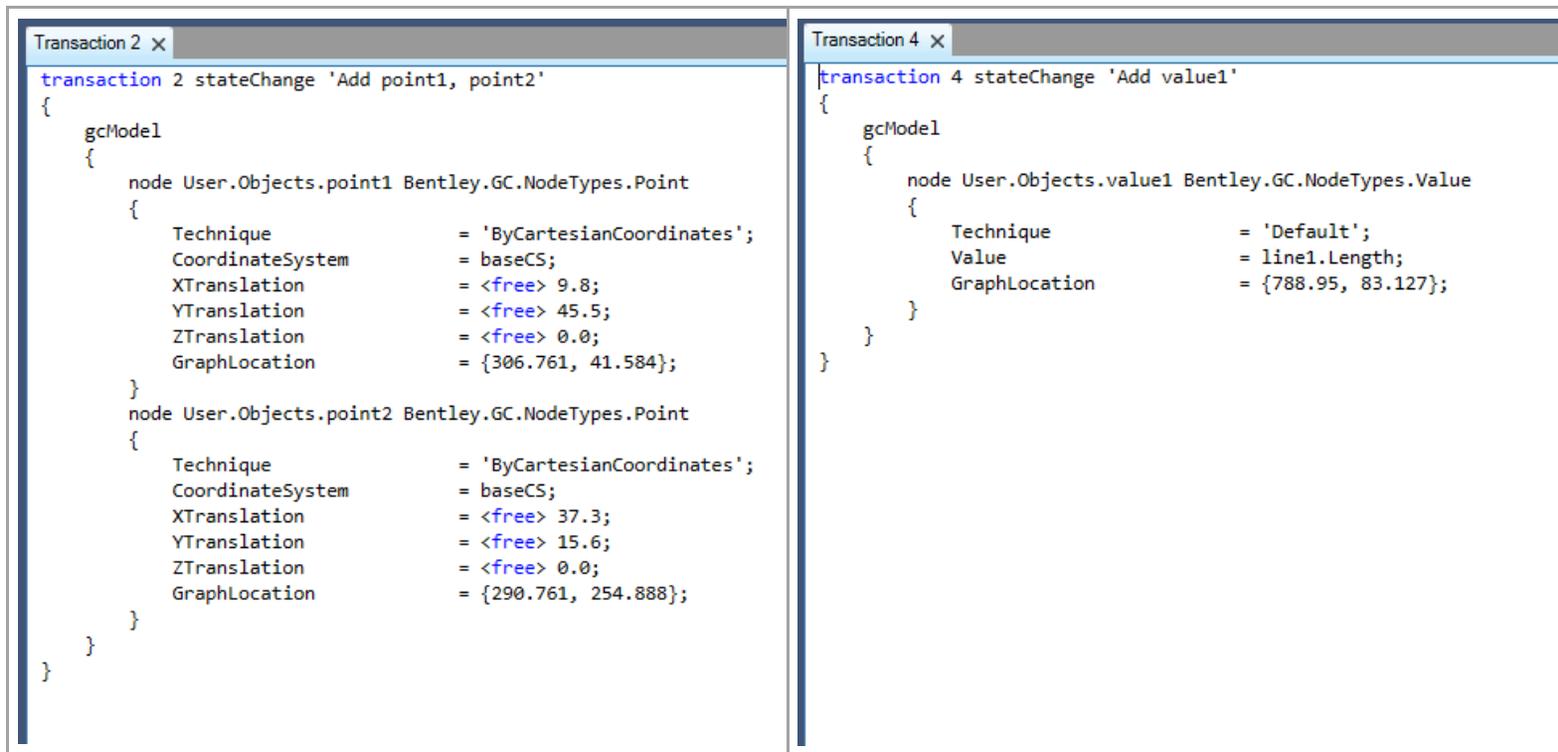
Internally, GC's transactions are stored in memory as data structures. They're translated to/from textual form only when needed: When the user edits a transaction, or when GC reads or writes a GCT file. So, when the user edits a transaction...

- At that moment, GC generates human-friendly text that corresponds to the contents of that transaction's data structure.
  - GC passes that text to the Editor so the user can view it and (optionally) change it.
1. When the user is finished making changes, GC parses that changed text back into its data-structure form and then discards that text.

## 5.1.2. Creating and performing a transaction

GC performs a lot of housekeeping when a transaction is played, and therefore that's the best way to create and/or modify nodes even at the C# level. (That's the way GC, itself, creates and/or modifies nodes when it needs to do so programmatically.)

As an example, let's programmatically create data structures corresponding to our transactions 2 and 4:



```
Transaction 2 x
transaction 2 stateChange 'Add point1, point2'
{
  gcModel
  {
    node User.Objects.point1 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByCartesianCoordinates';
      CoordinateSystem    = baseCS;
      XTranslation        = <free> 9.8;
      YTranslation        = <free> 45.5;
      ZTranslation        = <free> 0.0;
      GraphLocation       = {306.761, 41.584};
    }
    node User.Objects.point2 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByCartesianCoordinates';
      CoordinateSystem    = baseCS;
      XTranslation        = <free> 37.3;
      YTranslation        = <free> 15.6;
      ZTranslation        = <free> 0.0;
      GraphLocation       = {290.761, 254.888};
    }
  }
}

Transaction 4 x
transaction 4 stateChange 'Add value1'
{
  gcModel
  {
    node User.Objects.value1 Bentley.GC.NodeTypes.Value
    {
      Technique           = 'Default';
      Value               = line1.Length;
      GraphLocation       = {788.95, 83.127};
    }
  }
}
```

Here's the C# code that does it:

```
GCMModelChangeStatement GetTransactionStatement2(GCMModel model)
{
    Namespace namespac = model.NamespaceForNewInstances(); // Get the GC namespace that contains all of this GC
model's
  // node instances.

    GCMModelChangeStatement statement = GCSPACECHANGE_TRANSACTION.GetGCMModelChangeStatement(model); // Create an empty
statement.
  // 'gcModel'

    {
        // Add the information for 'point1'.

        NamePath namePath = new NamePath(namespac, "point1");

        UpdateNodeSubStatement subStatement = statement.AddUpdateNodeSubStatement(namePath, typeof(PointNode));

        subStatement.AddTechniqueSpecification("ByCartesianCoordinates"); // It's required to specify the node's
technique.

        // Now add the other property assignments.
        //
        // In each of these cases, we pass a third argument, InputExpressionFlags.Free, because the transaction that
// we're emulating (Transaction 2) sets all these property values to '<free>'. (That is, the user can use the
would // the mouse to move the point in all directions). If we wanted these property values to be non-free, we
        // simply omit that third argument in each case.

        subStatement.AddPropertyAssignment("XTranslation", 9.8, InputExpressionFlags.Free);
        subStatement.AddPropertyAssignment("YTranslation", 45.5, InputExpressionFlags.Free);
        subStatement.AddPropertyAssignment("ZTranslation", 0.0, InputExpressionFlags.Free);

        // We do NOT add an assignment statement for 'GraphLocation', because GC figures that out on its own. (That's
why // that assignment is marked '<auto>' in the transaction that we're emulating.)
    }
    {
        // Now do the same thing for 'point2'.

        NamePath namePath = new NamePath(namespac, "point1");
    }
}
```

```
UpdateNodeSubStatement subStatement = statement.AddUpdateNodeSubStatement(namePath, typeof(PointNode));

subStatement.AddTechniqueSpecification("ByCartesianCoordinates");

subStatement.AddPropertyAssignment("XTranslation", 37.3, InputExpressionFlags.Free);
subStatement.AddPropertyAssignment("YTranslation", 15.6, InputExpressionFlags.Free);
subStatement.AddPropertyAssignment("ZTranslation", 0.0, InputExpressionFlags.Free);
}

return statement;
}
```

```

GCModelChangeStatement GetTransactionStatement4(GCModel model)
{
    Namespace namespace = model.NamespaceForNewInstances();

    GCModelChangeStatement statement = GCSpaceChangeTransaction.GetGCModelChangeStatement(model);

    {
        // Add the information for 'value1'.

        NamePath namePath = new NamePath(namespace, "value1");

        UpdateNodeSubStatement subStatement = statement.AddUpdateNodeSubStatement(namePath, typeof(ValueNode));

        subStatement.AddTechniqueSpecification("Default");

        subStatement.AddPropertyAssignmentLiteral("Value", "line1.Length");

        // In the preceding statement, instead of calling 'AddPropertyAssignment', we call
        // 'AddPropertyAssignmentLiteral'. The latter is used when we want to specify explicitly
        // the expression script to be assigned to this property. Imagine if, instead, we had
        // said:
        //
        //     subStatement.AddPropertyAssignment("Value", line1.Length); // Won't work as intended.
        //
        // Firstly, the current C# code doesn't know what 'line1' is. Secondly, even if it did,
        // the code expression "line1.Length" would evaluate to just a bare number, which would
        // then be recorded into the property assignment. There would be no connection between
        // 'value1' and 'line1'.
    }

    return statement;
}

```

- **Performing the transaction**

So now we have a way to get a `GCMModelChangeStatement` that corresponds to the body of a transaction. What do we do with it?

We tell GC to perform the transaction. This is analogous to the user playing the transaction in GC's Transactions panel.

```
GCMModel model = GCTools.UIDefaultGCMModel(); // Get the GC model object.
GCMModelChangeStatement statement = GetTransactionStatement(model);
statement.PerformTransaction(NodeUpdateReason.Internal, TransactionPerformanceOptions.LockAndUnlockGCspace);
    // NodeUpdateReason.Internal means, essentially, "programmatically".
    //
    // TransactionPerformanceOptions.LockAndUnlockGCspace means to make sure that the enclosing GC
space    // (outer-level model container) is locked against any possible changes being made by something
else    // in the GC environment.
```

## 5.2. Create new nodes that are attached to existing nodes

This demonstrates how to programmatically find all GC nodes that are of a certain type (or otherwise fulfill certain criteria), and, in each case, create a new GC node (of the same or a different type) that's connected to the node that was found.

This example looks for all Line nodes, and, for each one, adds a new Point node that's connected to the Line node as its center point.



Following is a complete C# code module that defines a class, `NodeCreationDemo`, whose single method, `AddCenterPointsToAllLines`, performs this operation.

Caveat: Calling this method more than once will create additional Point nodes, which you'll see in GC's graph. However, since each of those new Point nodes will coincide with a Point node that was created in a previous call to this method, the graphical view won't seem to change. As an alternative, you could modify this method to take a 'T' value parameter (of type `double`), then use that parameter instead of 0.5 when assigning to each new point's 'T' property.

```

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Bentley.GenerativeComponents;
using Bentley.GenerativeComponents.ElementBasedNodes;
using Bentley.GenerativeComponents.ElementBasedNodes.Specific;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.MicroStation;
using Bentley.GenerativeComponents.Transactions;
using Bentley.GenerativeComponents.Transactions.GCSpaceChange;

namespace SampleAddIn
{
    static class NodeCreationDemo
    {
        static internal void AddCenterPointsToAllLines()
        {
            // This method finds all the Line-type nodes in the default GC model and, for each one, creates a new Point
            // node that's attached to the center point of that line. (That is, one of the new Point's inputs is the
            // Line that it's attached to.)

            // This method demonstrates how to...
            //
            // 1. Iterate through all the GC nodes in a GC model, finding those nodes that fulfill a specific criteria,
            //    such as being of a specific type.
            // 2. Add new GC nodes to the GC model by building an in-memory transaction and then submitting it to GC
            //    (which, in GC terminology, is called performing the transaction).

            // Start by getting the default GC model. Alternatively, the GC model could be passed as an argument to this
            // method.

            GCModel model = GCTools.UIDefaultGCModel();

            // We lock the GC model's enclosing GC space (outer-level model container) before we perform any manipulations,
            // then we'll unlock it when we're done. This prevents any possible conflicts with something else that may
            // may be trying to manipulate the model at the same time we're manipulating it.

```

```

model.LockGCSpace();

try
{
    // Get all the lines (LineNodes) in the model. We must cache the result to a local variable because we're
    // about to add additional nodes to the model, which will otherwise trigger a "collection has changed" error
    // if we try to iterate directly through the model.TopLevelNodes().

    LineNode[] lines = model.TopLevelNodes().OfType<LineNode>().ToArray();

    // Iterate through all the line nodes we've found, programmatically adding a center point to each one.

    foreach (LineNode line in lines)
        addCenterPointToLine(line);

    // The following call doesn't do anything directly; it "merely" sets an internal flag such that, when we
    // subsequently unlock the GCSPACE (below), that unlocking will perform some additional gyrations to ensure
    // that MicroStation is synchronized with the changes we've made to the GC model.

    model.SyncUpMicroStation();
}
finally
{
    model.UnlockGCSpace(); // Relinquish our lock so that the GC model is available, again, to other processes.
}

void addCenterPointToLine(LineNode line)
{
    // This local function programmatically builds a transaction in memory, then performs that transaction
    // (i.e., submits it to GC).
    //
    // If this transaction were recorded into a GC transaction list, would look like this (for example):
    //
    // transaction stateChange
    // {
    //     gcModel
    //     {
    //         node User.Objects.point7 Bentley.GC.NodeTypes.Point
    //         {
    //             Technique           = 'ByParameterAlongCurve';
    //             Curve                = line1;
    //             T                    = <free> 0.5;
    //         }
    //     }
    // }
}

```

```

//      }
//    }
//
// Actually, what we build herein is just the BODY of the transaction, starting from "gcModel". The outer-level
// "transaction stateChange" shell is generated implicitly.

// So we'll start by creating an empty 'gcModel' block.

GCModelChangeStatement statement = GCSpaceChangeTransaction.GetGCModelChangeStatement(model);

// The following passage adds the information to create a new GC node (in this case, the Point node that will
// be the centerpoint of the given line).
//
// This example creates only one node within this transaction. However, if you wanted to create additional
// nodes simultaneously, you could repeat the following passage (with appropriate modifications) to add more
// sub-statements to the statement.

// Next, we'll add an empty 'node' block, which specifies the name and type of the new node.
// We must provide the new node's name in the form of a "name path" (namespace + name).

NamePath namePathOfNewPoint = getNamePathOfNewNode(typeof(PointNode));

UpdateNodeSubStatement subStatement = statement.AddUpdateNodeSubStatement(namePathOfNewPoint, typeof(PointNode));

// Next, we add an assignment statement that assigns the node's technique, which is required.

subStatement.AddTechniqueSpecification("ByParameterAlongCurve");

// Finally, one by one, we add the property assignments.

subStatement.AddPropertyAssignment("Curve", line);
subStatement.AddPropertyAssignment("T", 0.5, InputExpressionFlags.Free);

// This is the end of the passage that adds one node to the transaction statement.

// Now our in-memory transaction is complete, so all we need to do is perform it:

statement.PerformTransaction(NodeUpdateReason.Internal); // Essentially, NodeUpdateReason.Internal means
// "programmatically".

NamePath getNamePathOfNewNode(Type nodeType)
{

```

```

created,          // Compute and return the default name path (that is, GC namespace + name) of a node that hasn't been
Lines            // yet. For example, if the node type is Line (i.e., typeof(LineNode)) and the GC model already contains
                // named 'line1' through 'line3', this method will return 'line4' (within the appropriate GC namespace).

                Namespace namespac = model.NamespaceForNewInstances();
namespac);       NamePath result = NodeNamePathFormulator.FormulateUniqueNodeNamePathFromNativeType(model, nodeType,
                return result;
                }
            }
        }
    }
}

```

## 6. Using GC as a Service

## 6.1. Introduction

GC can be controlled externally from a separate, standalone application.

In that scenario, GC exposes a "service" interface by which the separate application -- the "client" -- can create a GC model, modify an existing model, query a model for its values, and perform other operations.

When GC (or, rather, its host application, such as OBD) is running in "GC service mode", it's dedicated solely to servicing client applications; it's unavailable for user operations.

- **Dependence on the .NET Framework**

GC's service model is based on WCF (Windows Communication Foundation), one of Microsoft's .NET technologies.

Unfortunately, WCF is no longer supported in the newest generation of .NET, .NET Core (and beyond).

Therefore, any client application that wants to use GC's service model must be based on the previous .NET generation, .NET Framework. According to [Microsoft](#), .NET Framework will continue to be distributed with future versions of Windows.

## 6.2. How to start GC in service mode

To start GC -- or, rather, its host application, such as Open Buildings Designer -- in service mode, we must add the switch "/service" to the command line that launches that application.

Here's one way to do that. We'll use Open Buildings Designer (OBD) as our example.

1. Navigate to — but do not click — the shortcut that launches OBD. In a default installation, that shortcut can be found at:

```
Start Menu
  → Programs
    → OpenBuildings Designer CONNECT Edition
      → OpenBuildings Designer CONNECT Edition
        → OpenBuildings GenerativeComponents CONNECT Edition
```

2. Right-click on that shortcut and select "Copy".
3. Switch to your desktop (or any local folder of your choice) and select "Paste" (or "Paste Shortcut").
4. Rename the newly-pasted shortcut to something like "OBD GC Service".
5. Right-click on the newly-pasted shortcut and select "Properties".

Look at the "Target" field. Its value will be something like this:

```
"C:\Program Files\Bentley\OpenBuildings Connect Edition\OpenBuildingsDesigner\OpenBuildingsDesigner.exe" -wsGC=1
```

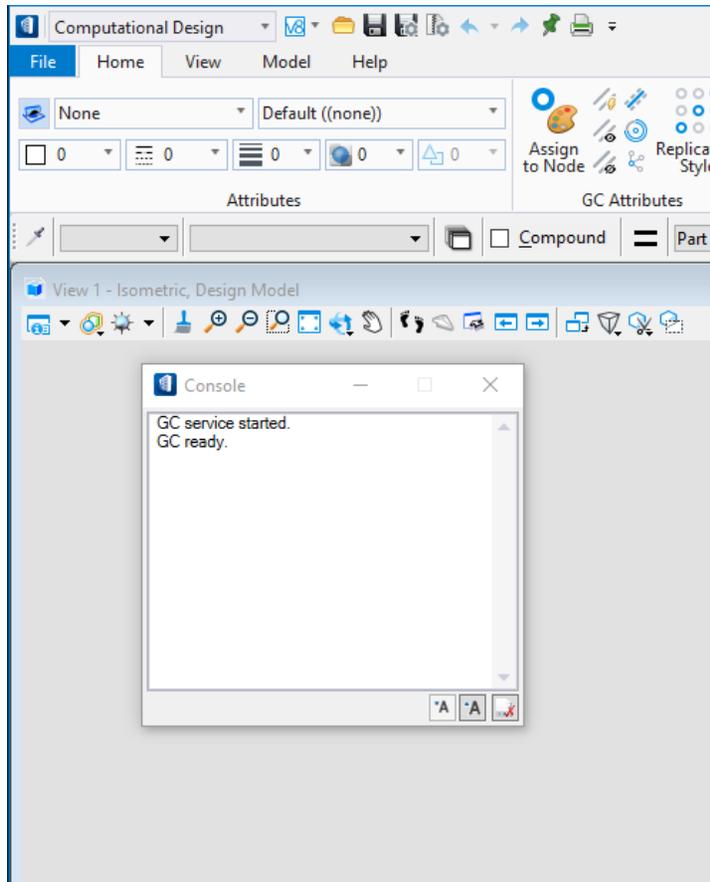
Add the following option (separated by a space as shown):

```
"C:\Program Files\Bentley\OpenBuildings CONNECT Edition\OpenBuildingsDesigner\OpenBuildingsDesigner.exe" -wsGC=1 -wsGC_SERVICE
```

Then click OK to close the Properties dialog.

◆ The "-wsGC\_SERVICE" option can take parameters. There's more information [here](#).

6. Double-click on the shortcut to launch OBD/GC.
7. Although you won't be doing anything within OBD/GC, per se, it will still ask you to select or create a design file or a transaction file. We suggest creating an empty DGN or GCT file just for this purpose, and thereafter opening that same file.
8. When OBD/GC fully opens, it displays a console window as shown:



During your sessions when using GC as a service in this way, that window will display informational messages about the service's activities.

Note that, when GC is running as a service in this way, you cannot, and should not, perform any GC operations directly within OBD/GC. At this point, GC is patiently waiting for a service client to connect and start telling it what to do.

