
Last edition 16.10.2017

# Quick programming guide

# Introduction

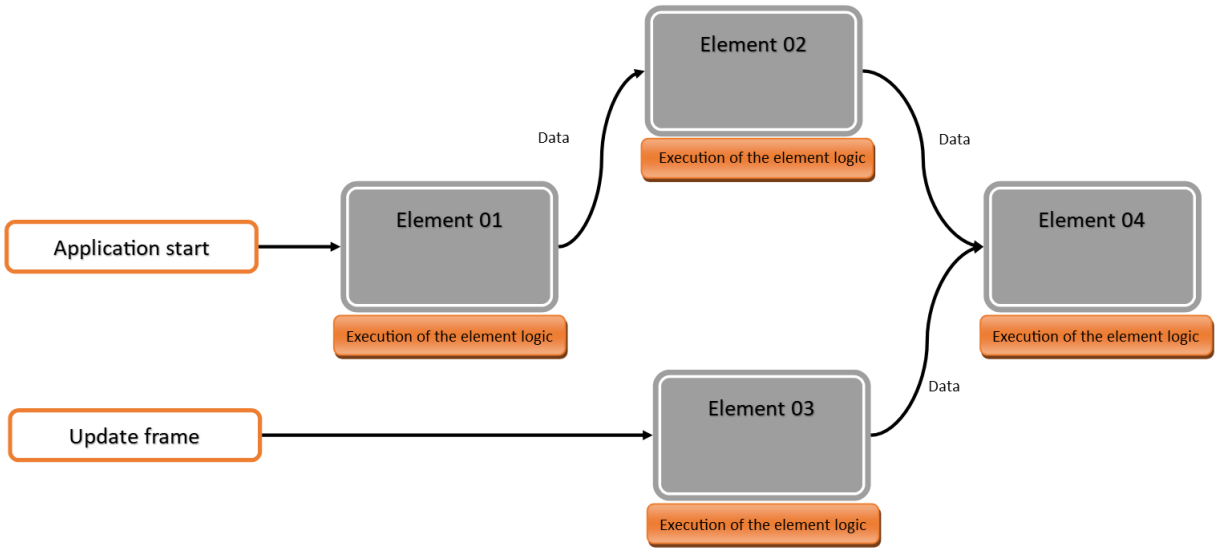
Welcome to the programming guide for diagram elements and their visual representations for the Panthea VS system. This section of the documentation will describe the process of creating new elements, variables, and redefining the visual representation for them.

# Programming the logic of elements

## Introduction

Panthea VS has many ready-made elements of different specialization, which cover a wide range of tasks necessary for developing the application. In many ways, this provides sufficient flexibility for novice users. However, for complex projects with unique features, an individual approach is often required. To achieve this goal, Panthea VS allows user to implement custom diagram elements, which provides the developer with infinite extensibility in terms of its use.

Below is a generalized scheme for describing the system from the position of the code.




**Note**: in this case, application start and update of frames are also processed in the elements, they are just isolated for better perception.

Element is a part of the application logic that takes certain data to the input, carries out operations on them, and outputs the result of these operations. In the particular case, the element may not receive data, but issue it and vice versa. Based on this, the diagram is a chain of receiving and transferring data between elements.

Each element is a collection of input points (which may not exist), a set of output points (which may also not exist) and logic that processes the data, or implements some functional. When creating links between elements, a reference is established at the output point of one element to the input point of another element, by passing a reference to the method of the element class.

What logic can be enclosed in an element? Absolutely anything, from a simple mathematical operation, to the full implementation of the game. Everything depends on the desire and imagination of the programmer.

## Diagram element

As mentioned above, the element is part of the application logic, which generally accepts and transmits data. From the programming point of view, this is a class that inherits from abstraction and this inheritance is mandatory.

### Definition of an element

To define an element, you must create a class and inherit it from the **Element** abstraction, which in turn is inherited from **ScriptableObject**.

**Example**:

```
public class CustomElement : Element
{

}
```

Despite the simplicity, this class will already be automatically defined as an element and it can be added to the diagram. In this case, its description will be formed from the default values. In order to personalize the description of the element, you must use the **ElementDefinition** attribute.

**Example**:

```
[ElementDefinition(Name = "MyCustom", Path = "MyFolder/MySubfolder", Tooltip = "this my
element", Color = VSEColor.Green)]
public class CustomElement : Element
{

}
```

**Notes**:

**Path** – the path that will be used to display the element in the catalog.

**Color** – color in HTML format ("RRGGBBAA" HEX string).

### Definition of input points

Input points are the part of the element that determines the transfer of data to it, or initializing the beginning of its operation. In the particular case, the element may not have input points and work by itself.

