ModelCenter Component Plug-In Getting Started Guide

Introduction

Welcome to the ModelCenter Component Plug-In Getting Started Guide. This document intends to help Plug-In developers who would like to extend the capabilities of ModelCenter by adding connections to external data and analysis tools like Excel and Databases.

To obtain access to the ModelCenter Component Plug-In SDK, please contact Ansys ModelCenter support.

1.1. Definitions

- Builder UI – The UI portion of the Plug-In which is responsible for allowing a user to edit a PACZ.
- Component – A workflow element that has inputs, some black box execution, and produces outputs. Often used interchangeably with the term Analysis.
- Component Plug-In – A Plug-In to the ModelCenter framework that implements a Runner and optionally a Builder UI.
- Driver – A workflow element that "drives" or controls other sub-workflows. If this is what you want to do, you are in the wrong document.
- PACZ – The standard file format for a harnessed external component. The PACZ may be stored compressed as a zip, or uncompressed as a set of files in a directory. The PACZ contains a common metadata definition, `component.pacj`, which sufficiently describes the instance so that it can be generically managed in an "app store" and jobs can be submitted without needing to instantiate the required plug-in.
- Runner – The non-UI portion of the Plug-In which is responsible for batch execution.
- Workflow – An automated set of Components, Drivers, and other control elements that together compute some type of engineering simulation.

2. Getting Started (.NET)

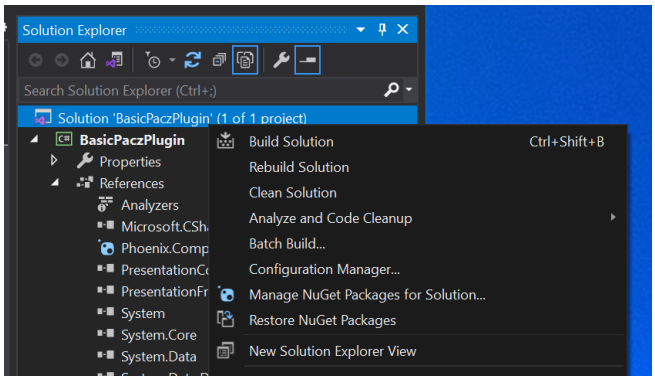
2.1. Install the TestUI Application

- Start from the "artifacts" folder in the `CAMpluginSDK` zip file.
- Move the downloaded zip file to your development machine.
- Locate the "TestUI" folder which contains the **TestUI.exe** application needed later. This location will later be referred to as `[TestUI_Install_Directory]`.

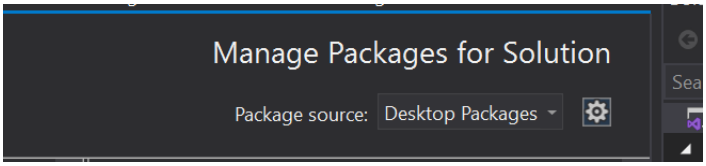
NOTE: The TestUI is not a production tool. It is meant to be used merely as a testing and troubleshooting tool.

2.2. Configure Visual Studio to use Repository

1) Inside Visual Studio 2019, right-click the solution item in Solution Explorer. Then select the option for "Manage NuGet Packages for Solution...":

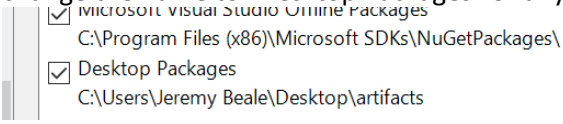


2) Click the gear icon to bring up settings:

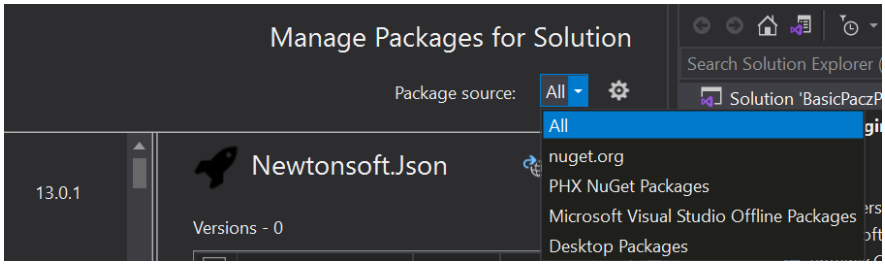


3) Click the **+** icon to add a new package source:

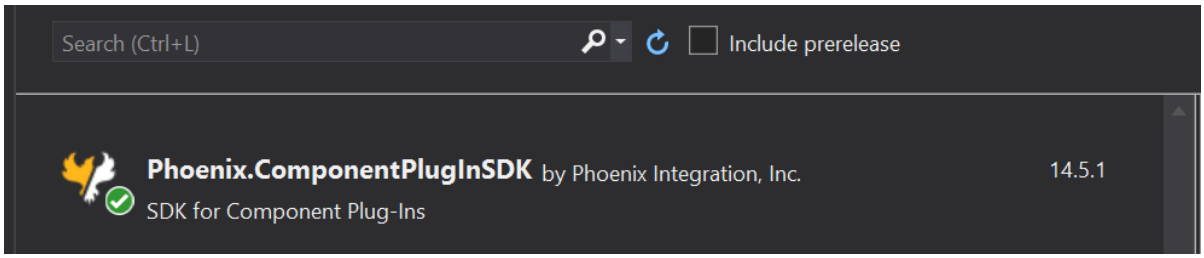
4) Change the Source text field to point to the "artifacts" folder in the `CAMpluginSDK` zip file. Also change the name to "Desktop Packages" or any other arbitrary string.



5) After accepting changes, you should now see "Desktop Packages" as an option in the Package Source dropdown menu:



6) Click "Install" for `Phoenix.ComponentPlugInSDK` for Visual Studio:



2.3. Install the Component Plug-In Wizard for Visual Studio

> Requires Visual Studio 2019.

1) Adjacent to the TestUI folder downloaded in Section 2.1 is the `ComponentPlugInSDKTemplate` folder.
2) Double-click `Phoenix.ComponentPlugInSDKTemplateWizard.vsix` file.
3) Complete the VSIX Installer Wizard.

2.4. Create a Plug-In Project

1) In Visual Studio, create a new Project.
2) On the "Create a new project" page:
   a) Select the template that was installed earlier: "Pacz Plug-In Template".
   b) Click **Next**.
3) On the "Configure your new project" page:
   a) Enter the Project name. This name will be used as the default namespace and plug-in name. This name will be referred to as `[Name]` throughout this document.
   b) Choose the project location.
   c) For Framework, choose at least .NET Framework 4.6.2
   d) Click **Create**.
4) On the "New Pacz Plug-In Project Wizard" dialog:
   a) Choose "Variable Based".
   b) Click **OK**.

   Visual Studio will create a new project for you.

> NOTE: Visual Studio 2019 has a bug where the project may take some time before building as it resolves the package dependencies. If the project references do not show "Phoenix.ComponentPlugInSDK", try the following steps to fix references:
>
> 1. Right-click solution in Solution Explorer.
>
> 2. Choose "Restore NuGet Packages".
>
> 3. Close and reopen solution.

3. New Project Overview (.NET)

3.1. 2 Project Files

1. `[Name].cs` implements `IHarnessRunner`
    a. This is the class that is used to execute runs of the plug-in.
    b. There should be no UI associated with this class.
    c. There are 2 Public Methods that will be called:
        i. `ConstructAsync`
            1. Always called first.
            2. Provides the HarnessRunnerHost
            3. Initial setup and loading from the config should be done here.
        ii. `RunAsync`
            1. Used to execute a run.
            2. Provides inputs to evaluate.
            3. Update output dictionary with execution results.
2. `[Name]BuilderUI.cs` extends `AbstractVariableBasedBuilderUI<[Name]Runner>`
    a. This is the class that holds the UI of the plug-in.
    b. It builds on top of a pre-built UI form that allows for editing variables.
    c. The `ComponentName` property should be overridden to provide a display name for the plug-in.
    d. There are 5 base class methods that can be overridden.
        i. `LoadFromPaczAsync`: Used to load the plug-in instance.
        ii. `GetTreeProperties`: Used to customize the properties tree.
        iii. `GetFileLoadProperties`: Configures the built-in file open feature.
        iv. `SetupView`: Used to add `menuItems` to the `mainMenu`.
        v. `SaveToPaczAsync`: Used to save the plug-in instance.