## 6.2.1. Service mode parameters

As described on [the previous page](#), you start GC in service mode by adding the following option to the command that launches GC's host application (e.g., OBD):

`-wsGC_SERVICE`

For example:

```
"C:\Program Files\Bentley\OpenBuildings CONNECT Edition\OpenBuildingsDesigner\OpenBuildingsDesigner.exe" -wsGC=1 -wsGC_SERVICE
```

By default, the GC service is started in named-pipe protocol mode, suitable for connecting to a client app that's running on the same computer. However, the "-wsGC\_SERVICE" option can be revised to handle other situations. All of its forms are listed below.

|                                                                       |                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-wsGC_SERVICE</code>                                            | Activates the service using the default options; the same as <code>-wsGC_SERVICE=pipe</code> .                                                                                                                                                                                                                                            |
| <code>-wsGC_SERVICE=pipe</code>                                       | Activates the service using named pipes as the transport protocol. This is the recommended option when both ABD and the GC service client app are running on the same computer.                                                                                                                                                           |
| <code>-wsGC_SERVICE=tcp</code><br><code>-wsGC_SERVICE=tcp:8080</code> | Activates the service using TCP as the transport protocol. This is the recommended option when ABD and the GC service client app are running on different computers within the same local network. The specified number is the port number to use; for example, " <code>-wsGC_SERVICE=tcp:8080</code> ". The default port number is 8000. |
| <code>-wsGC_SERVICE=pipe,tcp:8080</code>                              | Options can be combined (separated by commas) to support multiple protocols simultaneously.                                                                                                                                                                                                                                               |
| <code>-wsGC_SERVICE=name:Fred</code>                                  | Specifies the name of the service as it will be presented to client apps. (Can be combined with any of the other options.) For example, instead of using the named pipe " <code>net.pipe://localhost/GenerativeComponents</code> ", client apps will use the named pipe " <code>net.pipe://localhost/Fred</code> ".                       |

## 6.3. Walkthrough: Create a small client application

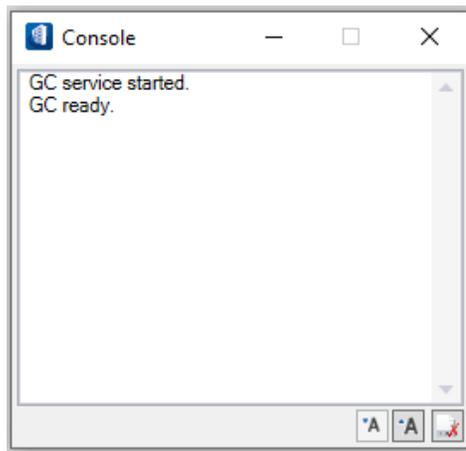
This section describes how to create a small client application that uses GC as a service. Hopefully, this will give you enough information to incorporate the GC service into your applications.

◆ If you want to peek at where we're going, [here](#) is a listing of the final version of the file, Program.cs, which we'll create during this walkthrough.

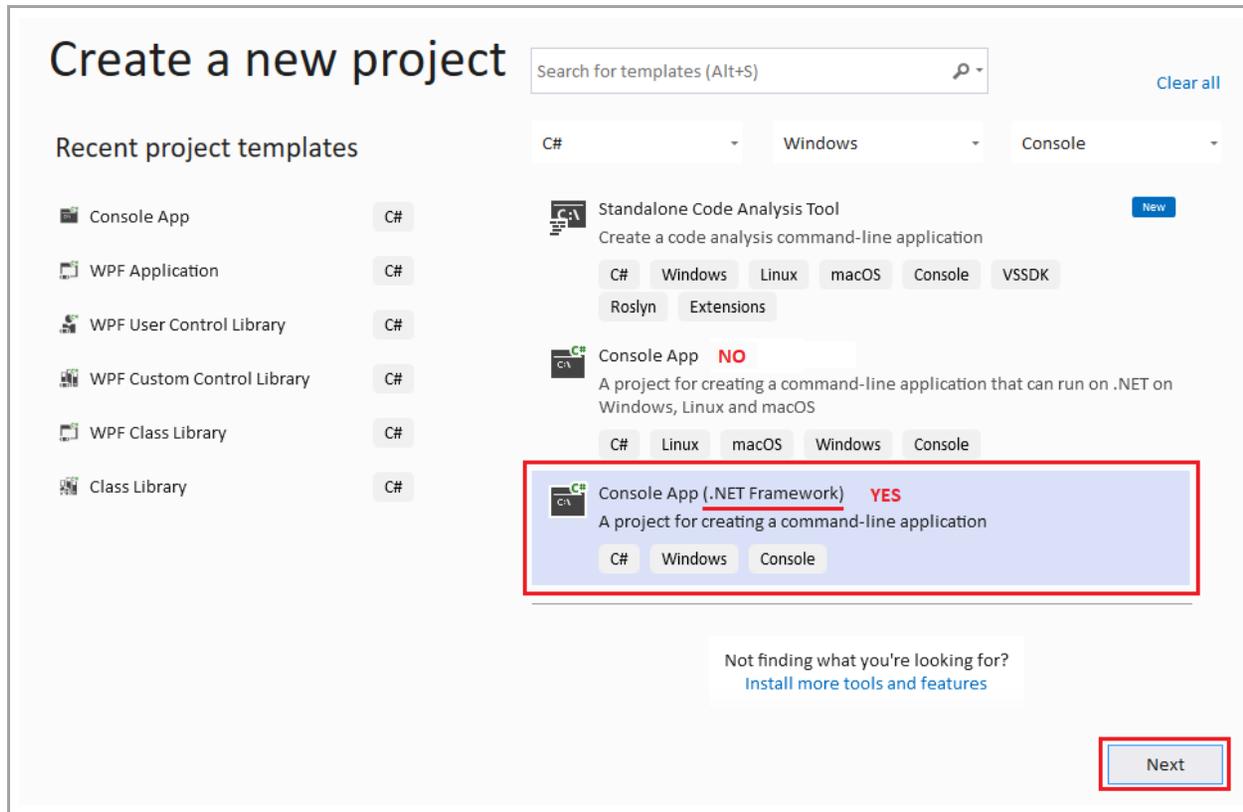
## 6.3.1. Getting started

1. Start GC in service mode, as described [here](#). Leave it up and running for the remainder of this walkthrough.

◆ The GC service is not self-starting. Whenever you want to use it, you must manually launch GC's host application (e.g. OBD) and create or select a DGN or GCT file. The GC service is ready to use when you get to this dialog within the host application:



2. Start Visual Studio. This walkthrough uses Visual Studio 2022 Community Edition, though any relatively-recent version of Visual Studio should do.
3. Create a new Console app. Make sure it's based on .NET Framework rather than the default (which is .NET Core).



4. Change the name of the project (and solution) to "SimpleGCServiceClient".

The location can be wherever you want on your local computer.

Choose the latest version of the .NET Framework (4.8 at this writing).

## Configure your new project

Console App (.NET Framework) C# Windows Console

Project name  
SimpleGCServiceClient

Location  
D:\OneDrive - Bentley Systems, Inc\GC\API documentation\

Solution name ⓘ  
SimpleGCServiceClient

Place solution and project in the same directory

Framework  
.NET Framework 4.8

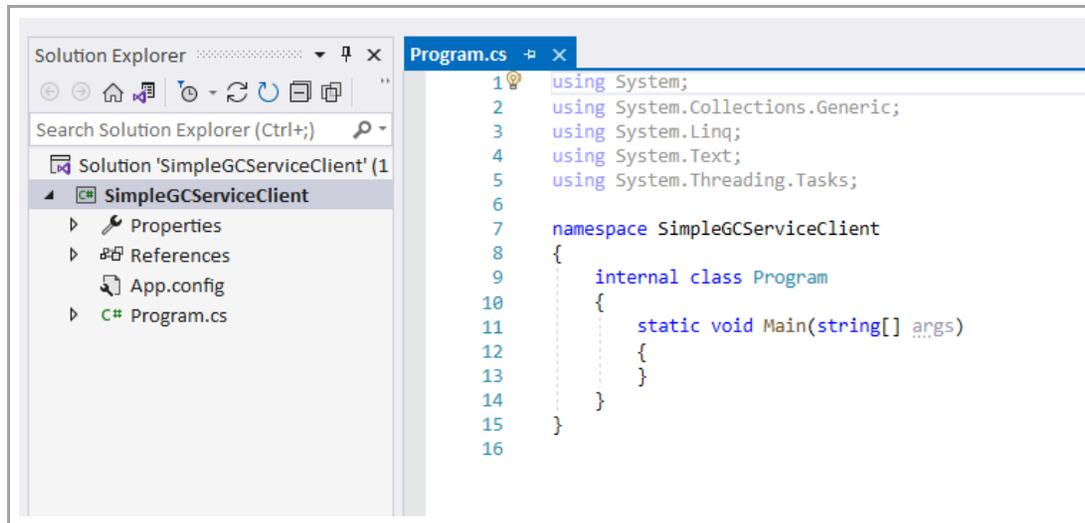
Back Create

◆, Unlike a GC add-in, a GC service client app doesn't need to conform to any specifications beyond being based on the .NET Framework rather than .NET Core.

In particular...

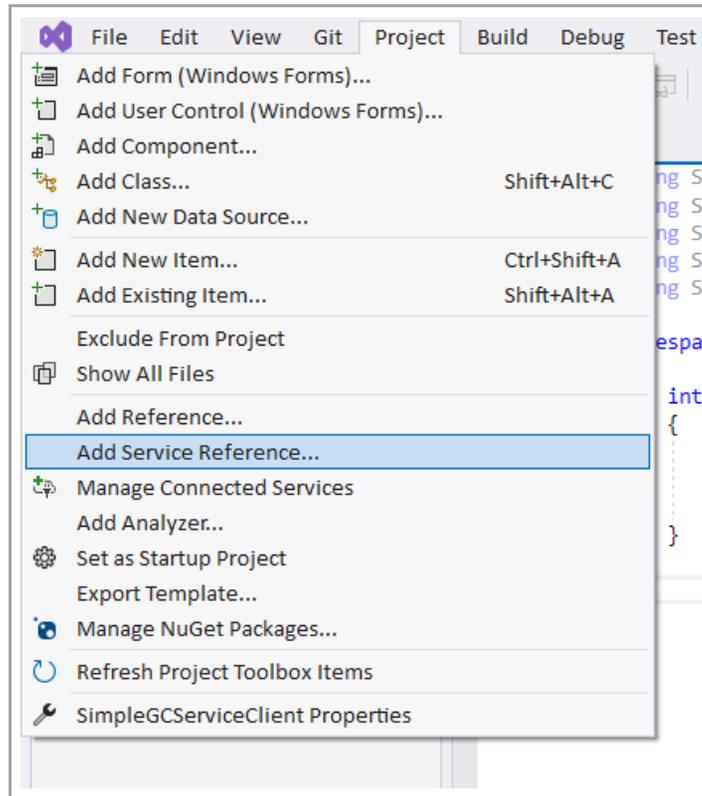
- The client app doesn't require any references to any Bentley assemblies.
- It doesn't require a particular version of the .NET Framework (though we recommend the latest version, which is 4.8 in this writing).
- It doesn't matter whether it targets "Any CPU", "x64", or anything else.
- It doesn't even need to be a C# app! It could be written in an entirely different language, such as Visual Basic .NET. However, this walkthrough assumes we're creating a C# app.

This brings us to the main development screen for our new app:



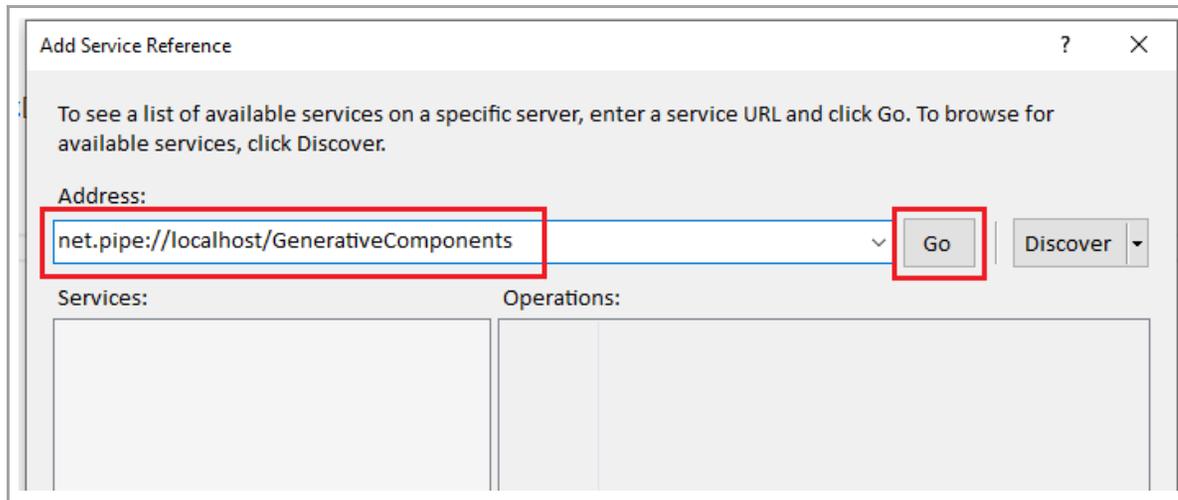
## 6.3.2. Adding the GC service reference

1. Open the Project menu and select "Add Service Reference..."

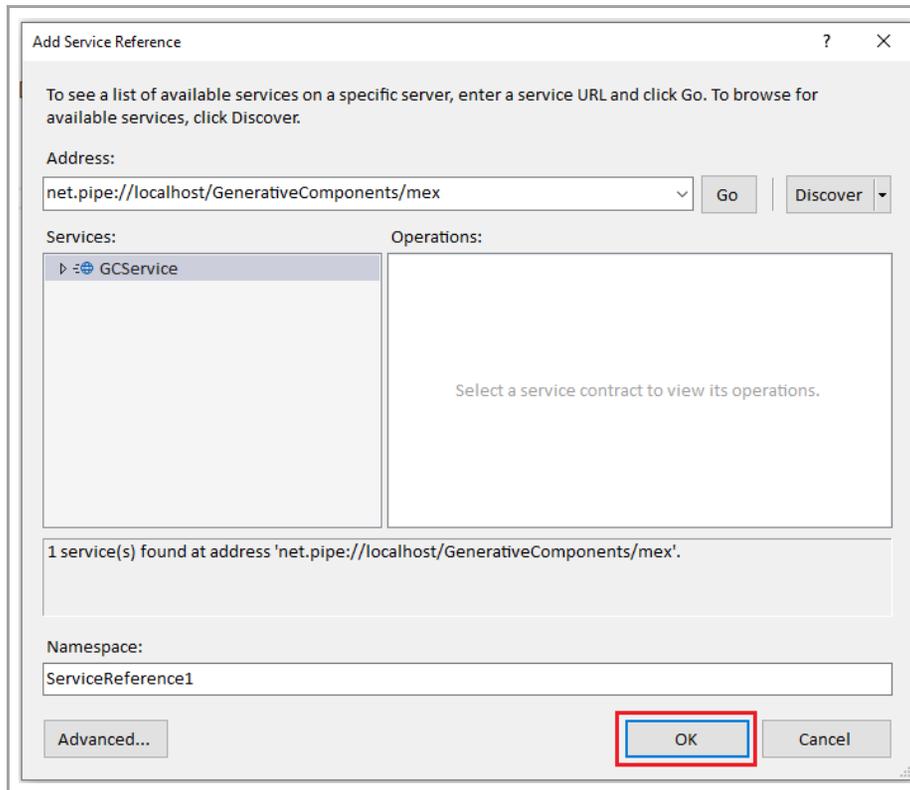


When the "Add Service Reference" dialog appears, enter the following into the "Address" field, then click "Go".

```
net.pipe://localhost/GenerativeComponents
```



Assuming all's well, Visual Studio will populate the remainder of the "Add Service Reference" dialog. Click OK to close it.



After a moment, Visual Studio will generate the corresponding "Connected Services" code and add it to your project.

2. Revise Program.cs to look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using SimpleGCSERVICEClient.ServiceReference1;
namespace SimpleGCSERVICEClient
```

```
{
  internal class Program
  {
    static GCServiceClient gcService = new GCServiceClient();

    static void Main(string[] args)
    {
    }
  }
}
```

You might have questions about this statement:

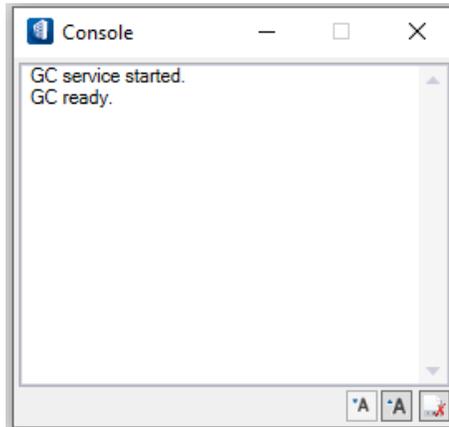
```
static GCServiceClient gcService = new GCServiceClient();
```

When Visual Studio generates the code for a service reference, it names the primary class "<ServiceName>Client". So, the primary class for the GCService reference is named GCServiceClient. That's a bit confusing, in this author's opinion; from our application's point of view, that generated class represents the service, itself. Meanwhile, the client is our application!

Well, whatever. I've used their generated class name -- GCServiceClient -- but, in defiance, I'll name the variable, itself, "gcService", which is what our application will treat it as.

3. Press F5 (or click the Run button) to build and run SimpleGCServiceClient as it stands so far, just to prove that everything builds and runs successfully. At this point, it's uneventful: It simply starts and then immediately closes.

Furthermore, if we switch back to OBD/GC, we see that the Console window has nothing new to report:



Fair enough. Now let's talk to GC from SimpleGCServiceClient.

### 6.3.3. Our first communications with the GC service

- **Say hello**

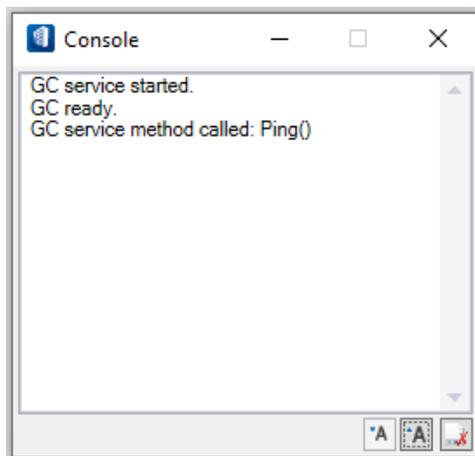
In our class Program, revise the definition of the Main method to look like this:

```
static void Main(string[] args)
{
    gcService.Ping();
}
```

The method "Ping()" simply establishes communication between the GC service client and the GC service, itself. It's usually a good idea to start every GC service client session with a call to Ping(), just to make sure everything is up and running as expected.

Now run SimpleGCServiceClient again. Again, it will simply start and then close without incident.

But now there's something new on the GC side:



So, when OBD/GC is running in GC service mode, the Console window provides a running commentary of the activities performed by the service on behalf of the client.

◆ Throughout the rest of this walkthrough, we encourage you to regularly peek back at this Console window to see what's going on from the GC service's point of view. In some cases, it may provide error information that's not available anywhere else.

- **Using GC as a calculator**

Now let's see how to have the GC service perform some calculations on our behalf, using GC's scripting language.

Among the types that Visual Studio generated for us are these two:

|                                                                                                                         |                                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>public struct LabeledNumber {     public string Label { get; set; }     public double Number { get; set; } }</pre> | <pre>public struct AnnotatedLabeledNumbers {     public string Annotation { get; set; }     public LabeledNumber[] LabeledNumbers { get; set; } }</pre> |
|-------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|

Many of the GC service's methods take and/or return instances of those types.

Let's revise the content of our Program class to look like this:

```
internal class Program
{
    static GCServiceClient gcService = new GCServiceClient();

    static void Main(string[] args)
    {
        gcService.Ping();
        PerformCalculations();
    }

    static void PerformCalculations()
    {
        string[] scriptExpressions = { "2 + 3", "Sqrt(9)", "new DVector2d(3, 4).Magnitude" };

        AnnotatedLabeledNumbers results = gcService.GetNumericValues("", scriptExpressions);
    }
}
```

```

    DisplayResults(results);
}

static void DisplayResults(AnnotatedLabeledNumbers results)
{
    foreach (LabeledNumber result in results.LabeledNumbers)
        Console.WriteLine($"{result.Label, -30} = {result.Number}");

    Console.WriteLine();
    Console.WriteLine("Finished. Press [Enter] to close. ");
    Console.ReadLine();
}
}

```

The key statement here is:

```

AnnotatedLabeledNumbers results = gcService.GetNumericValues("", scriptExpressions);

```

where the GC service method `GetNumericValues` takes an array of script expressions and returns a single instance of `AnnotatedLabeledNumbers`, which holds the results of those expressions.

◆ Many of the GC service methods take a first parameter named 'gcModelName', whose type is a string.

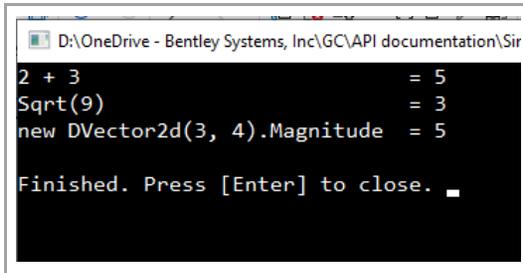
```

nbers = gcService.GetNumericValues(
    AnnotatedLabeledNumbers GCServiceClient.GetNumericValues(string gcModelName, string[] scriptExpressions)

```

However, currently, the value of that parameter is ignored in all cases. We recommend that you simply pass an empty string ("") when you call one of those GC service methods.

And now we get this output when we run:



```
D:\OneDrive - Bentley Systems, Inc\GC\API documentation\Sir
2 + 3 = 5
Sqrt(9) = 3
new DVector2d(3, 4).Magnitude = 5
Finished. Press [Enter] to close. █
```

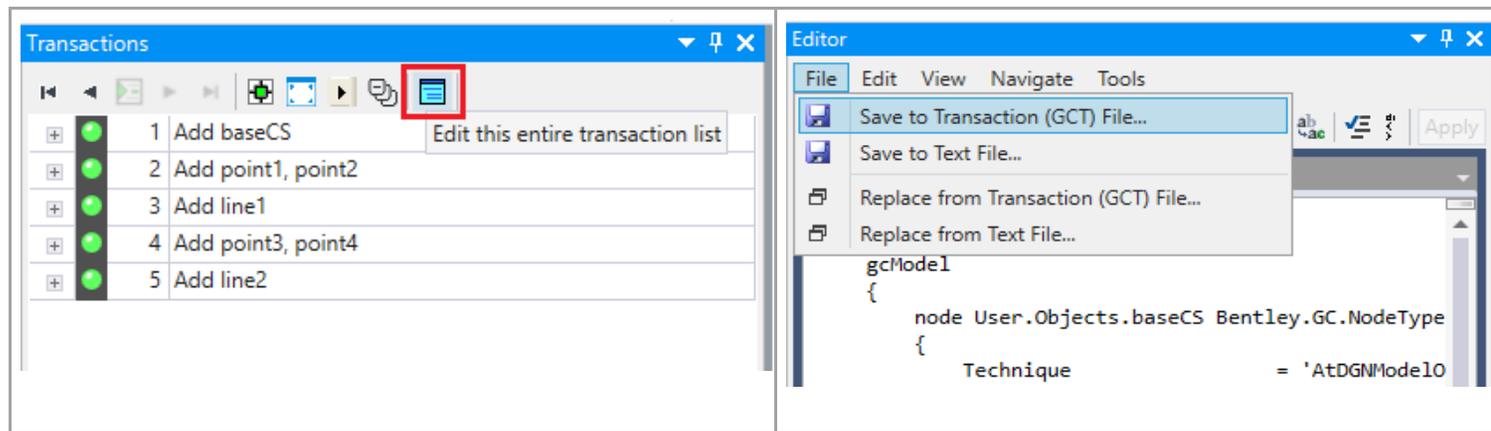
Ooo! Exciting stuff!

### 6.3.4. Preparing a transaction file for use with the GC service

- **Create a GC transaction file (GCT file) containing the GC model you want**

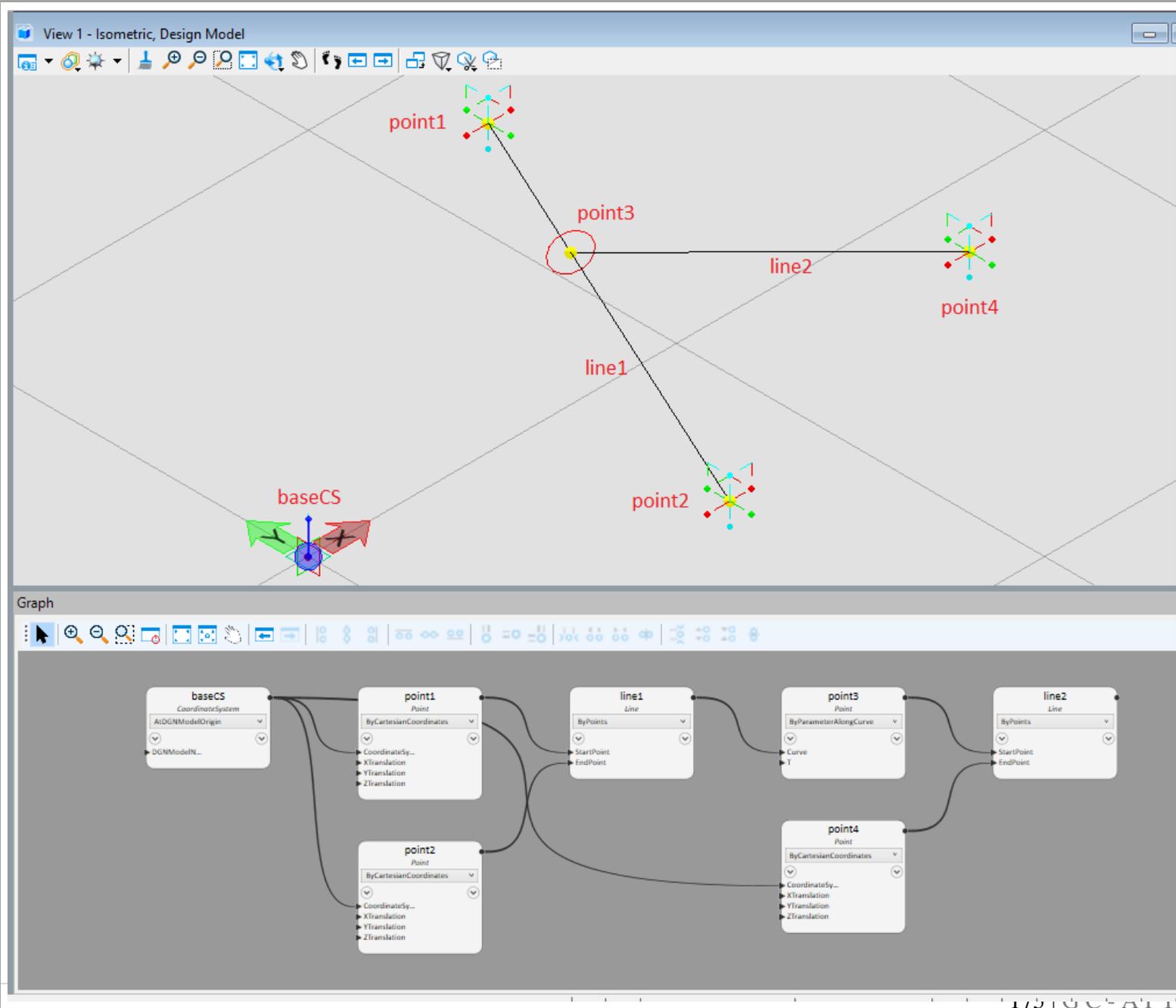
If you're not sure how to get a GCT file, here's the most common way:

- Start GC's host application normally (not in GC service mode) with an empty design file.
  - Using GC's normal tools (as a user would), build the GC model you want.
1. From GC's transaction player, select "Edit this entire transaction list", then, from GC's editor, select "Save to Transaction (GCT) File...".



- **Here's the GC model we'll use for this GC service walkthrough**

We'll use this simple model comprised only of a coordinate system, four points, and two lines.



Here's the transaction file, which I've elegantly named "Test model for GC service.gct" :

```
// Bentley GenerativeComponents Transaction File -- File structure version 1.60. (Please do not delete or change this line.)

environment
{
  GCVersion          = '99.99.99.99';
  MSVersion          = '10.00.00.99';
  MSProject          = '';
  MSDesignFile       = 'D:\GC API documentation\Test model for GC service.gct';
  MSMasterUnit       = {Meter, 'mu', Metric, 1.0, 1.0};
  MSSubUnit          = {Meter, 'mm', Metric, 1.0, 1000.0};
  MSSStorageUnit     = {Meter, '', Metric, 1.0, 1.0};
  MSUorsPerStorageUnit = 10000.0;
}

transaction 1 stateChange 'Add baseCS'
{
  gcModel
  {
    node User.Objects.baseCS Bentley.GC.NodeTypes.CoordinateSystem
    {
      Technique          = 'AtDGNModelOrigin';
      DGNModelName       = 'Design Model';
      SymbolSize         = 1.0;
      GraphLocation      = <auto> {40.0, 40.0, 0.0, 118.59};
    }
  }
}

transaction 2 stateChange 'Add point1, point2'
{
  gcModel
  {
    node User.Objects.point1 Bentley.GC.NodeTypes.Point
    {
      Technique          = 'ByCartesianCoordinates';
      CoordinateSystem   = baseCS;
      XTranslation        = <free> 12.6108737631947;
      YTranslation        = <free> 7.71973665348877;
    }
  }
}
```

```

        ZTranslation          = <free> 0.0;
        GraphLocation         = <auto> {314.0, 40.0, 0.0, 158.87};
    }
    node User.Objects.point2 Bentley.GC.NodeTypes.Point
    {
        Technique              = 'ByCartesianCoordinates';
        CoordinateSystem       = baseCS;
        XTranslation           = <free> 7.02123118305114;
        YTranslation           = <free> -4.42700554680833;
        ZTranslation           = <free> -1.11022302462516E-16;
        GraphLocation          = <auto> {314.0, 238.87, 0.0, 158.87};
    }
}

transaction 3 stateChange 'Add line1'
{
    gcModel
    {
        node User.Objects.line1 Bentley.GC.NodeTypes.Line
        {
            Technique          = 'ByPoints';
            StartPoint         = point1;
            EndPoint           = point2;
            GraphLocation      = <auto> {588.0, 40.0, 0.0, 132.017};
        }
    }
}

transaction 4 stateChange 'Add point3, point4'
{
    gcModel
    {
        node User.Objects.point3 Bentley.GC.NodeTypes.Point
        {
            Technique          = 'ByParameterAlongCurve';
            Curve              = line1;
            T                  = <free> 0.341482906359958;
            GraphLocation      = <auto> {862.0, 40.0, 0.0, 132.017};
        }
        node User.Objects.point4 Bentley.GC.NodeTypes.Point
        {
            Technique          = 'ByCartesianCoordinates';

```

```
        CoordinateSystem      = baseCS;
        XTranslation          = <free> 16.1313207376944;
        YTranslation          = <free> -1.83968739214067;
        ZTranslation          = <free> 0.0;
        GraphLocation         = <auto> {862.0, 212.017, 0.0, 158.87};
    }
}

transaction 5 stateChange 'Add line2'
{
    gcModel
    {
        node User.Objects.line2 Bentley.GC.NodeTypes.Line
        {
            Technique          = 'ByPoints';
            StartPoint         = point3;
            EndPoint           = point4;
            GraphLocation      = <auto> {1136.0, 40.0};
        }
    }
}
```

## 6.3.5. Loading and controlling the transaction file (part 1)

Back in our SimpleGCServiceClient app in Visual Studio, let's revise our Program class to look like this:

```
internal class Program
{
    static GCServiceClient gcService = new GCServiceClient();

    static void Main(string[] args)
    {
        gcService.Ping();
        // PerformCalculations(); // Comment out this statement.
        LoadGCModel();
    }

    static void PerformCalculations()
    {
        :
    }

    static void DisplayResults(AnnotatedLabeledNumbers results)
    {
        :
    }

    static void LoadGCModel()
    {
        // Replace the following file path with the full path to your own transaction file.
        const string transactionFilePath = @"D:\GC API Documentation\Test model for GC service.gct";

        gcService.LoadTransactionFile(transactionFilePath);
        gcService.PlayTransactions(100);
        gcService.FitViews(true);
    }
}
```

◆ If you try running this program as it stands, you'll discover it doesn't quite work as expected. We'll explain why in [part 2](#).

As you can see, our new method, `LoadGCModel()`, comprises calls to these methods on the GC service:

- `LoadTransactionFile(string fileAbsolutePath)`

Instructs GC to load the specified transaction (GCT) file, as though the user had opened it manually.

If a GC model is already present, it's discarded before loading the transaction file.

- `PlayTransactions(int lastTransactionNumberToPlay)`

The method 'LoadTransactionFile' loads a transaction file but doesn't play any of its transactions. This method, 'PlayTransactions', instructs GC to play one or more of those transactions.

The argument specifies the last transaction number you want GC to play. (The first transaction is number 1.) If that transaction has already been played, nothing happens. Otherwise, GC plays the transactions up to and including that transaction number.

If you specify a transaction number that's beyond the end of the transaction file, GC plays the entire transaction file. In this case, since we want GC to play the entire file, we specify a number (100) that we know is beyond the end.

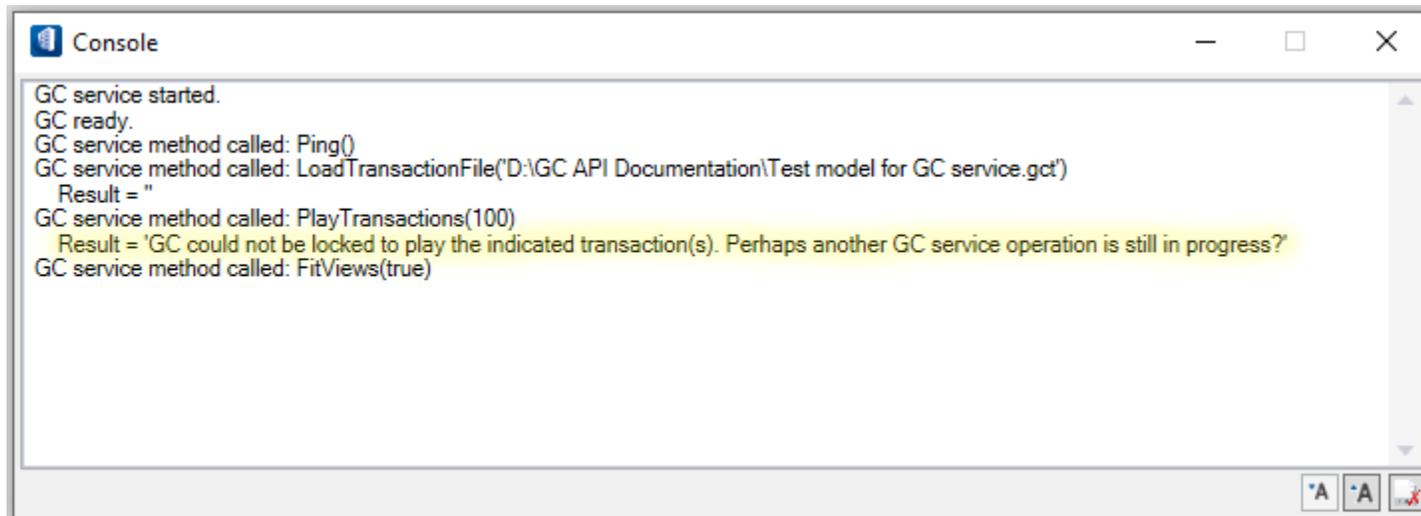
- `FitViews(bool turnOnAutoRefit)`

This method does not affect GC's operation, but it facilitates our observation of the GC model.

This method fits all the graphical views as well as GC's graph. Furthermore, if we pass 'true' for 'turnOnAutoRefit', those views will automatically re-fit whenever GC's model changes.

## 6.3.6. Loading and controlling the transaction file (part 2)

If you try running the Program shown in [part 1](#), you'll discover it doesn't quite work as expected: If we switch back to OBD/GC to see the results, we see this:



```
Console
GC service started.
GC ready.
GC service method called: Ping()
GC service method called: LoadTransactionFile('D:\GC API Documentation\Test model for GC service.gct')
Result = "
GC service method called: PlayTransactions(100)
Result = 'GC could not be locked to play the indicated transaction(s). Perhaps another GC service operation is still in progress?'
GC service method called: FitViews(true)
```

The problem is that after we call the method `gcService.LoadTransactionFile(transactionFilePath)`, we need to give GC a bit of time to complete that loading of the file before we call any other GC service method.

In most cases, waiting for one second will be sufficient, though you may need to experiment with that value for your transaction file(s).

Here's our Program method LoadGCModel with a one-second delay inserted:

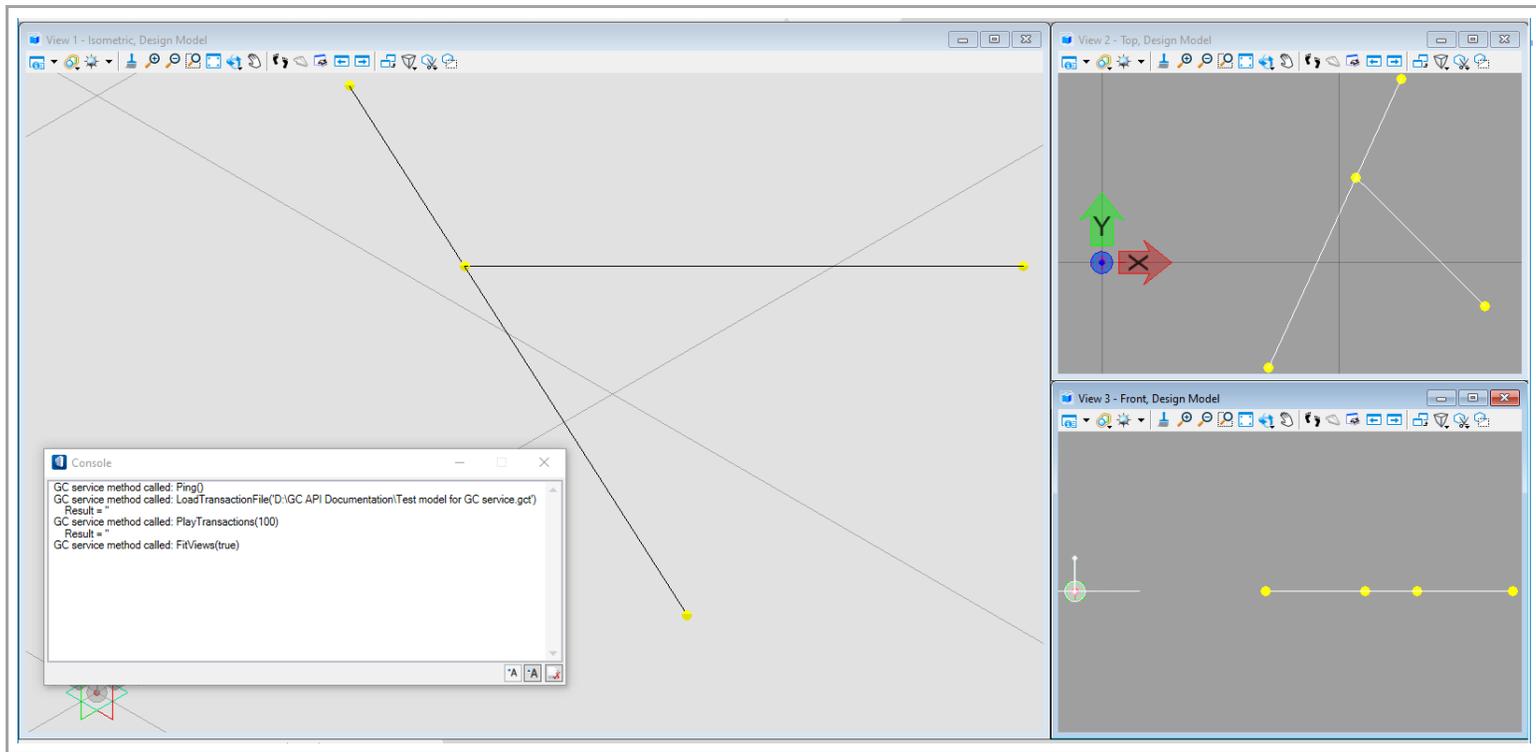
```
static void LoadGCModel()
{
    const string transactionFilePath = @"D:\GC API Documentation\Test model for GC service.gct";

    gcService.LoadTransactionFile(transactionFilePath);
    Thread.Sleep(1000); // Wait one second while GC loads the transaction file.
    gcService.PlayTransactions(100);
    gcService.FitViews(turnOnAutoRefit:true);
}
```

◆ The static class 'Thread' resides in the namespace System. Threading. So, you'll also need to add the following statement among the 'using' statements at the top of the Program.cs:

```
using System. Threading;
```

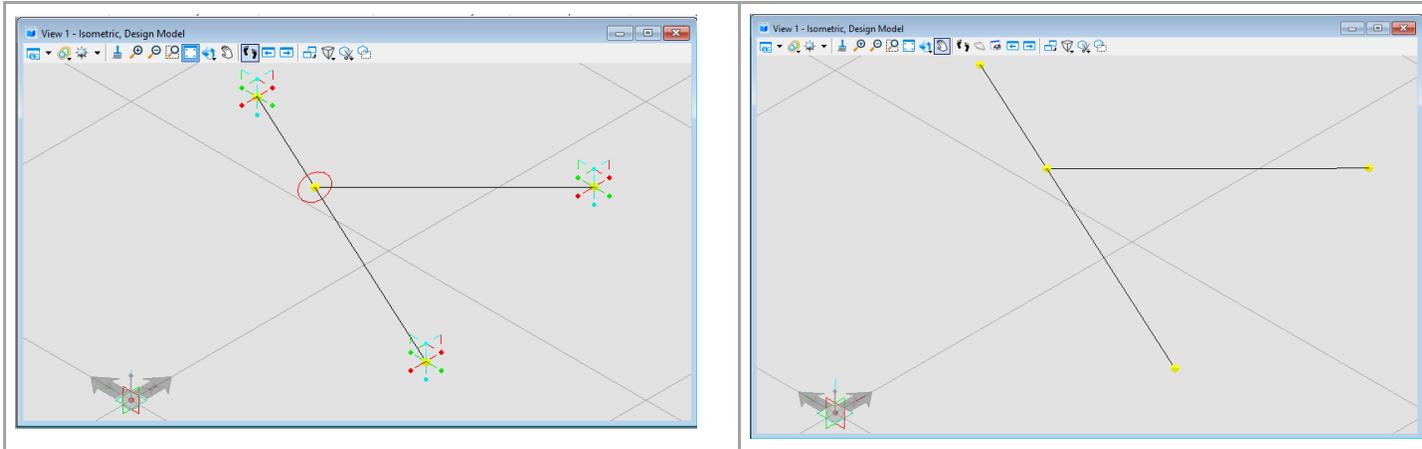
And now, if we re-run SimpleGCServiceClient, GC should show the expected results:



◆ Why do the points look different?

As produced directly within GC, by the user:

As produced through GC's service interface:



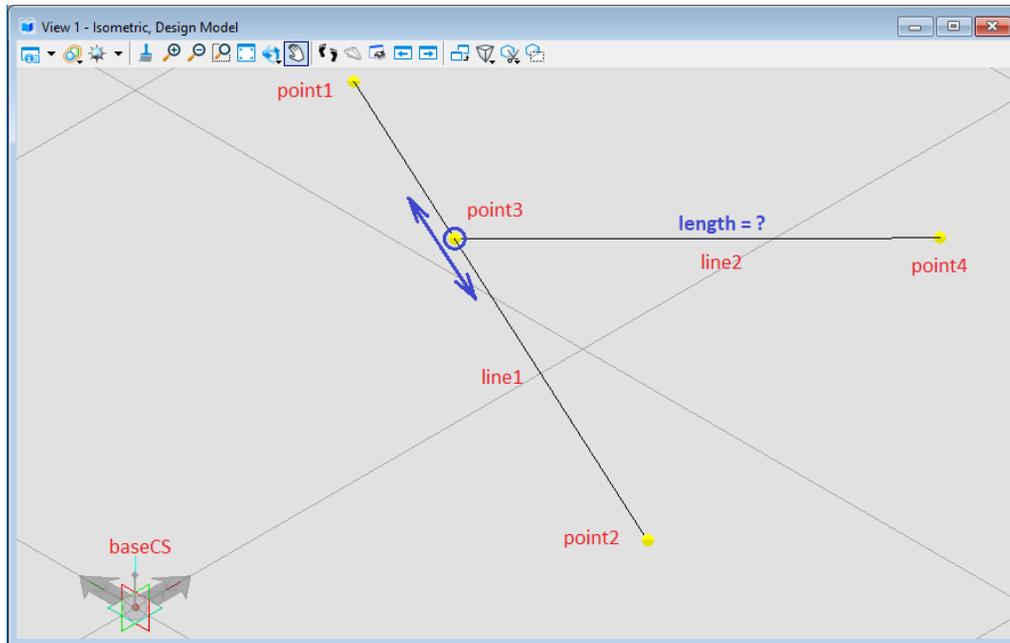
When GC performs transactions through its service interface -- e.g., `gcService.PlayTransactions(100)` -- it automatically ignores the "free" indicators on node inputs. Therefore, none of the resultant points have "handles" (places where the user can use the mouse to grab the point and move it around).

Of course, there's no reason why those points should have "handles", since a user can't directly manipulate those points, anyway, when GC is in service mode. Everything is under the control of the GC service client application (in this case, `SimpleGCServiceClient`).

### 6.3.7. Manipulating the GC model

Great, we've loaded a GC model through GC's service interface! Now what?

We can manipulate the GC model for analytic purposes. For example, suppose we want to find out how the length of 'line2' changes as we move 'point3' along 'line1'.



Currently, point3 has a 'T' value of about 0.34 -- that is, it's located about 34% of the way from point1 to point2. We know that by looking into the transaction file we're providing to the GC service:

```
transaction 4 stateChange 'Add point3, point4'
{
  gcModel
  {
    node User.Objects.point3 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByParameterAlongCurve';
      Curve               = line1;
      T                   = <free> 0.341482906359958;
      GraphLocation       = <auto> {862.0, 40.0, 0.0, 132.017};
    }
    node User.Objects.point4 Bentley.GC.NodeTypes.Point
    {
      Technique           = 'ByCartesianCoordinates';
      CoordinateSystem    = baseCS;
      XTranslation         = <free> 16.1313207376944;
      YTranslation        = <free> -1.83968739214067;
      ZTranslation        = <free> 0.0;
      GraphLocation       = <auto> {862.0, 212.017, 0.0, 158.87};
    }
  }
}
```

Specifically, we're now going to answer the question:

**If we change point3's 'T' value to 0.8, what will be the resultant length of the line2?**

Here's our next (and, actually, final) version of the Program.cs:

```
internal class Program
{
    static GCServiceClient gcService = new GCServiceClient();

    static void Main(string[] args)
    {
        gcService.Ping();
        // PerformCalculations();
        LoadGCModel();
        ExerciseGCModel();
    }

    static void PerformCalculations()
    {
        string[] scriptExpressions = { "2 + 3", "Sqrt(9)", "new DVector2d(3, 4).Magnitude" };

        AnnotatedLabeledNumbers results = gcService.GetNumericValues("", scriptExpressions);

        DisplayResults(results);
    }

    static void DisplayResults(AnnotatedLabeledNumbers results)
    {
        foreach (LabeledNumber result in results.LabeledNumbers)
            Console.WriteLine($"{result.Label,-30} = {result.Number}");

        Console.WriteLine();
        Console.WriteLine("Finished. Press [Enter] to close. ");
        Console.ReadLine();
    }

    static void LoadGCModel()
    {
        // Replace the following file path with the full path to your own transaction file.
        const string transactionFilePath = @"D:\GC API Documentation\Test model for GC service.gct";

        gcService.LoadTransactionFile(transactionFilePath);
        Thread.Sleep(1000); // Wait one second while GC loads the transaction file.
        gcService.PlayTransactions(100);
        gcService.FitViews(turnOnAutoRefit:true);
    }
}
```

```

}
static void ExerciseGCModel()
{
    LabeledNumber[] propertyValues = { new LabeledNumber() {Label="point3.T", Number=0.8} };

    gcService.ApplyNodePropertyNumericValues("", propertyValues);
    gcService.UpdateGCModel("");

    string[] scriptExpressions = { "line2.Length" };

    AnnotatedLabeledNumbers results = gcService.GetNumericValues("", scriptExpressions);
    DisplayResults(results);
}
}

```

As you can see, our new method, `ExerciseGCModel()`, comprises calls to these methods on the GC service:

- `ApplyNodePropertyNumericValues(string gcModelName, LabeledNumber[] propertyValues)`

Assigns the specified values to the specified node properties (inputs).

[Remember](#) that, for any service method call that takes a `gcModelName` as its first parameter, you can simply pass an empty string ("" ) as that corresponding argument.

The property/value pairs are provided in an array of one or more [LabeledNumber](#) instances. In this case, this statement:

```
LabeledNumber[] propertyValues = { new LabeledNumber() {Label="point3.T", Number=0.8} };
```

...defines an array comprised of a single `LabeledNumber` in which the property is "point3.T" and the value is 0.8.

◆ Note that the delimiters surrounding `Label="point3.T", Number=0.8` are curly braces, not parentheses.

```
... {Label="point3.T", Number=0.8} ... ; // Curly braces, not parentheses.
```

This is a standard way in C# to assign an object's property values immediately after the object -- in this case, `new LabeledNumber()` -- has been created.

- `UpdateModel(string gcModelName)`

Assigning new property values, as we did in the preceding method call, does merely that: Assigns new property values but doesn't react to those new values.

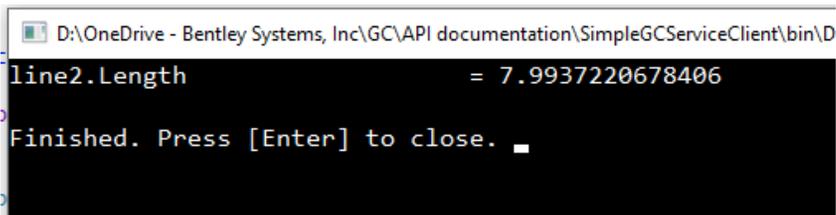
After we make one or more calls to `ApplyNodePropertyNumericValues(...)`, we typically call `UpdateModel("")` to propagate any changes resulting from those new node values, such as (in this case) 'line2' moving because one of its anchor points, 'point3', has moved.

- `GetNumericValues(string gcModelName, string[] scriptExpressions)`

This is precisely the same method we called [earlier](#) to get the results of arbitrary calculations. Now, we're using it to get the resultant length of the line2 ("`line2.Length`").

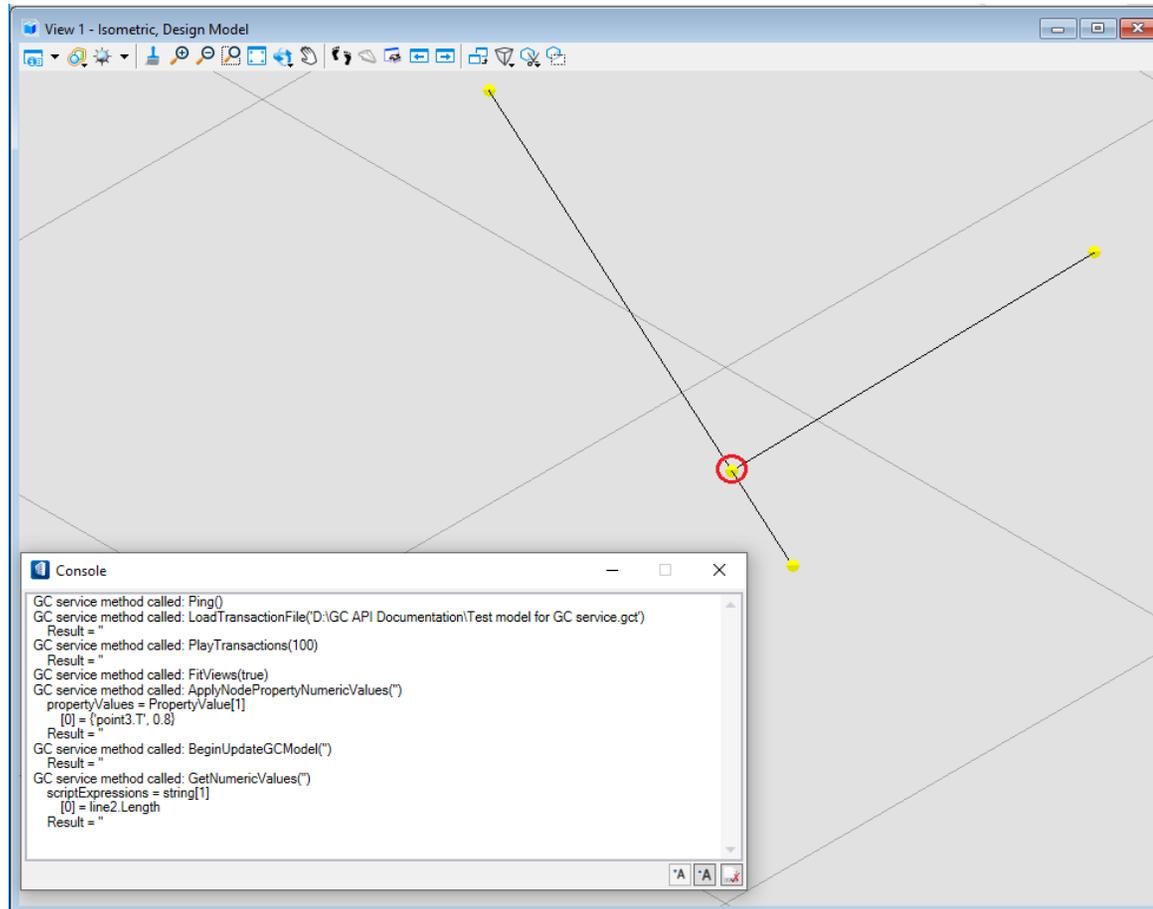
And finally, we call `DisplayResults` -- a method within our `Program` class -- to display the results, as we did [earlier](#).

Here is the result of running our new version of `SimpleGCServiceClient`:



```
D:\OneDrive - Bentley Systems, Inc\GC\API documentation\SimpleGCServiceClient\bin\D
line2.Length = 7.9937220678406
Finished. Press [Enter] to close. █
```

And here's how things look on the OBD/GC side. We can see that, as expected, the point has been moved.



## 6.4. The GC service interface (reference)

Remember, wherever the parameter "string gcModelName" appears, you can simply pass an empty string ("").

Many of these methods return a string. Normally, that resultant string is empty, but, if an error has occurred on GC's side, the result will contain that error message. (It will be the same error message you'll see in GC's console panel.)

- **string ApplyNodePropertyNumericValues(string gcModelName, [LabeledNumber\[\]](#) propertyValues)**

Assigns the specified numeric values to the specified node properties (inputs).

Each LabeledNumber corresponds to one property assignment. The LabeledNumber's 'Label' contains a script representation of the property to be assigned (such as "point3.T") and its 'Number' contains the value to be assigned to that property (such as 0.8).

To assign non-numeric data to a node, use one of the methods CreateOrChangeNodes or CreateOrChangeNodesFromFile (below).

- **string ClearGCModel(string gcModelName)**

Clears the GC model, removing all GC nodes and their corresponding elements.

- **string CreateOrChangeNodes(string gcModelName, bool firstlyClearGCModel, string nodeDescriptions)**

Process the given node "descriptions" as though they're being played in the context of a transaction. A node description looks like this:

|                                                                                                                                                                                                                       |                                                                                                                                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><i>General form:</i></p> <pre>node &lt;fullyQualifiedNodeName&gt; &lt;fullyQualifiedNodeType&gt; {   &lt;propertyName&gt; = &lt;inputExpression&gt;;   &lt;propertyName&gt; = &lt;inputExpression&gt;;   : }</pre> | <p><i>Example:</i></p> <pre>node User.Objects.point1 Bentley.GC.NodeTypes.Point {   Technique           = 'ByCartesianCoordinates';   CoordinateSystem    = baseCS;   XTranslation         = 12.6;   YTranslation         = 7.7; }</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

You can build the node descriptions programmatically within your GC service client app. Here's a (contrived) example:

```
static void ExampleOfCreateOrChangeNodes()
{
    string nodeDescriptions = getNodeDescriptions(12.6, 7.7, 7.0, -4.4);
    gcService.CreateOrChangeNodes("", true, nodeDescriptions);

    string getNodeDescriptions(double x1, double y1, double x2, double y2)
    {
        // The proverbial line by two points.

        StringBuilder builder = new StringBuilder();
        builder.AppendLine("node User.Objects.baseCS Bentley.GC.NodeTypes.CoordinateSystem");
        builder.AppendLine("{");
        builder.AppendLine("    Technique          = 'AtDGNModelOrigin'");
        builder.AppendLine("    DGNModelName      = 'Design Model'");
        builder.AppendLine("}");
        builder.AppendLine("node User.Objects.point1 Bentley.GC.NodeTypes.Point");
        builder.AppendLine("{");
        builder.AppendLine("    Technique          = 'ByCartesianCoordinates'");
        builder.AppendLine("    CoordinateSystem  = baseCS");
        builder.AppendLine($"    XTranslation       = {x1}");
        builder.AppendLine($"    YTranslation       = {y1}");
        builder.AppendLine("}");
        builder.AppendLine("node User.Objects.point2 Bentley.GC.NodeTypes.Point");
        builder.AppendLine("{");
        builder.AppendLine("    Technique          = 'ByCartesianCoordinates'");
        builder.AppendLine("    CoordinateSystem  = baseCS");
        builder.AppendLine($"    XTranslation       = {x2}");
        builder.AppendLine($"    YTranslation       = {y2}");
        builder.AppendLine("}");
        builder.AppendLine("node User.Objects.line1 Bentley.GC.NodeTypes.Line");
        builder.AppendLine("{");
        builder.AppendLine("    Technique          = 'ByPoints'");
        builder.AppendLine("    StartPoint         = point1");
        builder.AppendLine("    EndPoint           = point2");
        builder.AppendLine("}");
        return builder.ToString();
    }
}
```

On a smaller scale, you can call `CreateOrChangeNodes` to tweak the nodes of an existing GC model. For example:

```
string nodeDescription = "node User.Objects.line1 Bentley.GCNodeTypes.Line { StartPoint = point3; }";
gcService.CreateOrChangeNodes("", false, nodeDescription);
```

- **string `CreateOrChangeNodesFromFile(string gcModelName, bool firstlyClearGCModel, string fileAbsolutePath)`**

This method behaves the same as the preceding method (`CreateOrChangeNodes`), but the node descriptions are obtained from a text file rather than being built programmatically. (Of course, that file, itself, may have been built programmatically in a previous phase of your app.)

The text file contains a sequence of node descriptions, just like the `StringBuilder` string we built in the previous example.

- **void `FitViews(bool turnOnAutoRefit)`**

This method does not affect GC's operation, but it facilitates our observation of the GC model.

This method fits all the graphical views as well as GC's graph. Furthermore, if we pass 'true' for 'turnOnAutoRefit', those views will automatically re-fit whenever GC's model changes.

- **[AnnotatedLabeledNumbers](#) `GetNumericValues(string gcModelName, string[] scriptExpressions)`**

This method evaluates the given script expressions, which are expected to evaluate numbers. For each expression, this method returns a `LabeledNumber` in which the 'Label' is a copy of the expression and the 'Value' is the result of evaluating that expression.

If all goes well, the 'Annotation' property of the resultant `AnnotatedLabeledNumbers` will be empty; otherwise, that property will contain an error message.

- **string `LoadTransactionFile(string fileAbsolutePath)`**

Instructs GC to load the specified transaction (GCT) file, as though the user had opened it manually.

If a GC model is already present, it's discarded before loading the transaction file.

The file is loaded on OBD/GC's side, using the specified file path. If that's not practical -- e.g., if OBD/GC is running on a different computer than your GC service client -- then use the following method, `LoadTransactionFileContents`, instead.

- **`string LoadTransactionFileContents(string transactionFilePath, string transactionFileContents)`**

◆ The interface definition of this method has a bug, in that the parameters are mislabeled. That definition, which is wrong, appears as:

```
string LoadTransactionFileContents(string transactionFileContents, string homeFolderAbsolutePath) // Liar!
```

Instead, refer to this definition (which is also shown above this box):

```
string LoadTransactionFileContents(string transactionFilePath, string transactionFileContext) // Correct.
```

The given 'transactionFileContents' is the complete contents of a transaction (GCT) file. Use this method instead of the preceding method (`LoadTransactionFile`) when OBD/GC is running on a different computer and therefore can't access any local files on the current computer. Here's an example of its use:

```
string transactionFilePath = @"..."; // Complete path to local transaction file.  
string transactionFileContents = File.ReadAllText(transactionFilePath); // System.IO.File  
gcService.LoadTransactionFileContents(transactionFilePath, transactionFileContents);
```

- **`void Ping()`**

This method merely "nudges" the GC service to ensure that it's up and running. An exception will be thrown if it's not.

- **`string PlayTransactions(int lastTransactionNumberToPlay)`**

The method 'LoadTransactionFile' loads a transaction file but doesn't play any of its transactions. This method, 'PlayTransactions', instructs GC to play one or more of those transactions.

The argument specifies the last transaction number you want GC to play. (The first transaction is number 1.) If that transaction has already been played, nothing happens. Otherwise, GC plays the transactions up to and including that transaction number.

If you specify a transaction number that's beyond the end of the transaction file, GC plays the entire transaction file. In this case, since we want GC to play the entire file, we specify a number (100) that we know is beyond the end.

- **string UpdateGCModel(string gcModelName)**

Updates entire the GC model, as though the user had clicked the "Update Model" button. Generally, you should call this method after having changed one or more property values by calling any of the methods `ApplyNodePropertyNumericValues`, `CreateOrChangeNodes`, or `CreateOrChangeNodesFromFile`.

## 7.Code Samples

## 7.1. Calculated properties

### 7.1.1. SimpleLineNodeWithCalculatedLength.cs

This class is a modified version of our standard sample class, [SimpleLineNode.cs](#). This class contains additions (highlighted below) that, collectively, implement a calculated property named Length. That process is explained [here](#).

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using Bentley.GenerativeComponents;
using Bentley.GenerativeComponents.AddInSupport;
using Bentley.GenerativeComponents.ElementBasedNodes;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.GCTypes;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.MicroStation;
using Bentley.GenerativeComponents.View;
using Bentley.DgnPlatformNET;
using Bentley.DgnPlatformNET.Elements;
using Bentley.GeometryNET;
using Bentley.MstnPlatformNET;

namespace MyAddIn
{
    [GCNamespace("User")]
    [GCNodeTypePaletteCategory("My Add-In")]
    [GCNodeTypeIcon("Resources/SimpleLineNode.png")]
    public class SimpleLineNodeWithCalculatedLength: GeometricNode
    {
        [GCDefaultTechnique]
        [GCSummary("Creates a line between a given start and end point.")]
        [GCParameter("StartPoint", "Start point of the line.")]
        [GCParameter("EndPoint", "End point of the line.")]
    }
}
```

```

public NodeUpdateResult ByPoints
(
    NodeUpdateContext                updateContext,
    [GCDgnModelProvider, GCReplicable] IPointNode StartPoint,
    [GCReplicable] IPointNode        EndPoint
)
{
    if (StartPoint == EndPoint)
        return new NodeUpdateResult.TechniqueInvalidArguments(nameof(StartPoint), nameof(EndPoint));

    DPoint3d startPt = StartPoint.GetDPoint3d();
    DPoint3d endPt   = EndPoint.GetDPoint3d();

    DPoint3d uorStartPt = NativeDgnTools.FromMUToUOR(startPt);
    DPoint3d uorEndPt   = NativeDgnTools.FromMUToUOR(endPt);

    LineElement lineElement = new LineElement(GCDgnModel().DgnModel(), TemplateElement(),
  new DSegment3d(uorStartPt, uorEndPt));

    SetElement(lineElement);

    return NodeUpdateResult.Success;
}

[GCTechnique, GCUnselectable]
public NodeUpdateResult DefaultSimpleLineTechnique(NodeUpdateContext updateContext, double Length)
{
    return NodeUpdateResult.TechniqueGenericFailure;
}

[GCPropertyCalculator("Length")]
public void LengthCalculator(DependeeReevaluationScope effectOnParentNode)
{
    if (Element() is LineElement lineElement)
    {
        CurveVector curveVector = lineElement.GetCurveVector();

        curveVector.GetStartEnd(out DPoint3d uorStartPt, out DPoint3d uorEndPt);

        DPoint3d startPt = NativeDgnTools.FromUORToMU(uorStartPt);
        DPoint3d endPt   = NativeDgnTools.FromUORToMU(uorEndPt);

        double length = startPt.Distance(endPt);
    }
}

```

```
    }  
    }  
    }  
}
```

## 7.2. Comparison of element-based node and utility node

For comparison, this sub-section contains the same (functionally equivalent) node class implemented as both an element-based node and a utility node.

### 7.2.1. BioChamberElementBasedNode.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.Linq;
using Bentley.DgnPlatformNET;
using Bentley.DgnPlatformNET.Elements;
using Bentley.GenerativeComponents;
using Bentley.GenerativeComponents.AddInSupport;
using Bentley.GenerativeComponents.ElementBasedNodes;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.GCTypes;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.GCScript.ReflectedNativeTypeSupport;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.MicroStation;
using Bentley.GenerativeComponents.ScriptEditor;
using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;
using Bentley.GenerativeComponents.UtilityNodes;
using Bentley.GenerativeComponents.View;
using Bentley.GeometryNET;
using Bentley.MstnPlatformNET;

namespace MyAddIn
{
    using EECC = ExpressionEditorCustomConfiguration;
    using EECCAttr = GCExpressionEditorCustomConfigurationAttribute;

    [GCNamespace("User")]
    [GCNodeTypePaletteCategory("My Add-In")]
    [GCNodeTypeIcon("Resources/BioChamberNode.png")]
}
```

```

[GCSummary("Essentially, a high-tech animal cage for long-term care.")]
public class BioChamberElementBasedNode: ElementBasedNode
{
    static EECC GetEECCForAnimal()
    {
        return new EECC(getScriptChoices);

        IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
        {
            BioChamberElementBasedNode node = (BioChamberElementBasedNode) parentExpressionEditor.MeaningOfThis;

            ScriptChoiceList result = node.GetGroupedAnimals();

            return result;
        }
    }

    ScriptChoiceList GetGroupedAnimals()
    {
        // Simulate determining the animals at runtime, based on the current state of this node.
        // (For this example, we're simply hard-coding the result.)

        string[] mammals      = {"Bat", "Elephant", "Squirrel", "Whale"};
        string[] fish         = {"Fish", "Eel", "Minnow", "Salmon", "Shark"};
        string[] birds        = {"Chicken", "Falcon", "Owl", "Penguin"};
        string[] reptiles     = {"Crocodile", "Lizard", "Snake", "Tortoise"};
        string[] amphibians   = {"Frog", "Newt", "Salamander", "Toad"};
        string[] invertebrates = {"Earthworm", "Insect", "Jellyfish", "Spider"};

        ScriptChoiceList result = new ScriptChoiceList();

        addGroup("Mammals",      mammals);
        addGroup("Fish",        fish);
        addGroup("Birds",       birds);
        addGroup("Reptiles",    reptiles);
        addGroup("Amphibians",  amphibians);
        addGroup("Invertebrates", invertebrates);

        return result;

        void addGroup(string groupName, string[] animals)
        {
            ScriptChoice groupChoice = result.Add(groupName);

```

```

        foreach (string animal in animals)
            groupChoice.AddSubChoice(animal.ToQuotedScriptText());
    }
}

[GCDefaultTechnique]
[GCSummary("Creates a bio-chamber from a given anchor point and other criteria.")]
[GCPParameter("AnchorPoint", "Anchor point of the bio-chamber.")]
[GCPParameter("CubicMeters", "Interior volume of this bio-chamber.")]
[GCPParameter("Animal", "The type of animal this bio-chamber is designed for.")]
[GCPParameter("CaretakerNotes", "A file that contains caretaking instructions.")]
[GCPParameter("IsAnimalCuddly", "Whether this type of animal is nice to cuddle with.")]
public NodeUpdateResult Default
(
    NodeUpdateContext                updateContext,
    [GCDgnModelProvider] IPointNode  AnchorPoint,
    [GCInitialValue(60)] double      CubicMeters,
    [EECCAttr(nameof(GetEECCForAnimal))] string Animal,
    [GCOptional, GCInitiallyPinned,
     GCFileBrowser(fileFilterDescription:"Text files",
                    fileFilterMask:"*.txt")]
    string                            CaretakerNotes,
    [GCOut] ref bool                  IsAnimalCuddly
)
{
    // Do whatever this method does with the given inputs.

    IsAnimalCuddly = Animal == "Cat" || Animal == "Dog"; // Or however this method
   // calculates this value.

    return NodeUpdateResult.Success;
}
}
}

```

## 7.2.2. BioChamberUtilityNode.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.Linq;
using Bentley.DgnPlatformNET;
using Bentley.DgnPlatformNET.Elements;
using Bentley.GenerativeComponents;
using Bentley.GenerativeComponents.AddInSupport;
using Bentley.GenerativeComponents.ElementBasedNodes;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.GCTypes;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.GCScript.ReflectedNativeTypeSupport;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.MicroStation;
using Bentley.GenerativeComponents.ScriptEditor;
using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;
using Bentley.GenerativeComponents.UtilityNodes;
using Bentley.GenerativeComponents.View;
using Bentley.GeometryNET;
using Bentley.MstnPlatformNET;

namespace MyAddIn
{
    using EECC = ExpressionEditorCustomConfiguration;

    [GCNamespace("User")]
    [GCNodeTypePaletteCategory("My Add-In")]
    [GCNodeTypeIcon("Resources/BioChamberNode.png")]
    [GCSummary("Essentially, a high-tech animal cage for long-term care.")]
    public class BioChamberUtilityNode: UtilityNode
    {
        static EECC GetEECCForAnimal(INodeBase nodeBase)
        {
            return new EECC(getScriptChoices);

            IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
            {
```

```

        BioChamberUtilityNode bioChamber = (BioChamberUtilityNode) nodeBase;

        ScriptChoiceList result = bioChamber.GetGroupedAnimals();

        return result;
    }
}

static public EECC GetEECCForCaretakerNotes(INodeBase nodeBase)
{
    FileBrowserConfiguration configuration = FileBrowserConfiguration.Get(fileFilterDescription:"Text files",
fileFilterMask:"*.txt");
    return configuration.GetEECC(nodeBase);
}

static void AddAdditionalMembersToGCType(IGCEnvironment environment, GCType gcType,
NativeNamespaceTranslator namespaceTranslator)
{
    UtilityNodeTechnique technique = gcType.AddDefaultNodeTechnique(nameof(Default), Default,
  Ls.Literal("Creates a bio-chamber from given
criteria."));

    technique.AddParameter(environment, nameof(AnchorPoint), typeof(DPoint3d), "",
Ls.Literal("Anchor point of this bio-chamber."));

    technique.AddParameter(environment, nameof(CubicMeters), typeof(double), "60",
Ls.Literal("Interior volume of this bio-chamber."))
        .SetIsInitiallyPinned(PropertyDirections.Input);

    technique.AddParameter(environment, nameof(Animal), typeof(string), default(string),
Ls.Literal("The type of animal this bio-chamber is designed for."))
        .SetGetEECC("Default", GetEECCForAnimal);

    technique.AddParameter(environment, nameof(CaretakerNotes), typeof(string), default(string),
Ls.Literal("A file that contains caretaking instructions."),
NodePortRole.TechniqueOptionalInput)
        .SetGetEECC(nameof(Default), GetEECCForCaretakerNotes);

    technique.AddParameter(environment, nameof(IsAnimalCuddly), typeof(bool), default(string),
Ls.Literal("Whether this type of animal is nice to cuddle with."),
NodePortRole.TechniqueOutputOnly);
}
}

```

```

// The following static method implements this node type's one-and-only technique, "Default".
// Every technique method implementation must have this signature. (The method's name can
// be anything.)

static NodeUpdateResult Default(UtilityNode node, NodeUpdateContext updateContext)
{
    BioChamberUtilityNode bioChamber = (BioChamberUtilityNode) node; // It's safe to assume that the
  // given node is of this class type,
  // BioChamberUtilityNode.

    DPoint3d anchorPoint    = bioChamber.AnchorPoint;
    double    cubicMeters   = bioChamber.CubicMeters;
    string    animal        = bioChamber.Animal;
    string    caretakerNodes = bioChamber.CaretakerNotes;

    // Do whatever this method does with the given inputs.

    bioChamber.IsAnimalCuddly = animal == "Cat" || animal == "Dog"; // Or however this method
  // calculates this value.

    return NodeUpdateResult.Success;
}

// ===== End of static members; beginning of instance members =====

ScriptChoiceList GetGroupedAnimals()
{
    // Simulate determining the animals at runtime, based on the current state of this node.
    // (For this example, we're simply hard-coding the result.)

    string[] mammals      = {"Cat", "Dog", "Elephant", "Squirrel"};
    string[] fish         = {"Fish", "Eel", "Minnow", "Salmon", "Shark"};
    string[] birds        = {"Chicken", "Falcon", "Owl", "Penguin"};
    string[] reptiles     = {"Crocodile", "Lizard", "Snake", "Tortoise"};
    string[] amphibians   = {"Frog", "Newt", "Salamander", "Toad"};
    string[] invertebrates = {"Earthworm", "Insect", "Jellyfish", "Spider"};

    ScriptChoiceList result = new ScriptChoiceList();

    addGroup("Mammals",    mammals);
    addGroup("Fish",      fish);
    addGroup("Birds",     birds);
    addGroup("Reptiles",  reptiles);
}

```

```

addGroup("Amphibians",  amphibians);
addGroup("Invertebrates", invertebrates);

return result;

void addGroup(string groupName, string[] animals)
{
    ScriptChoice groupChoice = result.Add(groupName);
    foreach (string animal in animals)
        groupChoice.AddSubChoice(animal.ToQuotedScriptText());
}

// The following two members -- State and GetInitialState -- are simply "must be defined"
// methods that every utility node class must have. You can simply copy and paste these
// two definitions into your utility classes.

internal new NodeState State => (NodeState) base.State;

protected override UtilityNode.NodeState GetInitialState(NodeTechniqueDetermination initialActiveTechniqueDetermination)
    => new NodeState(this, initialActiveTechniqueDetermination);

// The following property associates a WPF framework element with this node type. (Typically,
// as is the case here, the framework element is a kind of WPF UserControl.) Subsequently,
// whenever this node type is instantiated, an instance of the framework element will be
// instantiated, as well, and appear on the face of the node within GC's graph.

public override Type TypeOfCustomViewContent => default; // typeof(BioChamberNodeViewContent);

// The following five "convenience" properties give us easy access, at the C# level, to
// the values of this node's technique inputs and outputs. They also allow us to use
// C#'s 'nameof' operator in various places within this class.
//
// (Furthermore, these properties can serve as binding sources for this node type's WPF layer.)

public DPoint3d AnchorPoint
{
    get => State.AnchorPointProperty.GetNativeValue<DPoint3d>();
    set => State.AnchorPointProperty.SetNativeValueAndInputExpression(value);
}

public string Animal
{

```

```

    get => State.AnimalProperty.GetNativeValue<string>();
    set => State.AnimalProperty.SetNativeValueAndInputExpression(value);
}

public string CaretakerNotes
{
    get => State.CaretakerNotesProperty.GetNativeValue<string>();
    set => State.CaretakerNotesProperty.SetNativeValueAndInputExpression(value);
}

public double CubicMeters
{
    get => State.CubicMetersProperty.GetNativeValue<double>();
    set => State.CubicMetersProperty.SetNativeValueAndInputExpression(value);
}

public bool IsAnimalCuddly
{
    get => State.IsAnimalCuddlyProperty.GetNativeValue<bool>();
    set => State.IsAnimalCuddlyProperty.SetNativeValueAndInputExpression(value);
}

public new class NodeState: UtilityNode.NodeState
{
    // There must be one NodeProperty field for each unique input and output of your node
    // type, regardless of which technique(s) each input and output belongs to. The order
    // of these fields is irrelevant, but we suggest listing them alphabetically.

    internal readonly UtilityNodeProperty AnchorPointProperty;
    internal readonly UtilityNodeProperty AnimalProperty;
    internal readonly UtilityNodeProperty CaretakerNotesProperty;
    internal readonly UtilityNodeProperty CubicMetersProperty;
    internal readonly UtilityNodeProperty IsAnimalCuddlyProperty;

    internal protected NodeState(BioChamberUtilityNode parentNode, NodeTechniqueDetermination
initialActiveTechniqueDetermination):
        base(parentNode, initialActiveTechniqueDetermination)
    {
        // IMPORTANT: This constructor calls 'AddProperty' to get each property field,
        // whereas the following constructor calls 'GetProperty'.

        AnchorPointProperty = AddProperty(nameof(AnchorPoint));
        AnimalProperty      = AddProperty(nameof(Animal));
    }
}

```

```

        CaretakerNotesProperty = AddProperty(nameof(CaretakerNotes));
        CubicMetersProperty    = AddProperty(nameof(CubicMeters));
        IsAnimalCuddlyProperty = AddProperty(nameof(IsAnimalCuddly));
    }

    protected NodeState(NodeState source): base(source)
    {
        // IMPORTANT: This constructor calls 'GetProperty' to get each property field,
        // whereas the preceding constructor calls 'AddProperty'.

        AnchorPointProperty    = GetProperty(nameof(AnchorPoint));
        AnimalProperty          = GetProperty(nameof(Animal));
        CaretakerNotesProperty = GetProperty(nameof(CaretakerNotes));
        CubicMetersProperty     = GetProperty(nameof(CubicMeters));
        IsAnimalCuddlyProperty = GetProperty(nameof(IsAnimalCuddly));
    }

    // The following two members -- UtilityNode and GetInitialState -- are simply
    // "must be defined" methods that every utility node's State class must have.
    // You can simply copy and paste these two definitions into your own utility
    // node class, changing each type name in the property UtilityNode to the name
    // of your own node class.

    protected new BioChamberUtilityNode UtilityNode => (BioChamberUtilityNode) base.UtilityNode();

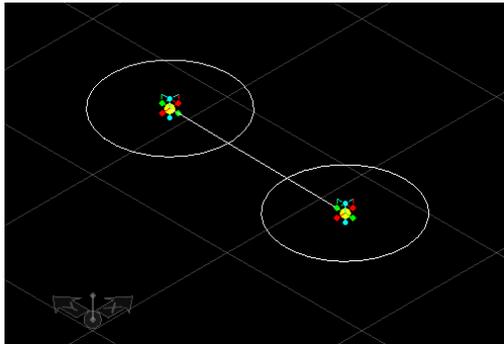
    public override UtilityNode.NodeState Clone() => new NodeState(this);
}
}
}

```

## 7.3. Constituent nodes

### 7.3.1. SimpleLineNodeWithConstituentProperties.cs

This class is a modified version of our standard sample class, [SimpleLineNode.cs](#). This class contains additions (highlighted below) that, collectively, implement two constituent nodes -- circles -- that will display as part of the line, as shown below. The process is explained [here](#).