Input points are divided into two types - points with data transfer and points without data transfer. From the position of the code, it is a class that is a wrapper over a reference to a handler method. For the first type, this class is **INPUT_POINT**, for the second type it is the generic class **INPUT_POINT <T>,** where T is the type of data to receive and process (T - has no use restrictions).

**Example of input point definitions:**

Definition of an input point without data transmission

```
public INPUT_POINT SimpleInputPoint = new INPUT_POINT();
```

The definition of an input point with the transfer of data of type int

```
public INPUT_POINT<int> IntegerInputPoint = new INPUT_POINT<int>();
```

After defining the input points, you must set the handler for each of them. For more information, see the «Element initialization» chapter.

## Definition of output points

Output points are the parts of an element that determine the transfer of data from it, or inform about the occurrence of an event. In the particular case, the element may not have output points.

The output points are divided into two types - points with data transfer and points without data transfer. From the position of the code, it is a class with one single method that causes data to be transferred. For the first type, this class is **OUTPUT_POINT**, for the second type it is the generic class **OUTPUT_POINT <T>**, where T is the data type for the transfer (T - has no use restrictions).

**Example of output point definitions:**

Definition of an output point without data transmission

```
public OUTPUT_POINT SimpleOutputPoint = new OUTPUT_POINT();
```

The definition of an output point with the transfer of data of type int

```
public OUTPUT_POINT<int> IntegerOutputPoint = new OUTPUT_POINT<int>();
```

**Example of calling the output point:**

Calling the output point without data transmission:

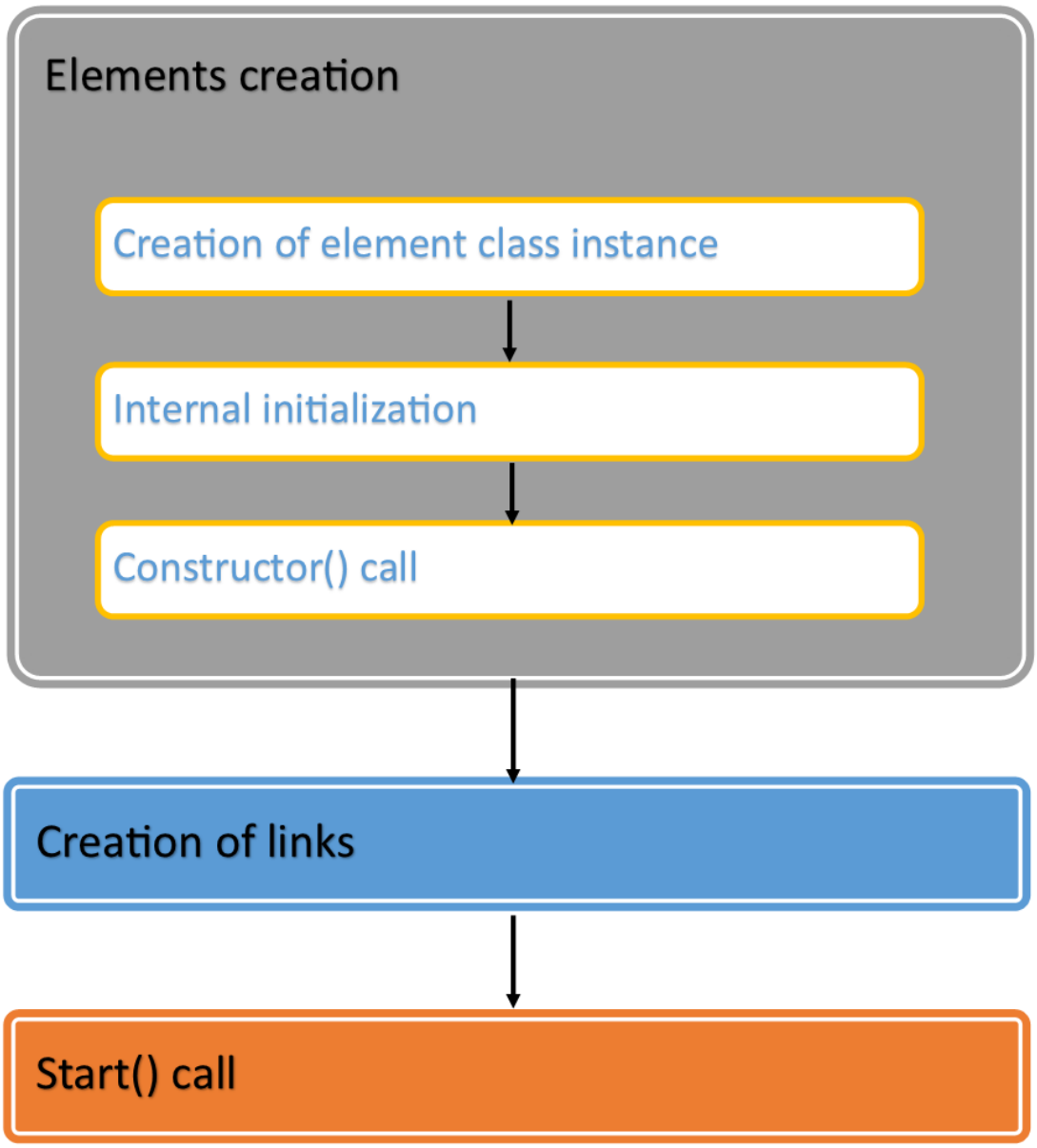
```
void MySimpleFunction()
{
	SimpleOutputPoint.Execute();
}
```