3.2. AbstractBuilderUI Base Class

The `AbstractBuilderUI` base class contains some properties and methods that are useful:

1. Properties:
    a. `Host (HarnessBuilderUIHost)`
        i. Properties
            1. `ExtractedPacz`: The pacz object and config.
            2. `Logger`: Provides logging for the plug-in.
        ii. Methods
            1. `CallRunnerAsync`
            2. `TestRunAsync`
            3. `SaveAsync`
2. Methods:
    a. `AddMenuItem` and `AddAsyncMenuItem`(requires reference to `PresentationCore`)
    b. `SelectVariables`: Opens the CommonSelectVariables Form.
    c. `SetPaczIcon`

3.3. VariableBasedBuilderViewModel

This class provides access to the variables in the plug-in. It is provided as an argument to the method called after a new file is opened and to the `SetupView` method. Since this is provided to the `SetupView` method, it can also be made available to the functions that are called when custom `menuItems` are clicked.

4. Testing the Plug-In

4.1. Test with TestUI

1) Build the plug-in project.
2) Copy the Debug folder (from `[PlugIn Dir]/bin`) to the Plug-Ins folder of the TestUI App.
3) Run the TestUI.
4) New Component.
5) Plug-Ins / Debug / `[Name]`
6) You will now get the same new component wizard as you would in MCD. Note that unlike in MCD, you can cancel out of this and it will generate a completely in-memory instance backed with temp folders. This enables you to debug without building up piles of files

After making changes to the project, rebuild and replace the directory in the TestUI App, then rerun the TestUI App.

An advanced user may want to use `'mklink'` to create a soft link from the TestUI Plug-Ins folder to project compile output so that they do not need manually copy after each build.

By default TestUI runs the plug-in in process. This is advantageous because you can attach a debugger before you launch your plug-in. However, MCD always runs the plug-in out of process. Simulate this in TestUI by running it with the `-oop` command line option.

4.2. Test with ModelCenter 14.5+

1) Install ModelCenter version 14.5 or higher.
2) Copy the Debug folder (from `[PlugIn Dir]/bin`) to the Plug-Ins folder of the ModelCenter installation. Typically located in `[C:\Program Files\Phoenix Integration\ModelCenter {version}\Plug-Ins]`.
3) Open ModelCenter. Save the workflow to allow use of CAM components.
4) Locate your plugin from `Server Browser > component plug-in > {your plug-in's name}`.
5) Drag the plug-in icon into the workflow to launch the new component wizard.

5. Customizing Your Plug-In

5.1. Creating the UI

5.1.1. Name the plug-in.

Set `BuilderUI.ComponentName`

5.1.2. LoadFromPaczAsync

Called when the plug-in is first created or loaded. State should be loaded from the passed in IExtractedPacz. Most of the state is likely stored in the Config.Properties and the Config.InstanceFiles. Variables will be saved and loaded as part of the viewModel; it may not be necessary to load them here.

Note that the Model object (type VariableBasedBuilderModel) is directly accessible in this function's scope and can be used to create input/output variables. The following example shows inputs and outputs being created if none exist when the LoadFromPaczAsync(...) function is called:

```
protected override async Task LoadFromPaczAsync(IExtractedPacz extractedPacz)
{
    if (Model.InputVariables.Count == 0)
    {
        // Create the initial set of variables for the view
        Model.MoveInputVariablesFrom(_createInputs());
        Model.MoveOutputVariablesFrom(_createOutputs());
    }
    await Task.CompletedTask;
}

private IEnumerable<IRuntimeVariable> _createInputs()
{
    RuntimeVariable x1 = new RuntimeVariable("x1", VariableType.Real, new RealValue(0.5));
    RuntimeVariable x2 = new RuntimeVariable("x2", VariableType.Real, new RealValue(1.75));

    IEnumerable<IRuntimeVariable> inputs = new List<IRuntimeVariable>() { x1, x2 };
    return inputs;
}

private IEnumerable<IRuntimeVariable> _createOutputs()
{
    RuntimeVariable y = new RuntimeVariable("y", VariableType.Real, new RealValue(0.01));

    IEnumerable<IRuntimeVariable> outputs = new List<IRuntimeVariable>() { y };
    return outputs;
}
```

5.1.3. GetFileLoadProperties

If instances of the plug-in will use an existing file, usually from another application, as part of the setup or run evaluation, override GetFileLoadProperties to enable the built-in file load dialog. Create a FileLoadProperties object with an action that will be called when the user selects a new file and a file filter for the open dialog to use. The file will likely need to be copied to the Host.ExtractedPacz.ExtractionFolder and added to the Config.InstanceFiles.