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using Bentley.GenerativeComponents;
using Bentley.GenerativeComponents.AddInSupport;
using Bentley.GenerativeComponents.ElementBasedNodes;
using Bentley.GenerativeComponents.ElementBasedNodes.Specific; // Where CircleNode lives.
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.GCTypes;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.MicroStation;
using Bentley.GenerativeComponents.View;
using Bentley.DgnPlatformNET;
using Bentley.DgnPlatformNET.Elements;
using Bentley.GeometryNET;
using Bentley.MstnPlatformNET;
```

```

namespace MyAddIn
{
    [GCNamespace("User")]
    [GCNodeTypePaletteCategory("My Add-In")]
    [GCNodeTypeIcon("Resources/SimpleLineNode.png")]
    public class SimpleLineNodeWithConstituentProperties: GeometricNode
    {
        [GCDefaultTechnique]
        [GCSummary("Creates a line between a given start and end point.")]
        [GCParameter("StartPoint", "Start point of the line.")]
        [GCParameter("EndPoint", "End point of the line.")]
        public NodeUpdateResult ByPoints
        (
            NodeUpdateContext                updateContext,
            [GCDgnModelProvider, GCReplicable] IPointNode StartPoint,
            [GCReplicable] IPointNode        EndPoint
        )
        {
            if (StartPoint == EndPoint)
                return new NodeUpdateResult.TechniqueInvalidArguments(nameof(StartPoint), nameof(EndPoint));

            DPoint3d startPt = StartPoint.GetDPoint3d();
            DPoint3d endPt   = EndPoint.GetDPoint3d();

            DPoint3d uorStartPt = NativeDgnTools.FromMUToUOR(startPt);
            DPoint3d uorEndPt   = NativeDgnTools.FromMUToUOR(endPt);

            LineElement lineElement = new LineElement(GCDgnModel().DgnModel(), TemplateElement(),
                new DSegment3d(uorStartPt, uorEndPt));

            SetElement(lineElement);

            // Create and add the two constituent nodes.
            DeleteConstituentNodes(updateContext);

            double circleRadius = startPt.Distance(endPt) / 3; // Arbitrary radius.
            IPlaneNode support = GCMModel().ActivePlane(); // Arbitrary support plane.
            {
                CircleNode circle = CreateNodeThatWillBecomeAConstituentOfThis<CircleNode>();
                circle.LatestUpdateResult = circle.ByCenterRadius(updateContext, StartPoint, circleRadius, support, false);
            }
        }
    }
}

```

```
        AddConstituentNode(circle);
    }
    {
        CircleNode circle = CreateNodeThatWillBecomeAConstituentOfThis<CircleNode>();
        circle.LatestUpdateResult = circle.ByCenterRadius(updateContext, EndPoint, circleRadius, support, false);
        AddConstituentNode(circle);
    }
    return NodeUpdateResult.Success;
}
}
```

## 7.4. Custom expression editors, part 1

### 7.4.1. BioChamberNode.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.IO;
using System.Linq;
using Bentley.DgnPlatformNET;
using Bentley.DgnPlatformNET.Elements;
using Bentley.GenerativeComponents;
using Bentley.GenerativeComponents.AddInSupport;
using Bentley.GenerativeComponents.ElementBasedNodes;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.GCTypes;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.MicroStation;
using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;
using Bentley.GenerativeComponents.View;
using Bentley.GeometryNET;
using Bentley.MstnPlatformNET;

namespace MyAddIn
{
    using EECC = ExpressionEditorCustomConfiguration;
    using EECCAttr = GCExpressionEditorCustomConfigurationAttribute;

    [GCNamespace("User"), GCNodeTypePaletteCategory("My Add-In")]
    public class BioChamberNode: ElementBasedNode
    {
        static EECC GetEECCForAnimal()
        {
            return new EECC(getScriptChoices);

            IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
            {
                BioChamberNode node = (BioChamberNode) parentExpressionEditor.MeaningOfThis;
            }
        }
    }
}
```

```

        ScriptChoiceList result = node.GetGroupedAnimals();

        return result;
    }
}

static EECC GetEECCForDailySnacks()
{
    return new EECC(typeof(BioChamberDailySnacksEditor));
}

ScriptChoiceList GetGroupedAnimals()
{
    // Simulate determining the animals at runtime, based on the current state of this node.
    // (For this example, we're simply hard-coding the result.)

    string[] mammals      = {"Bat", "Elephant", "Squirrel", "Whale"};
    string[] fish         = {"Fish", "Eel", "Minnow", "Salmon", "Shark"};
    string[] birds        = {"Chicken", "Falcon", "Owl", "Penguin"};
    string[] reptiles     = {"Crocodile", "Lizard", "Snake", "Tortoise"};
    string[] amphibians   = {"Frog", "Newt", "Salamander", "Toad"};
    string[] invertebrates = {"Earthworm", "Insect", "Jellyfish", "Spider"};

    ScriptChoiceList result = new ScriptChoiceList();

    addGroup("Mammals",      mammals);
    addGroup("Fish",        fish);
    addGroup("Birds",       birds);
    addGroup("Reptiles",    reptiles);
    addGroup("Amphibians",  amphibians);
    addGroup("Invertebrates", invertebrates);

    return result;

    void addGroup(string groupName, string[] animals)
    {
        ScriptChoice groupChoice = result.Add(groupName);
        foreach (string animal in animals)
            groupChoice.AddSubChoice(animal.ToQuotedScriptText());
    }
}

[GCDefaultTechnique]

```

```

public NodeUpdateResult Default
(
    NodeUpdateContext          updateContext,
    [GCDgnModelProvider] IPointNode  AnchorPoint,
    double                    CubicMeters,
    [EECCAttr(nameof(GetEECCForAnimal))] string Animal,
    [EECCAttr(nameof(GetEECCForDailySnacks))] IGCObject[] DailySnacks
)
{
    // Process the other inputs... Then:

    if (DailySnacks != null && DailySnacks.Length >= 2)
    {
        string name = DailySnacks[0].Unbox<string>();
        int count = DailySnacks[1].Unbox<int>();

        // Do whatever we do with those values.
    }

    return NodeUpdateResult.Success;
}
}
}

```

## 7.4.2. BioChamberDailySnacksEditor.xaml

```
<gcse:Editor x:Class="MyAddIn.BioChamberDailySnacksEditor"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:local="clr-namespace:MyAddIn"
  xmlns:gcse="urn:GC.ScriptEditor"
  mc:Ignorable="d"
  d:DesignHeight="450" d:DesignWidth="800">
  <StackPanel Orientation="Horizontal" >
    <TextBlock Text="Snack:" />
    <TextBox x:Name="_nameTextBox" Width="80" Margin="2,-4,2,0" FontSize="10" />
    <TextBlock Text="Count:" />
    <TextBox x:Name="_countTextBox" Width="30" Margin="2,-4,2,0" FontSize="10" />
  </StackPanel>
</gcse:Editor>
```

### 7.4.3. BioChamberDailySnacksEditor.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;

namespace MyAddIn
{
    /// <summary>
    /// Interaction logic for BioChamberDailySnacksEditor.xaml
    /// </summary>
    public partial class BioChamberDailySnacksEditor : Editor
    {
        public BioChamberDailySnacksEditor()
        {
            InitializeComponent();
        }

        // Minimal method overrides we need to implement so our custom editor
        // will be displayed, at least. We'll revisit these methods in the future.

        protected override bool IsExpressionScriptRepresentableByThisEditor(Script script)
        {
            // We'll just always return true. In some implementations, it makes sense
            // to pre-examine the script to ensure it's something we can deal with.
            // But, in this case, we'll handle any invalid script when we actually try
            // to interpret the script, in the method RefreshFromChangedExpressionScript,
            // below.
        }
    }
}
```

```

    return true;
}

protected override Script ExpressionScriptAsRepresentedByThisEditor()
{
    // Here we take the data components represented by this custom editor (i.e.,
    // a snack name and a snack count) and convert them to the script expression
    // that our node expects (i.e., a list: {'name', count}).

    string name = _nameTextBox.Text; // Get the name value.

    int.TryParse(_countTextBox.Text, out int count); // Get the count value
  // (or 0 if it's invalid).

    StringBuilder builder = new StringBuilder(); // Build the script, e.g.:

    builder.Append('{'); // {
    builder.Append(name.ToQuotedScriptText()); // {'pizza'
    builder.Append(", "); // {'pizza',
    builder.Append(count); // {'pizza', 2
    builder.Append('}'); // {'pizza', 2}

    string scriptText = builder.ToString();

    return scriptText.ToScript();
}

protected override UIElement OnActivatedWhileSelected()
{
    // This method determines which (if any) of our custom editor's child
    // controls should get the focus whenever this editor becomes active, and
    // returns that child control (or default, which is the same as null).

    return default;
}

protected override void RefreshFromChangedExpressionScript(Script newScript, Script oldScript)
{
    // Here we take the script representation of this property's value, i.e. {'name', count},
    // and use it to populate the controls of this custom editor.

    string name = ""; // Establish the default values we'll use

```

```

int count = 0;    // in case the given 'newScript' is invalid.

if (TryGetValueOfScript(out GCList value, newScript)) // GCList is a C# structure that
  // corresponds to a script list.
{
    // A GCList holds a collection of IGCOBjects, where each IGCOBject holds the C#
    // representation of a script object. To retrieve the underlying C# object from
    // each IGCOBject instance, we must "unbox" the instance.
    //
    // Actually, in this case, we're going to TRY to unbox it: The method 'TryUnbox'
    // returns false if the IGCOBject cannot be converted to a C# object of the
    // desired type.

    if (value.Count > 0 && value[0].TryUnbox(out string n))
        name = n;
    if (value.Count > 1 && value[1].TryUnbox(out int c))
        count = c;
}

_nameTextBox.Text = name;           // Finally, we assign the values
_countTextBox.Text = count.ToString(); // to their corresponding controls.
}
}
}

```

## 7.5. Custom expression editors, part 2

### 7.5.1. BioChamberNode.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.IO;
using System.Linq;
using Bentley.DgnPlatformNET;
using Bentley.DgnPlatformNET.Elements;
using Bentley.GenerativeComponents;
using Bentley.GenerativeComponents.AddInSupport;
using Bentley.GenerativeComponents.ElementBasedNodes;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.GCTypes;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.MicroStation;
using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;
using Bentley.GenerativeComponents.View;
using Bentley.GeometryNET;
using Bentley.MstnPlatformNET;

namespace MyAddIn
{
    using EECC = ExpressionEditorCustomConfiguration;
    using EECCAttr = GCExpressionEditorCustomConfigurationAttribute;

    [GCNamespace("User"), GCNodeTypePaletteCategory("My Add-In")]
    public class BioChamberNode: ElementBasedNode
    {
        static EECC GetEECCForAnimal()
        {
            return new EECC(getScriptChoices);

            IEnumerable<ScriptChoice> getScriptChoices(ExpressionEditor parentExpressionEditor)
            {
                BioChamberNode node = (BioChamberNode) parentExpressionEditor.MeaningOfThis;
            }
        }
    }
}
```

```

        ScriptChoiceList result = node.GetGroupedAnimals();

        return result;
    }
}

static EECC GetEECCForDailySnacks()
{
    return new EECC(typeof(BioChamberDailySnacksEditor));
}

ScriptChoiceList GetGroupedAnimals()
{
    // Simulate determining the animals at runtime, based on the current state of this node.
    // (For this example, we're simply hard-coding the result.)

    string[] mammals      = {"Bat", "Elephant", "Squirrel", "Whale"};
    string[] fish         = {"Fish", "Eel", "Minnow", "Salmon", "Shark"};
    string[] birds        = {"Chicken", "Falcon", "Owl", "Penguin"};
    string[] reptiles     = {"Crocodile", "Lizard", "Snake", "Tortoise"};
    string[] amphibians   = {"Frog", "Newt", "Salamander", "Toad"};
    string[] invertebrates = {"Earthworm", "Insect", "Jellyfish", "Spider"};

    ScriptChoiceList result = new ScriptChoiceList();

    addGroup("Mammals",      mammals);
    addGroup("Fish",        fish);
    addGroup("Birds",       birds);
    addGroup("Reptiles",    reptiles);
    addGroup("Amphibians",  amphibians);
    addGroup("Invertebrates", invertebrates);

    return result;

    void addGroup(string groupName, string[] animals)
    {
        ScriptChoice groupChoice = result.Add(groupName);
        foreach (string animal in animals)
            groupChoice.AddSubChoice(animal.ToQuotedScriptText());
    }
}

[GCDefaultTechnique]

```

```

public NodeUpdateResult Default
(
    NodeUpdateContext          updateContext,
    [GCDgnModelProvider] IPointNode  AnchorPoint,
    double                    CubicMeters,
    [EECCAttr(nameof(GetEECCForAnimal))] string Animal,
    [EECCAttr(nameof(GetEECCForDailySnacks))] IGCObject[] DailySnacks
)
{
    // Process the other inputs... Then:

    if (DailySnacks != null && DailySnacks.Length >= 2)
    {
        string name    = DailySnacks[0].Unbox<string>();
        int    count   = DailySnacks[1].Unbox<int>();
        double unitCost = DailySnacks.Length > 2 ? DailySnacks[2].Unbox<double>() : 0.0;

        // Do whatever we do with those values.
    }

    return NodeUpdateResult.Success;
}

internal IEnumerable<string> DailySnackChoices()
{
    string animal = GetPropertyValue("Animal", "");

    // Maybe in the future we'll introduce a switch statement here, e.g.:
    //
    //     switch(animal)
    //     {
    //         case "Bat":      // Return a collection of snacks that are appropriate for bats.
    //         case "Chicken": // Return a collection of snacks that are appropriate for chickens.
    //         ...             // Etc.
    //     }
    //
    // But, for now, we'll just hard-code some silly snacks that all the animals can enjoy.

    return new string[] {"Chips", "Pizza", "Popcorn", "Toast"};
}
}

```

## 7.5.2. BioChamberDailySnacksEditor.xaml

```
<gcse:Editor x:Class="MyAddIn.BioChamberDailySnacksEditor"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:local="clr-namespace:MyAddIn"
  xmlns:gcse="urn:GC.ScriptEditor"
  mc:Ignorable="d"
  d:DesignHeight="450" d:DesignWidth="800">
  <DockPanel>
    <Button x:Name="_launchItemEntryDialogButton" DockPanel.Dock="Left" Width="18" Height="18"
      Click="_launchItemEntryDialogButton_Click"
      BorderBrush="LightGray" Style="{StaticResource {x:Static ToolBar.ButtonStyleKey}}"
      Content="..." Padding="0,-4,0,4" ToolTip="Click to launch item-entry dialog"
      />
    <TextBlock x:Name="_presentationTextBlock" Margin="2,0,0,0"/>
  </DockPanel>
</gcse:Editor>
```

## BioChamberDailySnacksEditor.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.GCScript;
```

```

using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.ScriptEditor.Controls.ExpressionEditor;

namespace MyAddIn
{
    public partial class BioChamberDailySnacksEditor: Editor
    {
        string _name;    // For ease of implementation, we maintain local copies
        int _count;     // of the three components encapsulated within our node
        double _unitCost; // property, BioChamber.DailySnacks.

        public BioChamberDailySnacksEditor()
        {
            InitializeComponent();
        }

        protected override bool IsExpressionScriptRepresentableByThisEditor(Script script)
        {
            // We'll just always return true. In some implementations, it makes sense
            // to pre-examine the script to ensure it's something we can deal with.
            // But, in this case, we'll handle any invalid script when we actually try
            // to interpret the script, in the method RefreshFromChangedExpressionScript,
            // below.

            return true;
        }

        protected override Script ExpressionScriptAsRepresentedByThisEditor()
        {
            // Here we take the data components represented by this custom editor (i.e.,
            // snack name, snack count, and snack unit cost) and convert them to the script
            // expression that our node expects (i.e., a list: {'name', count, unitCost}).

            StringBuilder builder = new StringBuilder(); // Build the script, e.g.:

            builder.Append('{'); // {
            builder.Append(_name.ToQuotedScriptText()); // {'pizza'
            builder.Append(", "); // {'pizza',
            builder.Append(_count); // {'pizza', 2
            builder.Append(", "); // {'pizza', 2,
            builder.Append(_unitCost); // {'pizza', 2, 7.95
            builder.Append('}'); // {'pizza', 2, 7.95}
        }
    }
}

```

```

    string scriptText = builder.ToString();

    return scriptText.ToScript();
}

protected override UIElement OnActivatedWhileSelected()
{
    // This method determines which (if any) of our custom editor's child
    // controls should get the focus whenever this editor becomes active, and
    // returns that child control (or default, which is the same as null).

    return _launchItemEntryDialogButton; // We want our launch button to get
  // the focus each time.
}

protected override void RefreshFromChangedExpressionScript(Script newScript, Script oldScript)
{
    // Here we take the script representation of this property's value, i.e.,
    // {'name', count, unitCost}, and use it to populate the local fields of
    // of this custom editor.

    _name      = ""; // Establish the default values we'll use
    _count     = 0;  // in case the given 'newScript' is invalid.
    _unitCost  = 0.0;

    if (TryGetValueOfScript(out GCList value, newScript)) // GCList is a C# structure that
   // corresponds to a script list.
    {
        // A GCList holds a collection of IGCOBjects, where each IGCOBject holds the C#
        // representation of a script object. To retrieve the underlying C# object from
        // each IGCOBject instance, we must "unbox" the instance.
        //
        // Actually, in this case, we're going to TRY to unbox it: The method 'TryUnbox'
        // returns false if the IGCOBject cannot be converted to a C# object of the
        // desired type.

        if (value.Count > 0 && value[0].TryUnbox(out string s))
            _name = s;
        if (value.Count > 1 && value[1].TryUnbox(out int n))
            _count = n;
        if (value.Count > 2 && value[2].TryUnbox(out double d))
            _unitCost = d;
    }
}

```

```

    // Following is where we make an attractive display for the user.

    if (_name.Length == 0)
        _presentationTextBlock.Text = "";
    else
        _presentationTextBlock.Text = $"{_count} {_name}(s) @ {_unitCost} each";
}

private void _launchItemEntryDialogButton_Click(object sender, RoutedEventArgs e)
{
    // Now we're going to create, populate, launch, and process the results from, our
    // custom dialog, which we've "obtained from somewhere". This is just an example:
    // The way you work with your own custom dialog may be very different.

    // Create an instance of the custom dialog.

    ItemEntryDialog dialog = new ItemEntryDialog();

    // Before we show the custom dialog to the user, we'll pre-populate its field values.
    //
    // First we need to query the current BioChamber node to get the choices of daily
    // snacks. The following statement may look weird, but it's the standard way of
    // getting the current node instance from within a custom expression editor.

    BioChamberNode bioChamber = (BioChamberNode) ParentExpressionEditorMeaningOfThis;

    // Now we assign the custom dialog's field values from our values that are based
    // on the underlying node property (BioChamber.DailySnacks). This particular custom
    // dialog gives us nice C# properties that we can simply assign values to.

    dialog.ItemChoices = bioChamber.DailySnackChoices();

    dialog.ItemName     = _name;
    dialog.Count        = _count;
    dialog.UnitCost     = _unitCost;

    // Now we can show the dialog to the user (modally), and wait for them to fill or
    // edit the fields before they close the dialog.

    bool? b = dialog.ShowDialog();

    if (b.GetValueOrDefault()) // i.e., if the user clicked OK rather than Cancel.

```

```
{  
    // Retrieve the field values from the dialog...  
  
    _name      = dialog.ItemName;  
    _count     = dialog.Count;  
    _unitCost  = dialog.UnitCost;  
  
    // ...then notify GC that this expression has changed. In turn, GC will call  
    // the method ExpressionScriptAsRepresentedByThisEditor(), allowing us to  
    // formulate the actual script expression that will be passed to our node.  
  
    CommitChangesMadeToExpressionScript();  
}  
}
```

## 7.5.3. ItemEntryDialog.xaml

If you're curious, here's the code (provided as-is) of the custom dialog that we "obtained from somewhere".