Calling the output point with data transmission:

```
void MyDataFunction()
{
	IntegerOutputPoint.Execute(5);
}
```

## Element initialization

On the figure below the diagram loading process is shown.




Initialization of all elements occurs in several stages.

1. Primary internal initialization of an element after creating an instance of the class. At this point, the element gets a reference to the root **MonoBehavior** object (for more details, read this in the «Coroutine» section).
2. Initialization of input point handlers. This process is performed using the **Awake** analog method, which in the Panthea VS environment is called **Constructor** (this is a virtual method that you need to redefine for an element.) You do not need to call this method from the parent class.) Also, an override is not required if the element does not contain input points.

   **Important**: you can't use Awake in the code of the elements! Because all the elements are **ScriptableObject** and element instances are created in the diagram editor, at the time the scripted object is created, Awake is automatically called from it, which in turn leads to unwanted errors.

**Important**: initialization of the methods of input point handlers must be done only in **Constructor**, because at the time of creating links between the elements, all handlers must be known.

3. Call the **Monobehaviour Start** method. It is carried out after creation of links between elements. This method is used if you need to perform some operations on the diagram variables referenced by the element. Read more in the relevant section.

**Example of initialization of input point handlers:**

```
[ElementDefinition(Name = "MyCustom", Path = "MyFolder/MySubfolder", Tooltip = "this my
element", Color = VSEColor.Green)]
public class CustomElement : Element
{
	public INPUT_POINT SimpleInputPoint = new INPUT_POINT();
	public INPUT_POINT<int> IntegerInputPoint = new INPUT_POINT<int>();

	public override void Constructor()
	{
		SimpleInputPoint.Handler = HandlerSimpleInput;
		IntegerInputPoint.Handler = HandlerIntegerInput;
	}

	private void HandlerSimpleInput()
	{

	}

	private void HandlerIntegerInput(int data)
	{

	}
}
```

## Monobehavior methods

To make it easier for programmers who work with Unity3d to adapt to the development of elements, and to increase flexibility, Panthea VS supports calling a number of familiar MonoBehaviour methods. Currently available:

- Start
- Update
- LateUpdate
- FixedUpdate

In addition to the classical use of these methods, there is an additional feature in Panthea VS, which is absent in Unity3d - ability to set the calling order. This functionality is implemented using the attribute **ExecuteOrder**.

**Example**:

```
[ExecuteOrder(Order = 1)]
void Start()
{

}
```

**Order** – this is the value that determines the order in which methods are called. The methods with the lowest value are called first.

**Note**: methods that do not have an attribute will be called the most recent.

## Coroutine

It was said above that after creating an instance of an element class, its internal initialization takes place, in which a reference to the root MonoBehavior object is passed. This action is necessary to enable the use of **Coroutine** within the element. A reference to this object is available in the element class through the protected property **CoroutineHost**.

**Example**:

```
void CallCoroutine()
{
	CoroutineHost.StartCoroutine(MyCoroutine());
}

IEnumerator MyCoroutine()
{
	yield return new WaitForSeconds(1f);
}
```

**Note**: since all elements refer to the same **MonoBehavior** object, all the coroutines of all the elements are executed in it, like the **MonoBehaviour** methods described in the previous section.

## Dynamic definition of input and output points

In addition to the mechanism described above for determining input and output points, there is an opportunity for their dynamic creation. This mechanism can be useful for the possibility of configuring points depending on the parameters of the element.

The creation of points is carried out as before, through the creation of instances of the class. In this case, the input points must still be initialized by handlers in **Constructor()**. However, for the framework to recognize points and correctly create links when loading a diagram, the class of the element must implement specialized interfaces. For input points, this interface is **IInputPointParse**, and for output points **IOutputPointParse**.

**Example**:

```
private Dictionary<string, object> _inputPoints = new Dictionary<string, object>();

public override void Constructor()
{
	_inputPoints.Clear();

	for (var i = 0; i < NumberOfInputPoints; i++)
	{
		var inputPoint = new INPUT_POINT<int>();

		inputPoint.Handler = (value) =>
		{
			Debug.Log(value);
		};

		_inputPoints.Add("Value {0}".Fmt(i + 1), inputPoint);
	}
}

public Dictionary<string, object> GetInputPoints()
{
	return _inputPoints;
}
```

In this example, the code that allows you to display points in the diagram editor is skipped, for more details about this read in the programming section for visual representations of elements.