An example of this would be choosing a CAD file for a CAD plug-in. The builder may then interrogate this file in the FileLoadProperties Action to determine which inputs and outputs to add to the plug-in.

5.1.4. SelectVariables

In some plug-ins, users will create all the variables manually in the variable tree, but usually a better option is for the plug-in to assist with variable creation. This can be done in the Action called when the build in file dialog is used to select a file or from a menu button press.

This can be done entirely without user input by adding the variables into the viewModel.Variable List or the SelectVariables form can be used to allow some customization of the available variables.

To use the SelectVariables form, you must pass lists of available inputs and outputs. Call the SelectVariables method of the AbstractBuilderUI, and it will provide a form with the variables to allow the user to filter and select the variables they wish to include.

5.1.5. GetTreeProperties

Allows customization of the variable tree.

- `componentName`: Name to display at the root of the `VariableTree`.
- `canAddRemove`: If the UI controls can be used to add and remove variables.
  If variables are created programmatically or using `SelectVariables`, this can allow user from creating additional variables or editing the properties of the programmatically created ones.
- `hasNamedVariables`: If the variables use a named variable property.
  `NamedVariables` allow variables to have a mapping to another name that can be used by the plug-in. An example of this could be that a plug-in that uses Excel could use a named variable to provide a mapping to a range. The variable's name in the plug-in may be `cost` but could map to a range in Excel. The plug-in could use the range when communicating with Excel, while the variable is displayed as "Cost".
- `namedVariableDisplayName`: The display name of named variables.
  This defines the label to use for the `namedVariable` field. In the example using Excel, this might be "Range".
- `namedVariableToken`: The token to use for named variables in the Text Based Variable Editor.
  This is like the `namedVariableDisplayName` but is used in sterilization (by the text based variable editor) and must be alphanumeric characters.

5.1.6. SetupView

This method is called after the form is created and allows customization. This is typically done by adding menu items to the main Menu. By default, `VariableBasedPlugIns` will have an **Apply** button that saves the plug-in instance. Additional menu items can be added as either buttons or containers of subItems.

Use the `AddMenuItem` method of the `AbstractBuilderUI` base class to add items and provide an event handler as the action to perform on click. The `VariableBasedBuilderViewModel` can be made available to use in the event handler if desired.