```
<Window x:Class="MyAddIn.ItemEntryDialog" x:ClassModifier="internal"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:MyAddIn"
  xmlns:system="clr-namespace:System;assembly=mscorlib"
  mc:Ignorable="d"
  Title="Item Entry" Height="133" Width="410" ResizeMode="NoResize">
  <DockPanel Margin="10,8,10,10" LastChildFill="True">
    <StackPanel Dock="Bottom" Orientation="Horizontal" HorizontalAlignment="Right" Margin="0,5,0,0" Height="25" >
      <StackPanel.Resources>
        <Style TargetType="Button" >
          <Setter Property="Margin" Value="3,3,3,0" />
          <Setter Property="Width" Value="50" />
        </Style>
      </StackPanel.Resources>
      <Button x:Name="_okButton" Content="OK" IsDefault="True" Click="_okButton_Click" />
      <Button x:Name="_cancelButton" Content="Cancel" IsCancel="True" Click="_cancelButton_Click" />
    </StackPanel>
    <Grid Margin="0,0,0,0" >

      <Grid.Resources>
        <Style TargetType="TextBlock" >
          <Setter Property="HorizontalAlignment" Value="Center" />
          <Setter Property="VerticalAlignment" Value="Center" />
        </Style>
        <Style TargetType="TextBox" >
          <Setter Property="HorizontalContentAlignment" Value="Center" />
          <Setter Property="Margin" Value="5" />
          <Setter Property="VerticalContentAlignment" Value="Center" />
        </Style>
      </Grid.Resources>

      <Grid.RowDefinitions>
        <RowDefinition Height="1*" />
        <RowDefinition Height="3*" />
      </Grid.RowDefinitions>
    </Grid>
  </DockPanel>
</Window>
```

```

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
  <ColumnDefinition Width="3*" />
  <ColumnDefinition Width="2*" />
  <ColumnDefinition Width="2*" />
  <ColumnDefinition Width="2*" />
</Grid.ColumnDefinitions>

<TextBlock Text="Item Name" Grid.Row="0" Grid.Column="0" />
<TextBlock Text="Unit Cost" Grid.Row="0" Grid.Column="1" />
<TextBlock Text="Count" Grid.Row="0" Grid.Column="2" />
<TextBlock Text="Extension" Grid.Row="0" Grid.Column="3" />

<ComboBox x:Name="_itemNameComboBox" Grid.Row="1" Grid.Column="0" Margin="5" />

<TextBox x:Name="_unitCostTextBox" Grid.Row="1" Grid.Column="1" TextChanged="_unitCostTextBox_TextChanged" />
<TextBox x:Name="_countTextBox" Grid.Row="1" Grid.Column="2" TextChanged="_countTextBox_TextChanged" />
<TextBox x:Name="_extensionTextBox" Grid.Row="1" Grid.Column="3" IsReadOnly="True" Background="#FFF0F0" />
</Grid>
</DockPanel>
</Window>

```

## 7.5.4. ItemEntryDialog.xaml.cs

If you're curious, here's the code (provided as-is) of the custom dialog that we "obtained from somewhere".

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace MyAddIn
{
    partial class ItemEntryDialog : Window
    {
        public ItemEntryDialog()
        {
            InitializeComponent();
        }

        public IEnumerable<string> ItemChoices { get => _itemNameComboBox.Items.OfType<string>();
  set {
  _itemNameComboBox.Items.Clear();
  if (value != null)
  {
  foreach (string s in value)
  _itemNameComboBox.Items.Add(s);
  }
  }

        public string ItemName { get => _itemNameComboBox.Text;
                                set => _itemNameComboBox.Text = value; }
        public double UnitCost { get { double.TryParse(_unitCostTextBox.Text, out double d); return d; }
    }
}
```

```

        set => _unitCostTextBox.Text = value.ToString("N2"); }
public int    Count    { get { int.TryParse(_countTextBox.Text, out int n); return n; }
                      set => _countTextBox.Text = value.ToString(); }

void _unitCostTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    CalculateExtension();
}

void _countTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    CalculateExtension();
}

void CalculateExtension()
{
    if (_extensionTextBox != null) // Make sure the dialog isn't still initializing.
        _extensionTextBox.Text = (UnitCost * Count).ToString("N2");
}

void _okButton_Click(object sender, RoutedEventArgs e)
{
    DialogResult = true;
}

void _cancelButton_Click(object sender, RoutedEventArgs e)
{
    DialogResult = false;
}
}
}

```

## 7.6. GC Service Client (SimpleGCServiceClient)

### 7.6.1. Program.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using SimpleGCServiceClient.ServiceReference1;

namespace SimpleGCServiceClient
{
    internal class Program
    {
        static GCServiceClient gcService = new GCServiceClient();

        static void Main(string[] args)
        {
            gcService.Ping();
            // PerformCalculations();
            LoadGCModel();
            ExerciseGCModel();
        }

        static void PerformCalculations()
        {
            string[] scriptExpressions = { "2 + 3", "Sqrt(9)", "new DVector2d(3, 4).Magnitude" };

            AnnotatedLabeledNumbers results = gcService.GetNumericValues("", scriptExpressions);

            DisplayResults(results);
        }

        static void DisplayResults(AnnotatedLabeledNumbers results)
        {
            foreach (LabeledNumber result in results.LabeledNumbers)
                Console.WriteLine($"{result.Label,-30} = {result.Number}");
        }
    }
}
```

```

        Console.WriteLine();
        Console.Write("Finished. Press [Enter] to close. ");
        Console.ReadLine();
    }

    static void LoadGCModel()
    {
        // Replace the following file path with the full path to your own transaction file.
        const string transactionFilePath = @"D:\GC API Documentation\Test model for GC service.gct";

        gcService.LoadTransactionFile(transactionFilePath);
        Thread.Sleep(1000); // Wait one second while GC loads the transaction file.
        gcService.PlayTransactions(100);
        gcService.FitViews(turnOnAutoRefit:true);
    }

    static void ExerciseGCModel()
    {
        LabeledNumber[] propertyValues = { new LabeledNumber() {Label="point3.T", Number = 0.8} };

        gcService.ApplyNodePropertyNumericValues("", propertyValues);
        gcService.UpdateGCModel("");

        string[] scriptExpressions = { "line2.Length" };

        AnnotatedLabeledNumbers results = gcService.GetNumericValues("", scriptExpressions);
        DisplayResults(results);
    }
}

```

## 7.7. SampleAddIn (distributed with GC)

For reference purposes, the subpages under this page list the contents of each of the source-code files included in GC's sample add-in project. They are:

- [SimpleLineNode.cs](#)

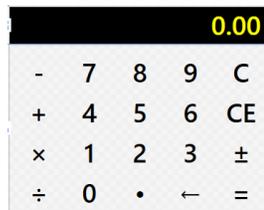
A class that defines an example element-based node named 'SimpleLine'.

- [CalculatorNode.cs](#)

A class that defines an example utility node named 'Calculator'.

- [CalculatorNodeViewContent.xaml](#), [CalculatorNodeViewContent.xaml.cs](#)

Collectively, these files define the UI for the Calculator node. In Visual Studio's designer, it looks like this:



- [ScriptFunctions.cs](#)

A static class that defines a new global script function named 'ConvertTemperature'.

Technically, there's no reason why this particular function couldn't be implemented in GC's scripting language rather than C#. More commonly, script functions are implemented in C# because (1) they make direct calls into GC's and/or MicroStation's API that can't be made from script code, and/or (2) performance speed is a concern.

- [Initializer.cs](#)

A static class that demonstrates how to define code that's run automatically when this add-in is loaded into GC.

This is important for script functions implemented in C# (such as 'ConvertTemperature') because this is the mechanism by which those functions get added to GC's script function library.

- **GC's Sample Add-in project**

To get the latest versions of these files, as well as the additional files that define the C# project, find a file named "GCSampleAddIn.zip" in your GC installation folder. In a standard installation of Open Buildings Designer, that file is in this folder:

<C:\Program Files\Bentley\OpenBuildings CONNECT Edition\OpenBuildingsDesigner\GenerativeComponents\SampleSolution\>

To "install" the sample add-in project, extract the contents of that zip file to a local folder; for example, a new folder named "GCSampleAddIn" under your standard Documents folder.

Within that extracted folder, read and follow the instructions in the file, "Please Read Me First.txt". Important: That file describes a couple of environment variables you'll to define on your computer.

## 7.7.1. SimpleLineNode.cs

```
// This file contains the definition of a new GC node type named SimpleLine. It is an extremely
// simplified version of GC's Line node type.
//
// GC supports two distinct architectures for node types, differentiated by which of the
// two classes, ElementBasedNode or UtilityNode, the node type inherits from.
//
// Element-based node types are designed for creating and/or managing elements;
// typically, graphical elements in the geometry views. This class, SimpleLineNode, is an
// example: Its only technique, ByPoints, generates a new graphical line element.
//
// Utility node types are designed for non-element-related operations with GC's
// graph, having custom appearances and/or behaviors. The class CalculatorNode, defined
// elsewhere in this project is an example.

using System;
using System.Collections.Generic;
using System.ComponentModel;
using Bentley.GenerativeComponents;
using Bentley.GenerativeComponents.AddInSupport;
using Bentley.GenerativeComponents.ElementBasedNodes;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.GCTypes;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.MicroStation;
using Bentley.GenerativeComponents.View;
using Bentley.DgnPlatformNET;
using Bentley.DgnPlatformNET.Elements;
using Bentley.GeometryNET;
using Bentley.MstnPlatformNET;

namespace SampleAddIn
{
    [GCNamespace("User")] // The GCNamespace attribute lets us
                        // specify where this SimpleLine node
                        // type will appear within GCScript's
                        // namespace tree (that is, the namespaces
                        // that are perceived by the GC user).
                        // This namespace has no relation to our
                        // C# namespace, which is (in this case)
```

```

// SampleAddIn.

[GCNodeTypePaletteCategory("Sample Add-In")] // The GCNodeTypePaletteCategory attribute
// lets us specify where this SimpleLine
// node type will appear within GC's Node
// Types panel. So, it will appear within
// a group named "Sample Add-In".

[GCNodeTypeIcon("Resources/SimpleLineNode.png")] // The GCNodeTypeIcon attribute lets us
// specify the graphical image (icon)
// that will appear on the SimpleLine node
// type's button within GC's Node Types
// panel.

[GCSummary("A line that connects two points in space.")] // The GCSummary attribute lets us provide
// a brief description of this node's
// intended purposed. The text will be
// displayed when the user hovers over our
// node type in GC's Node Types panel.

public class SimpleLineNode: ElementBasedNode // Every element-based node-type class derives
// from the class ElementBasedNode, or from
// another class type that, itself, derives
// from ElementBasedNode, such as GeometricNode.
{
    // You can see that this class is named "SimpleLineNode" rather than just "SimpleLine",
    // which is the name that will be presented to the user. By convention, all classes
    // that define GC node types are named with the suffix "Node".

    // An explicit constructor is neither needed nor wanted. The default constructor
    // is sufficient.

    // protected override void OnInitialized()
    // {
    //     // Here is where we can perform any custom initialization of this class instance.
    //     // (It is NOT necessary to call the base class implementation.)
    // }

    // One of the fundamental differences between the element-based node architecture
    // and the utility node architecture is that, in the former, reflection is used
    // to extract the technique names, the documentation, and the names and types of the inputs
    // and outputs, directly from the compiled C# class. GC uses several custom attributes
    // to provide more information on that reflection process.

```

```

[GCDefaultTechnique] // Every technique method -- that is, every method that implements
                      // a technique of an element-based node type -- must be marked
                      // with one of the attributes [GCTechnique] or [GCDefaultTechnique].
                      // If a method doesn't have either of those attributes, then it's not
                      // exposed to the GC user; it's just an ordinary C# class method.
                      //
                      // (Only ONE method in the class may have the [GCDefaultTechnique]
                      // attribute.)

// The following attributes -- GCSummary and GCParameter -- do not affect the functionality
// of this node within GC. They do, however, enhance the user's experience by providing
// documentation, which appears in various tooltip/flyover labels in GC's UI.

[GCSummary("Creates a line between a given start and end point, or a list of lines between lists of start points and/or
endpoints.")]
[GCParameter("StartPoint", "Start point of the line.")]
[GCParameter("EndPoint", "End point of the line.")]

public NodeUpdateResult ByPoints
(
    NodeUpdateContext          updateContext,
    [GCDgnModelProvider, GCReplicable] IPointNode StartPoint,
    [GCReplicable] IPointNode   EndPoint,
    [GCOut] ref double          Length
)
{
    // The first parameter of every technique method must be of the type, NodeUpdateContext.
    //
    // The remaining parameters will become the input and output properties that the GC user
    // sees and manipulates.
    //
    // Specific to these particular parameters:
    //
    // -- The [GCDgnModelProvider] attribute denotes that, if this is a new instance of
    //     SimpleLine, its elements will be added to the DGN model as those of the node
    //     the user inputs to this parameter (StartPoint).
    //
    // -- The [GCReplicable] attribute denotes that, at the GC level, the value assigned
    //     to this property may be either a single item or a list. If it's a list, then this
    //     node will become replicated, and (therefore) this method, ByPoints, will be called
    //     multiple times, once for each item in the given list.
    //

```

```

//      (Note that, except in very rare circumstances, you'll never need to worry about
//      whether the node is in singleton or replicated mode. You simply write your code
//      to handle the singleton case, and GC will take care of the rest.)
//
//  -- The interface type, IPointNode, is implemented by various node types, such as
//      Point and CoordinateSystem. Essentially, IPointNode represents any node type that
//      can provide X, Y, and Z coordinates. If, for some reason, you wanted to restrict
//      this technique to take only actual Point nodes, you would simply change the
//      parameter type accordingly (from IPointNode to PointNode).
//
//  -- Together, the GCOut attribute and the "ref" keyword denote that this parameter
//      represents an output property, rather than an input property, of this technique.
//      (This is slightly unusual; most techniques of most node types have only inputs,
//      not outputs.)
//
// We start by checking the validity of the inputs. If necessary, we return a result
// that indicates which particular inputs are invalid. (Subsequently, the node will be
// displayed with an error badge and a tooltip showing the names of the invalid inputs.)

if (StartPoint == EndPoint)
    return new NodeUpdateResult.TechniqueInvalidArguments(nameof(StartPoint), nameof(EndPoint));

// Okay, the inputs are valid. What follows is the actual functionality of this technique
// method.

// Creating a LineElement (Bentley.DgnPlatformNET.Elements.LineElement) will require
// that we first get the DPoint3d coordinates of the nodes that were input by the user.

DPoint3d startPt = StartPoint.GetDPoint3d();
DPoint3d endPt   = EndPoint.GetDPoint3d();

// GC measures things in Master Units, while the element types in
// Bentley.DgnPlatformNET.Elements work with UORs. So, we need to perform a conversion.

DPoint3d uorStartPt = NativeDgnTools.FromMUToUOR(startPt);
DPoint3d uorEndPt   = NativeDgnTools.FromMUToUOR(endPt);

// Now we can create a LineElement...

LineElement lineElement = new LineElement(GCDgnModel().DgnModel(), TemplateElement(),
   new DSegment3d(uorStartPt, uorEndPt));

// ...And associate that LineElement with this instance of SimpleLineNode.

```

```
SetElement(lineElement);

// Before we can return, we must populate our output parameter. Since, again, GC works
// in Master Units, our calculation must be based on the Master-Unit representations.

Length = startPt.Distance(endPt);

// Finally, if all's well, we return the indication of success. Other options include,
// for example, returning a value like this:
// return new NodeUpdateResult.TechniqueFailureMessage(Ls.Literal("Reason that the technique failed.")).

return NodeUpdateResult.Success;
    }
} // class
} // namespace
```

## 7.7.2. CalculatorNode.cs

```
// This file, in conjunction with the files CalculatorNodeViewContent.xml and
// CalculatorNodeViewContent.xaml.cs defines a new GC node type named Calculator.
// It emulates a simple, four-function handheld calculator; conceptually, it's an
// extremely simplified version of GC's own Value node type.
//
// GC supports two distinct architectures for node types, differentiated by which of the
// two classes, ElementBasedNode or UtilityNode, the node type inherits from.
//
// Element-based node types are designed for creating and/or managing elements;
// typically, graphical elements in the geometry views. The class SimpleLineNode, defined
// elsewhere in this project is an example.
//
// Utility node types are designed for non-element-related operations with GC's
// graph, having custom appearances and/or behaviors. This class, CalculatorNode, is an
// example: It provides a calculator simulation whose results can be passed along to other
// nodes in the GC model.

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Media;
using Bentley.GenerativeComponents;
using Bentley.GenerativeComponents.AddInSupport;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.GCTypes;
using Bentley.GenerativeComponents.GCScript.NameScopes;
using Bentley.GenerativeComponents.GCScript.ReflectedNativeTypeSupport;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.UtilityNodes;
using Bentley.GenerativeComponents.View;

namespace SampleAddIn
{
    [GCNamespace("User")] // The GCNamespace attribute lets us specify
                        // where this Calculator node type will appear
                        // within GCScript's namespace tree (that
```

```

// is the namespaces that are perceived by
// the GC user). This namespace has no relation
// to our C# namespace, which is (in this case)
// SampleAddIn.

[GCNodeTypePaletteCategory("Sample Add-In")] // The GCNodeTypePaletteCategory attribute
// lets us specify where this Calculator node
// type will appear within GC's Node Types
// panel. So, it will appear within a group
// named "Sample Add-In".

[GCNodeTypeIcon("Resources/CalculatorNode.png")] // The GCNodeTypeIcon attribute lets us specify
// the graphical image (icon) that will appear
// on the Calculator node type's button
// within GC's Node Types panel.

[GCSummary("An emulation of a simple calculator.")] // The GCSummary attribute lets us provide
// a brief description of this node's
// intended purpose. The text will be
// displayed when the user hovers over our
// node type in GC's Node Types panel.

public class CalculatorNode: UtilityNode // Every utility node-type class derives from the class
// UtilityNode, or from another class that, itself,
// derives from UtilityNode.
{
    // You can see that this class is named "CalculatorNode" rather than just "Calculator",
    // which is the name that will be presented to the user. By convention, all classes
    // that define GC node types are named with the suffix "Node".

    // The following static method, which must be named AddAdditionalMembersToGCType and have
    // this signature, is where we define this node type's techniques, as well as the inputs
    // and outputs to each technique.

    static void AddAdditionalMembersToGCType(IGCEnvironment environment, GCType gcType,
        NativeNamespaceTranslator namespaceTranslator)
    {
        // This Calculator node type has only one technique, which we've named "Default".
        // It takes two inputs, DecimalPlaces and NumberColor, and produces one output,
        // Result. (This is slightly unusual; most techniques of most node types have only
        // inputs, not outputs.)
        //
        // To define each of a technique's inputs and outputs, we call the method,

```

```

// technique.AddParameter. Each of those calls takes the following [C#] arguments:
//
// 1. The current environment, which we simply pass along.
//
// 2. The name of the input or output property, as it will be seen by the GC user.
//
// 3. The data type of the property (in C# terms).
//
// 4. The GCScript expression that will become the initial value of that property
// whenever the GC user creates a new instance of this node type. This is relevant
// only to inputs, although this argument must still be present when we define
// outputs (in which case, we simply pass "default(string)").
//
// 5. The description of this property, which will be seen by the GC user in various
// tooltips. The type of this argument is "Ls", which means "localized string".
// Since we won't be translating this project into multiple (human) languages,
// we can simply use the form Ls.Literal("string").
//
// 6. If the property is an output property, it takes a sixth argument,
// NodePortRole.TechniqueOutputOnly.

UtilityNodeTechnique technique1 = gcType.AddDefaultNodeTechnique("Default", DefaultTechnique);

technique1.AddParameter(environment, nameof(DecimalPlaces), typeof(int), "2",
    Ls.Literal("The number of digits to display after the decimal point.));

technique1.AddParameter(environment, nameof(NumberColor), typeof(Color), "Colors.Yellow",
    Ls.Literal("The color of the displayed number.));

technique1.AddParameter(environment, nameof(Result), typeof(double), default(string),
    Ls.Literal("The result of the calculation."),
    NodePortRole.TechniqueOutputOnly);
}

// The following static method implements this node type's one-and-only technique, "Default".
// Every technique method implementation must have this signature. (The method's name can
// be anything.)

static NodeUpdateResult DefaultTechnique(UtilityNode node, NodeUpdateContext updateContext)
{
    CalculatorNode calculator = (CalculatorNode) node; // It's safe to assume that the
// given node is of this class type,
// Calculator.

```

```

// Following is an example of an input validity test: Before proceeding, we check
// whether the input value DecimalPlaces, which is an integer, is in the range 0
// through 9 (inclusive). (9 is an arbitrary upper limit.)
//
// If it's outside of that range, we return a result that indicates the name of the
// offending property. The GC user will see an error badge on this Calculator node
// instance, along with a message indicating which input is problematic.

if (!Range<int>.IsInRange(calculator.DecimalPlaces, 0, 9))
    return new NodeUpdateResult.TechniqueInvalidArguments(nameof(DecimalPlaces));

// Following is where we define the main functionality of this node technique. It may
// comprise just a single statement (as it does here), or, more commonly, a block of
// code that processes the node instance.

node.NotifyAllPropertiesChanged(); // In the case of this Calculator node type, all
// of the calculator logic is implemented in the
// WPF layer, so this C# method simply forwards
// the new values of DecimalPlaces and NumberColor
// to that layer, using WPF data bindings.

// Finally, if all's well, we return the indication of success. Other options include,
// for example, returning a value like this:
// return new NodeUpdateResult.TechniqueFailureMessage(Ls.Literal("Reason that the technique failed.")).

return NodeUpdateResult.Success;
}

// Beginning of instance members.