Common usages of additional menu items are to show an options dialog or a help page for the plug-in. The following code sample shows a simple Windows Form made accessible through a custom options menu item. No further code is required to make a Form1 object accessible to the user.

```
public class BasicPaczPluginBuilderUI : AbstractVariableBasedBuilderUI<BasicPaczPluginRunner>

{

…

        protected override void SetupView(IExtractedPacz pacz, VariableBasedBuilderViewModel
        viewModel, Menu mainMenu)
        {
            AddMenuItem(parent: mainMenu, header: "Options", imageType: ImageType.OPTIONS,
               hasDownArrow: false, eventHandler: (s, e) => _editOptions(viewModel));
        }

        private void _editOptions(VariableBasedBuilderViewModel viewModel)
        {
            Form1 form1 = new Form1();
            Form1.Show();
        }

        …
}

public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

5.1.7. SaveToPaczAsync

This is called to allow the plug-in to save any necessary information into the PACZ. Most settings will be stored as name-value pairs in the `pacz.Config.Properties` dictionary. Instance files may also need to be updated. Variables are part of the `ViewModel` and will be saved and loaded by the `ViewModel`; typically, there is no need to do anything with the variables in `SaveToPaczAsync`.

6. Creating the Runner

6.1. ConstructAsync

Will always be called first allowing initialization of the runner. Most plug-ins will want to capture the `IHarnessRunnerHost` that is passed in to use later. If execution relies on resource intensive or slow-to-load applications, it may be advisable to delay loading the application until the first run.

6.2. RunAsync

Called to perform a run evaluation. Input variables are passed in with a value and a validity. Plug-Ins that do not support invalid inputs can use `SafeValue` to ensure that all input values are valid. After evaluation, the outputs dictionary should be updated with new a new `VariableState` for each output.

7. Common Issues/Solutions

7.1. User Variable metadata and display tags

Many plugins will map variables in the ModelCenter world to objects in the external system. For example, each Excel Plug-In variable will be mapped to a particular workbook range. Instead of storing this mapping themselves, the plug-in should use the existing user variable tags/metadata to consistently handle this.

7.2. Threading

The runner is run on a single, unique, operating system thread and all calls to it are serialized and proxied to this thread for execution. This provides a sane environment from which the plug-in can safely operate without having to worry about thread safety. If the Plug-In writer wishes to use multiple threads, they can start those threads themselves and deal with the ensuing thread synchronization issues.

The plug-in writer may also safely use Task Asynchronous Programming (TAP, or async/await). We install a `SynchronizationContext` so that all continuations will happen by default on the single thread. If the plug-in writer wishes to let work run on the thread pool, they can use `Task.Run()` or `.ContinueWith(false)` calls. For more information, refer to Microsoft's task based asynchronous pattern documentation.).

The builder UI is also run on a single, unique, UI thread and all calls to it are serialized and proxied to this thread for execution. This thread is different than the runner thread. All calls from the UI to the Runner should happen through `IHarnessBuilderUIHost.CallRunnerAsync()`.

8. Frequently Asked Questions

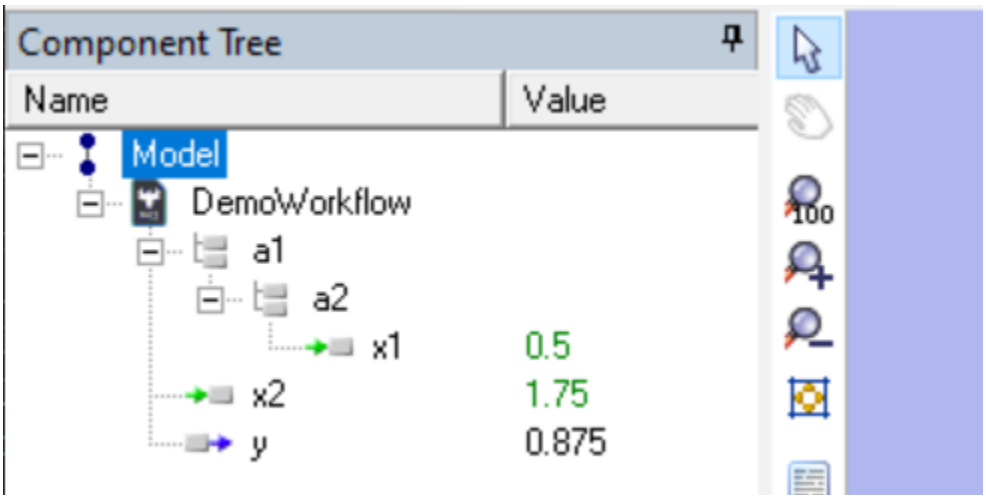
Q1. How do I programmatically define a ModelCenter assembly in the Component Tree?

A1. Write nested variable names using "`.`" separators:

```
1 reference
private IEnumerable<IRuntimeVariable> _createInputs()
{
    RuntimeVariable x1 = new RuntimeVariable("a1.a2.x1", VariableType.Real, new RealValue(0.5));
    RuntimeVariable x2 = new RuntimeVariable("x2", VariableType.Real, new RealValue(1.75));

    IEnumerable<IRuntimeVariable> inputs = new List<IRuntimeVariable>() { x1, x2 };
    return inputs;
}
```

ModelCenter takes care of assembly creation automatically:



Then use the same name in the RunAsync method to access the nested variable:

```
0 references
public async Task RunAsync(IReadOnlyDictionary<string, VariableState> inputs, VariableVa
{
    //TODO: Run the component, set the outpus as a function of the inputs
    // e.g.
    double x1 = (RealValue)inputs["a1.a2.x1"].SafeValue;
    double x2 = (RealValue)inputs["x2"].SafeValue;
    outputs["y"] = new VariableState(new RealValue(x1*x2));
```

Q2. What characters are allowed for plugin variable names?

A2. ModelCenter variables use the same convention as Java variables, except dollar signs are not supported.

Essentially alphanumeric + underscore, but first character must be either a letter or underscore.

Periods are only supported as the separator between different levels of a hierarchy.

Spaces, punctuation, and special characters are not allowed.

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html

Q3. How do I access `Value` and `DefaultValue` from the Runner context?

A3. Access both read-only values as follows:

```
1 reference
87  private IEnumerable<IRuntimeVariable> _createInputs()
88  {
89      RuntimeVariable x1 = new RuntimeVariable("a1.a2.x1", VariableType.Real, new RealValue(0.5));
90      RuntimeVariable x2 = new RuntimeVariable("x2", VariableType.Real, new RealValue(1.75));
91
92      IEnumerable<IRuntimeVariable> inputs = new List<IRuntimeVariable>() { x1, x2 };
93      return inputs;
94  }
```

Accessing `DefaultValue` is done with integer index rather than name. The user must search (LINQ function or traditional iterator) through the available indices to find the desired name / `DefaultValue`. You are technically reading directly from the PACZ file created during the Builder context.

Note that Value can only be accessed from the Runner context. There is no access to Value from the Builder context. In the Builder context, only `DefaultValue` can be set, which is later copied to create the variable's Value. This copy process occurs in ModelCenter's core process outside the Plug-In environment.

Q4. How do I create an array variable?

A4. Using a two-item boolean array as an example:

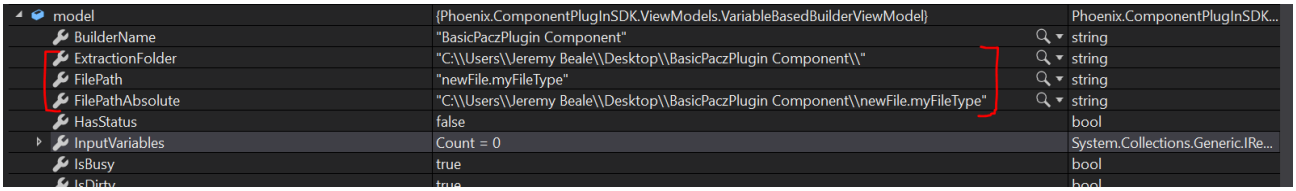
```
bool[] val = { true, false };

BooleanArrayValue booleanArrayValue = val;

RuntimeVariable boolVar = new RuntimeVariable("bool_arr_var", VariableType.BooleanArray,
booleanArrayValue);
```

Q5. Is it possible to query relative path from inside the plugin?

A5. The plugin's "model" object is aware of both relative and absolute paths –



Note that the "FilePath" property will change depending on whether the path is inside or outside the extraction folder:

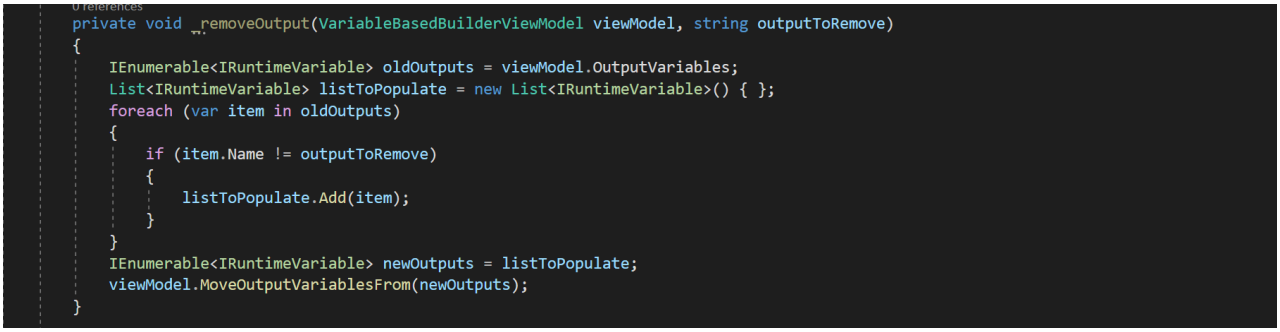
```
//
// Summary:
//     View model for the VariableBasedBuilderView.
public class VariableBasedBuilderViewModel : AbstractPlugInViewModel<VariableBasedBuilderModel>
{
    public VariableBasedBuilderViewModel(VariableBasedBuilderModel model, ILogger logger, string componentName, Func<Varia