// The following two members -- State and GetInitialState -- are simply "must be defined"
// methods that every utility node class must have. You can simply copy and
// paste these two definitions into your own utility classes.

internal new NodeState State => (NodeState) base.State;

protected override UtilityNode.NodeState GetInitialState(NodeTechniqueDetermination initialActiveTechniqueDetermination)
    => new NodeState(this, initialActiveTechniqueDetermination);

// The following property associates a WPF framework element with this node type. (Typically,
// as is the case here, the framework element is a kind of WPF UserControl.) Subsequently,
// whenever this node type is instantiated, an instance of the framework element will be

```

```

// instantiated, as well, and appear on the face of the node within GC's graph.

public override Type TypeOfCustomViewContent => typeof(CalculatorNodeViewContent);

// The following three "convenience" properties give us easy access, at the C# level, to
// the values of this node's technique inputs and outputs. They also allow us to use
// C#'s 'nameof' operator in various places within this class.
//
// (Furthermore, as in the case of this Calculator node type, these properties can serve
// as binding sources for this node type's WPF layer.)

public int DecimalPlaces
{
    get => State.DecimalPlacesProperty.GetNativeValue<int>();
    set => State.DecimalPlacesProperty.SetNativeValueAndInputExpression(value);
}

public Color NumberColor
{
    get => State.NumberColorProperty.GetNativeValue<Color>();
    set => State.NumberColorProperty.SetNativeValueAndInputExpression(value);
}

public double Result
{
    get => State.ResultProperty.GetNativeValue<double>();
    set => State.ResultProperty.SetNativeValueAndInputExpression(value);
}

// The following method (specific to this CalculatorNode class) will be called from
// the WPF layer, whenever the calculator has a new result because the user clicked
// the "=" (equals) key.

internal void SetResultAndUpdate(double result)
{
    Result = result;

    this.UpdateNodeTree(false); // This is how a node can force an update of its own
                                // successor/downstream nodes within the GC model.
                                //
                                // (Normally, a node doesn't do this; it lets GC's
                                // infrastructure handle all node updating. However, this
                                // Calculator node is a special case; we want the updating

```

```

        // to occur as soon as the user has clicked the "="
        // (equals) key on the WPF side.
    }

    // Following is the embedded NodeState class that every utility class must have.
    //
    // This embedded class can be copied and pasted from this class to your own node type class,
    // then you can replace the list of properties with the inputs and outputs that are specific
    // to your own class.

    public new class NodeState: UtilityNode.NodeState
    {
        // There must be one NodeProperty field for each unique input and output of your node
        // type, regardless of which technique(s) each input and output belongs to. The order
        // of these fields is irrelevant, but we suggest listing them alphabetically.

        internal readonly UtilityNodeProperty DecimalPlacesProperty;
        internal readonly UtilityNodeProperty NumberColorProperty;
        internal readonly UtilityNodeProperty ResultProperty;

        internal protected NodeState(CalculatorNode parentNode, NodeTechniqueDetermination
initialActiveTechniqueDetermination):
            base(parentNode, initialActiveTechniqueDetermination)
        {
            // IMPORTANT: This constructor calls 'AddProperty' to get each property field,
            // whereas the following constructor calls 'GetProperty'.

            DecimalPlacesProperty = AddProperty(nameof(DecimalPlaces));
            NumberColorProperty   = AddProperty(nameof(NumberColor));
            ResultProperty        = AddProperty(nameof(Result));
        }

        protected NodeState(NodeState source): base(source)
        {
            // IMPORTANT: This constructor calls 'GetProperty' to get each property field,
            // whereas the preceding constructor calls 'AddProperty'.

            DecimalPlacesProperty = GetProperty(nameof(DecimalPlaces));
            NumberColorProperty   = GetProperty(nameof(NumberColor));
            ResultProperty        = GetProperty(nameof(Result));
        }

        // The following two members -- UtilityNode and GetInitialState -- are simply

```



### 7.7.3. CalculatorNodeViewContent.xaml

```
<UserControl x:Class="SampleAddIn.CalculatorNodeViewContent"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  mc:Ignorable="d"
  d:DesignHeight="110" d:DesignWidth="140" FontSize="14" FontWeight="SemiBold">
  <Grid >
    <Grid.RowDefinitions>
      <RowDefinition Height="20"/>
      <RowDefinition Height="*" MinHeight="90"/>
    </Grid.RowDefinitions>

    <TextBlock Name="_displayTextBlock" Grid.Row="0" Background="Black" Foreground="Yellow"
      Padding="5,0,5,0" Text="0.00" TextAlignment="Right" />

    <UniformGrid Grid.Row="1" Rows="4" Columns="5" Margin="2" >
      <UniformGrid.Resources>
        <Style TargetType="Button" BasedOn="{StaticResource {x:Static ToolBar.ButtonStyleKey}}" >
          <Setter Property="HorizontalAlignment" Value="Stretch" />
        </Style>
      </UniformGrid.Resources>
      <Button Name="_subtractButton" Content="-" ToolTip="Subtract" Click="_subtractButton_Click" />
      <Button Name="_digit7Button" Content="7" Click="_digit7Button_Click" />
      <Button Name="_digit8Button" Content="8" Click="_digit8Button_Click" />
      <Button Name="_digit9Button" Content="9" Click="_digit9Button_Click" />
      <Button Name="_clearAllButton" Content="C" ToolTip="Clear All" Click="_clearAllButton_Click" />
      <Button Name="_addButton" Content="+" ToolTip="Add" Click="_addButton_Click" />
      <Button Name="_digit4Button" Content="4" Click="_digit4Button_Click" />
      <Button Name="_digit5Button" Content="5" Click="_digit5Button_Click" />
      <Button Name="_digit6Button" Content="6" Click="_digit6Button_Click" />
      <Button Name="_clearEntryButton" Content="CE" ToolTip="Clear Entry" Click="_clearEntryButton_Click" />
      <Button Name="_multiplyButton" Content="x" ToolTip="Multiply" Click="_multiplyButton_Click" />
      <Button Name="_digit1Button" Content="1" Click="_digit1Button_Click" />
      <Button Name="_digit2Button" Content="2" Click="_digit2Button_Click" />
      <Button Name="_digit3Button" Content="3" Click="_digit3Button_Click" />
      <Button Name="_changeSignButton" Content="±" ToolTip="Change Sign" Click="_changeSignButton_Click" />
      <Button Name="_divideButton" Content="÷" ToolTip="Divide" Click="_divideButton_Click" />
      <Button Name="_digit0Button" Content="0" Click="_digit0Button_Click" />
    </UniformGrid>
  </Grid>
</UserControl>
```

```
        <Button Name="_decimalPointButton" Content="•" ToolTip="Decimal Point" Click="_decimalPointButton_Click" />
        <Button Name="_backspaceButton" Content="←" ToolTip="Backspace" Click="_backspaceButton_Click" />
        <Button Name="_equalsButton" Content="=" ToolTip="Equals" Click="_equalsButton_Click" />
    </UniformGrid>
</Grid>
</UserControl>
```

## 7.7.4. CalculatorNodeViewContent.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.GeneralPurpose.Wpf;

namespace SampleAddIn
{
    // The following class, CalculatorNodeViewContent, is a WPF user control that defines the
    // appearance and behavior of the Calculator node type within the GC graph.
    //
    // It is (far) beyond the scope of this GC sample project to teach anything about WPF.
    // For that, there are lots of books and online materials.
    //
    // However, following is some information specific to the intercommunication between
    // a GC utility node instance and its corresponding WPF control, instance:
    //
    // 1. It is common that the values of one or more technique inputs need to be passed from
    //    the node to the WPF control.
    //
    //    That's the case with this Calculator node type: Suppose the GC user changes the value
    //    of the input, DecimalPlaces, which is a property on the node-type class (Calculator).
    //    We want that changed value to be passed to the WPF control so that the latter can
    //    adjust the number of decimal places it is showing in its "calculator display".
    //
    //    To accomplish that, we...
    //
    //    a. Define a dependency property in the WPF control that has the same name
```

```

//          (DecimalPlaces) and type (int) as the node property.
//
//      b. As part of that dependency property's definition, establish an "on changed"
//          method within the WPF control.
//
//      c. We programmatically establish a WPF binding from the node property to the WPF
//          dependency property.
//
// Then, within the node class, we can call OnPropertyChanged("DecimalPlaces") or
// OnAllPropertiesChanged() to refresh the binding, resulting in our "on changed" method
// being called within the WPF control. That "on changed" method can then, in turn,
// perform whatever UI adjustments need to be made.
//
// 2. While less common, it is often necessary to pass information from the WPF control to the
// node, and/or for the former to invoke some functionality on the latter.
//
// That's the case with this Calculator node type: When the GC user calculates a result by
// clicking the "=" (equals) key of the virtual calculator, we want that result to be passed
// to the node so that the node can (in turn) apply that value to any of its successor/
// downstream nodes.
//
// The conventional WPF way to do this would be to use a Command, but we can cut corners
// because we know that the WPF control's DataContext is our node instance. So, we can simply
// cast the DataContext to the appropriate type (Calculator) and then access that node
// instance's properties and methods directly.

public partial class CalculatorNodeViewContent: UserControl
{
    // ===== Types, constants, and static members =====

    enum Operator
    {
        None, Add, Divide, Subtract, Multiply
    };

    const int MaxDisplayLength = 12;          // Arbitrary upper limit on the number of characters
  // that can be entered into the display, excluding
  // the prefix minus-sign if there is one.

    const double ErrorValue      = 888888;   // What we display to indicate an error condition.

    static public read-only DependencyProperty DecimalPlacesProperty;
    static public read-only DependencyProperty NumberColorProperty;

```

```

    static CalculatorNodeViewContent()
    {
        {
            FrameworkPropertyMetadata md = new FrameworkPropertyMetadata(2, (d, e) => ((CalculatorNodeViewContent)
d).OnDecimalPlacesChanged(e));
            DecimalPlacesProperty = DependencyProperty.Register(nameof(DecimalPlaces), typeof(int),
typeof(CalculatorNodeViewContent), md);
        }
        {
            FrameworkPropertyMetadata md = new FrameworkPropertyMetadata(Colors.Yellow, (d, e) =>
((CalculatorNodeViewContent) d).OnNumberColorChanged(e));
            NumberColorProperty = DependencyProperty.Register(nameof(NumberColor), typeof(Color),
typeof(CalculatorNodeViewContent), md);
        }
    }

    // ===== Beginning of instance members =====

    Operator _pendingOperator = Operator.None; // None indicates there is no pending operation.

    double _pendingOperationFirstOperand;

    bool _nextDigitIsNewEntry = true;

    public CalculatorNodeViewContent()
    {
        InitializeComponent();
        ClearAll();
        this.SetBinding(DecimalPlacesProperty, nameof(CalculatorNode.DecimalPlaces));
        this.SetBinding(NumberColorProperty, nameof(CalculatorNode.NumberColor));
    }

    public int DecimalPlaces => (int) GetValue(DecimalPlacesProperty);

    public Color NumberColor => (Color) GetValue(NumberColorProperty);

    void OnDecimalPlacesChanged(DependencyPropertyChangedEventArgs e)
    {
        if (_nextDigitIsNewEntry || DisplayText == "")
        {
            double d = ValueFromDisplayText();
            DisplayText = DisplayTextFromValue(ref d);
        }
    }

```

```

    }
}

void OnNumberColorChanged(DependencyPropertyChangedEventArgs e)
{
    Color color = (Color) e.NewValue;
    Brush brush = WpfTools.GetSolidColorBrush(color);
    _displayTextBlock.Foreground = brush;
}

string DisplayText
{
    get
    {
        if (_nextDigitIsNewEntry)
        {
            _displayTextBlock.Text = DisplayTextThatMeansEmpty();
            _nextDigitIsNewEntry = false;
        }
        string result = _displayTextBlock.Text;
        if (result == DisplayTextThatMeansEmpty())
            result = "";
        return result;
    }
    set
    {
        if (string.IsNullOrEmpty(value))
            value = DisplayTextThatMeansEmpty();
        _displayTextBlock.Text = value;
    }
}

double ValueFromDisplayText()
{
    double result = 0.0;
    string text = DisplayText;
    if (text.Length > 0)
        result = double.Parse(text);
    return result;
}

string DisplayTextFromValue(ref double value)
{

```

```

// The result of this method could be simply "value.ToString()" if not for these
// shenanigans regarding the number of decimal places.

int decimalPlaces = DecimalPlaces;
value = Math.Round(value, decimalPlaces);
StringBuilder builder = new StringBuilder();
builder.Append(value.ToString());
int index = builder.IndexOf('.');
if (index < 0)
{
    if (decimalPlaces > 0)
    {
        builder.Append('.');
        builder.Append('0', decimalPlaces);
    }
}
else if (decimalPlaces == 0)
    builder.Length = index;
else
{
    int currentDecimalPlaces = builder.Length - index - 1;
    if (currentDecimalPlaces > decimalPlaces)
        builder.Length = index + decimalPlaces;
    else if (decimalPlaces > currentDecimalPlaces)
        builder.Append('0', decimalPlaces - currentDecimalPlaces);
}
return builder.ToString();
}

string DisplayTextThatMeansEmpty()
{
    double zero = 0.0;
    string result = DisplayTextFromValue(ref zero);
    return result;
}

void ProcessDigit(int digitValue)
{
    string text = DisplayText;
    if (text.Length < MaxDisplayLength)
        DisplayText += digitValue.ToString();
}

```

```

void InitiatePendingOperation(Operator op)
{
    if (!FinishPendingOperationIfAny(false)) // Ooo! Chain calculations!
        return; // An error occurred (which the previous call simply swallowed).

    _pendingOperator = op;
    _pendingOperationFirstOperand = ValueFromDisplayText();
    _nextDigitIsNewEntry = true;
}

bool FinishPendingOperationIfAny(bool pushResultToNode)
{
    // Extremely crude error handling: If any error occurs during any of these
    // arithmetic operations, this method performs a simple "Clear All" operation
    // and return false. Otherwise, we return true, meaning all's well (or there's
    // nothing to be done).
    //
    // In any case, when this method returns, the current pending operation (if
    // there was one) is cleared.

    bool success = true;
    if (_pendingOperator != Operator.None)
    {
        double firstOperand = _pendingOperationFirstOperand;
        double secondOperand = ValueFromDisplayText();
        double result = 0.0;
        switch (_pendingOperator)
        {
            case Operator.Add:      result = firstOperand + secondOperand; break;
            case Operator.Divide:   result = firstOperand / secondOperand; break;
            case Operator.Multiply: result = firstOperand * secondOperand; break;
            case Operator.Subtract: result = firstOperand - secondOperand; break;
        }
        if (double.IsNaN(result) || double.IsInfinity(result))
        {
            result = ErrorValue;
            success = false;
        }
        ClearPendingOperation();
        SetResultDisplay(ref result);
        if (success)
        {
            CalculatorNode node = (CalculatorNode) DataContext;

```

```

        node.SetResultAndUpdate(result);
    }
}
return success;
}

void SetResultDisplay(ref double result)
{
    DisplayText = DisplayTextFromValue(ref result);
    _nextDigitIsNewEntry = true;
}

void ClearPendingOperation()
{
    _pendingOperator = Operator.None;
    _pendingOperationFirstOperand = 0.0; // Just to be tidy.
}

void ClearDisplay()
{
    DisplayText = "";
    _nextDigitIsNewEntry = true;
}

void ClearAll()
{
    ClearPendingOperation();
    ClearDisplay();
}

void _addButton_Click(object sender, RoutedEventArgs e)
{
    InitiatePendingOperation(Operator.Add);
}

void _backspaceButton_Click(object sender, RoutedEventArgs e)
{
    string text = DisplayText;
    if (text.Length > 0)
        DisplayText = text.Substring(0, text.Length - 1);
}

void _changeSignButton_Click(object sender, RoutedEventArgs e)

```

```

{
    string text = DisplayText;
    if (text.Length > 0)
    {
        // Toggle the appearance of a '-' (minus character) at the beginning of the display.
        if (text[0] == '-')
            DisplayText = text.Substring(1);
        else
            DisplayText = '-' + text;
    }
}

void _clearAllButton_Click(object sender, RoutedEventArgs e)
{
    ClearAll();
}

void _clearEntryButton_Click(object sender, RoutedEventArgs e)
{
    ClearDisplay();
}

void _decimalPointButton_Click(object sender, RoutedEventArgs e)
{
    // Append a decimal point only if there isn't already one.
    string text = DisplayText;
    if (!text.Contains("."))
        DisplayText += '.';
}

void _digit0Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(0);
void _digit1Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(1);
void _digit2Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(2);
void _digit3Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(3);
void _digit4Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(4);
void _digit5Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(5);
void _digit6Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(6);
void _digit7Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(7);
void _digit8Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(8);
void _digit9Button_Click(object sender, RoutedEventArgs e) => ProcessDigit(9);

void _divideButton_Click(object sender, RoutedEventArgs e)
{

```

```
        InitiatePendingOperation(Operator.Divide);
    }

    void _equalsButton_Click(object sender, RoutedEventArgs e)
    {
        FinishPendingOperationIfAny(true);
    }

    void _multiplyButton_Click(object sender, RoutedEventArgs e)
    {
        InitiatePendingOperation(Operator.Multiply);
    }

    void _subtractButton_Click(object sender, RoutedEventArgs e)
    {
        InitiatePendingOperation(Operator.Subtract);
    }
}
}
```

## 7.7.5. ScriptFunctions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Bentley.GenerativeComponents.AddInSupport;
using Bentley.GenerativeComponents.GeneralPurpose;
using Bentley.GenerativeComponents.GCScript;
using Bentley.GenerativeComponents.GCScript.FundamentalValues;
using Bentley.GenerativeComponents.GCScript.GCTypes;
using Bentley.GenerativeComponents.GCScript.NameScopes;

namespace SampleAddIn
{
    static internal class ScriptFunctions // Note this is a static class.
    {
        static internal void Load()
        {
            // This method is called from within the constructor of the class, Initializer (elsewhere
            // within this project). So, this method will be called automatically whenever the user
            // loads this assembly, SampleAddIn, into GC.

            IGCEnvironment environment = UniversalGCEnvironment.TheOnlyInstance;
            NameCatalog nameCatalog = environment.TopLevelNameCatalog();

            // To add a new function to the GCScript processor, we call the method,
            // nameCatalog.AddNamespaceLevelFunction.
            //
            // 1. The first argument is the name of the new function, as the GC user will see it.
            //    (It's our responsibility to ensure that the function name doesn't conflict with
            //    the name of another top-level object within the GC world, such as a pre-existing
            //    node type.)
            //
            // 2. The second argument is the type (including signature) of the new function,
            //    expressed in GCScript form.
            //
            // 3. The third argument is the name of the C# method that implements the new function.
            //    That method can have any name.

            nameCatalog.AddGlobalFunction("ConvertTemperature",
                "double function(double temperature, string mode)",
```

```

    ConvertTemperatureFunction);
}

// Each method that implements a script function may be prefaced by a 'GCSummary' attribute,
// which provides user documentation for that function. The documentation text will appear
// in GC's Functions panel. (The presence or absence of a GCSummary attribute has no effect
// on how the function may be used in GC.)

[GCSummary("Converts a temperature value between degrees Celsius (C) and degrees Fahrenheit (F). The 'mode' parameter
indicates the direction of the conversion: 'c' means F -> C and 'f' means C -> F.")]

static void ConvertTemperatureFunction(CallFrame callFrame)
{
    // This is the implementation method for our new script function, ConvertTemperature.
    //
    // Every implementation method must have this signature:
    // static void <method name> (CallFrame callFrame)

    // Start by getting the "native" C# values of the given arguments.

    double temperature = callFrame.UnboxArgument<double>(0); // Get the first argument
   // (i.e., the argument at index 0)
   // as a C# double value.
    string mode        = callFrame.UnboxArgument<string>(1); // Get the second argument
   // (i.e., the argument at index 1)
   // as a C# string value.

    // Following is the main body of the ConvertTemperature function.

    double result;

    if (mode == "C" || mode == "c")
        result = (temperature - 32) / 1.8; // Standard formula to convert Fahrenheit
   // degrees to Celsius.
    else if (mode == "F" || mode == "f")
        result = temperature * 1.8 + 32; // Standard formula to convert Celsius degrees
   // to Fahrenheit.
    else
    {
        // To cause a runtime exception to occur (with an appropriate message presented
        // to the user), throw a GCUserException.
        //
        // The argument to GCUserException is the error message. It's of the type Ls,

```

```
    // which means "localized string". Since we're not concerned with international-
    // ization in this project, it's sufficient to merely enclose the quoted string
    // within a call to the static method, Ls.Literal.

    throw new GCUserException(Ls.Literal($"Invalid mode \"{mode}\" given to ConvertTemperature."));
}

// If this GCScript function returns a value -- which it does, in this case -- we call
// the method SetFunctionFunction on the given callFrame, as shown below.
//
// If our function does NOT return a value -- that is, if our function's return type,
// as presented to the GC user, is "void" -- then we don't need to do anything special
// at the end of our function implementation method.

callFrame.SetFunctionResult(result);
}
}
```

## 7.7.6. Initializer.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Bentley.GenerativeComponents;

namespace SampleAddIn
{
    public sealed class Initializer: IAssemblyInitializer
    {
        // Whenever the GC user loads a .NET assembly (DLL file) into GC, GC examines all
        // of the classes defined within that assembly, looking for those classes that...
        //
        // 1. Are public, and...
        //
        // 2. Implement the interface, IAssemblyInitializer, and...
        //
        // 3. Have a default constructor (that is, a public constructor that takes no arguments).
        //
        // For each such class it finds, GC instantiates it, automatically. (Then, GC does nothing
        // further with that instance. That class's sole worth is whatever it does in its
        // constructor.)

        public Initializer()
        {
            ScriptFunctions.Load(); // In this case, we're using the initialization mechanism
                                   // to load the script function we defined in the static
                                   // class, ScriptFunctions.
        }
    }
}
```