    public DelegateCommandAsync OpenFileCommand { get; }
    //
    // Summary:
    //     The folder where the files exist. May be a temporary folder, or the original
    public string ExtractionFolder { get; }
    //
    // Summary:
    //     The File path to a file the Builder can open. If the path is within the extraction
    //     folder this will return a relative path, otherwise the path will be an absolute
    //     path to the file.
    public string FilePath { get; set; }
    //
    // Summary:
    //     The Absolute Path to the Phoenix.ComponentPlugInSDK.ViewModels.VariableBasedBuilderViewModel.FilePath
    public string FilePathAbsolute { get; }

    protected override void _onInputsChanged();
    protected override void _onOutputsChanged();
}
}
```

Q6. How do I remove a single input/output variable from the model?

A6. Call the following function as follows: **_removeOutput(viewModel, "x2");**



```
0 references
private void _removeOutput(VariableBasedBuilderViewModel viewModel, string outputToRemove)
{
    IEnumerable<IRuntimeVariable> oldOutputs = viewModel.OutputVariables;
    List<IRuntimeVariable> listToPopulate = new List<IRuntimeVariable>() { };
    foreach (var item in oldOutputs)
    {
        if (item.Name != outputToRemove)
        {
            listToPopulate.Add(item);
        }
    }
    IEnumerable<IRuntimeVariable> newOutputs = listToPopulate;
    viewModel.MoveOutputVariablesFrom(newOutputs);
}
```


```
private void _removeOutput(VariableBasedBuilderViewModel viewModel, string
outputToRemove)
{
	IEnumerable<IRuntimeVariable> oldOutputs = viewModel.OutputVariables;
	List<IRuntimeVariable> listToPopulate = new List<IRuntimeVariable>() { };
	foreach (var item in oldOutputs)
	{
		if (item.Name != outputToRemove)
		{
			listToPopulate.Add(item);
		}
	}
	IEnumerable<IRuntimeVariable> newOutputs = listToPopulate;
	viewModel.MoveOutputVariablesFrom(newOutputs);
}
```

Q7. How do I define a file as an output variable?

A7. During output list construction you would have:

```
private IEnumerable<IRuntimeVariable> _createOutputs()
{
	RuntimeVariable y = new RuntimeVariable("y", VariableType.Real, new
RealValue(0.01));
	RuntimeVariable outvar = new RuntimeVariable("NASTRANMAP", VariableType.File,
FileValue.CreateFromString(null, null, "c:\\thermal\\stress\\case1.txt", null));

	IEnumerable<IRuntimeVariable> outputs = new List<IRuntimeVariable>() { y,
outvar };
	return outputs;
}
```

During execution time you would have:

```
public async Task RunAsync(IReadOnlyDictionary<string, VariableState> inputs,
VariableValueScope outputs, CancellationToken cancellation)
{
	//TODO: Run the component, set the outputs as a function of the inputs
	// e.g.
	double x1 = (RealValue)inputs["a1.a2.x1"].SafeValue;
	double x2 = (RealValue)inputs["x2"].SafeValue;
	outputs["y"] = new VariableState(new RealValue(x1*x2));

	FileValue outvarOld = (FileValue) outputs["NASTRANMAP"].SafeValue;
	outputs["NASTRANMAP"] = new
VariableState(FileValue.ReadFromFile(outvarOld.OriginalFileName));

	await Task.CompletedTask;
	//throw new NotImplementedException("Run method has not been implemented.");
}
```

Q8. From the runner context, I can iterate through input variables using `foreach (var key in inputs.Keys)`. I need to determine whether each input is type string or type real. I currently have `(StringValue)inputs[key].SafeValue`. That crashes when it is a real. Is there a way to check what type of input it is?

A8. There is a method `SafeValue.GetModelCenterType`. It returns a string which is either "string" or "double".

Q9. How to create a multi-dimensional ModelCenter array?

A9. Example using integers:

```
long[,] intArr = new long[3, 4] {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
IntegerArrayValue intArrVal = new IntegerArrayValue(intArr);
RuntimeVariable x3 = new RuntimeVariable("x3", VariableType.IntegerArray, intArrVal);
```

Q10. How to add a description to an `IRuntimeVariable` instance?

A10. See "x3" variable definition in the `BasicPaczPlugin` example (provided).