### Cleaning an element

If in the work process of the element the data was created, that requires manual removal after the application is finished or the scene is unloaded, then it is sufficient to implement the **IDisposable** interface. The **Dispose()** call occurs on the Unity3d **OnDestroy** event. An alternative option is to use the **OnDestroy** and **OnDisable** methods directly (remember that all elements are ScriptableObject).

## Diagram variable

Even though the diagram variable is inherently an element that does not contain input and output points, this part of the Panthea VS framework was isolated into a separate entity. This was done for better visual perception when developing diagrams and interacting with them when developing the elements code. Here are the several features of the variables:

1. Diagram variables are an entity that stores data that can be shared between elements.
2. The variables defined in the diagram can't be accessed in other diagrams. To transfer data between diagrams, other features are used. This is done on purpose, in order to avoid a deep and strong relationship between the diagrams, which would complicate the templating and loading them from the outside.
3. Variables can contain any data of any formats. The only requirement is serializability.

### Definition of a variable

The definition of a variable is almost the same as an element, except that all variables must be inherited from the generic class **Variable <T>**, where T is the type of data that the variable stores.

**Example**:

```
[ElementDefinition(Name = "Int", Path = "Variable/Base", Tooltip = "variable of int
type", Color = VSEColor.Cyan)]
public class VariableInt : Variable<int> { }
```

### Reference to a variable

To provide access to a diagram variable from an element, you must set a reference to it. This can be done using a specialized generic class **VARIABLE_LINK <T>**, where T is the data type in the variable.

**Example**:

```
public VARIABLE_LINK<int> Data = new VARIABLE_LINK<int>();
```

**Important**: references to variables in Panthea VS relate to the links between the elements, so they will not be initialized when the **Constructor()** method is called. If some preparatory actions are required with the data of the variables, they must be performed in the **Start** method.

Example:

```
public VARIABLE_LINK< int > IntVariable = new VARIABLE_LINK< int >();

void Start()
{
	IntVariable.Value = 10;
}
```

### Events of changing and setting a variable

During the operation of the element logic, you may need to be able to handle events related to changes in the values of the diagram variables, if the element refers to them. This can be done in two ways, the first is the automatic addition of subscribers at the time of the link creation (special fields in the class **VARIABLE_LINK <T>** are used for this purpose), the second is the subscription to events during the work manually.

*Automatic subscription at the time of creating a link*

**Example for setting a value for a variable (automatic subscription):**

```
public VARIABLE_LINK<int> Data = new VARIABLE_LINK<int>();

public override void Constructor()
{
	Data.HandlerSetEvent = (value)=>
	{
		Debug.Log(value);
	};
}
```

**Example for variable value change event (automatic subscription):**

```
public VARIABLE_LINK<int> Data = new VARIABLE_LINK<int>();

public override void Constructor()
{
	Data.HandlerChangedEvent= ()=>
	{
		Debug.Log("Value was changed");
	};
}
```

**Important**: you can't change automatically created handlers during the execution of the logic of the element, otherwise there will be a memory leak.

**Important**: Automatically created subscriptions are automatically deleted when the application finishes running, or when the scene is unloaded.

*Manual subscription to events*

A direct subscription is used if there is no requirement for subscribers to exist at the time when the Start methods are called for the elements, or before the start of the operation of some element. Also, a direct subscription should be used, if in the process of element working you need to get rid of the subscription.

**Important**: you can't use direct subscription to events in the **Constructor()** method, because at the time of its call, no variable references are created yet.

**Example of direct subscriptions to events:**

```
public VARIABLE_LINK<int> Variable = new VARIABLE_LINK<int>();

void Start()
{
	Variable.AddChangedEventHandler(ChangedEventHandler);
	Variable.AddSetEventHandler(SetEventHandler);
}

void ChangedEventHandler()
{
	Debug.Log("value was changed");
}

void SetEventHandler(int value)
{
	Debug.LogFormat("value was set to {0}", value);
}
```

**Example of manual unsubscription from events:**

```
void RemoveHandlers()
```

```
{
	Variable.RemoveChangedEventHandler(ChangedEventHandler);
	Variable.RemoveSetEventHandler(SetEventHandler);
}
```

**Note**: if the event handlers are not manually deleted, then when the application finishes, or when the scene is unloaded, they will be deleted automatically.

### Working with Unity3d objects

Since all diagrams and serialized data of elements are stored in external files, the use of classical references to scene objects through the Unity3d variables of types is impossible, since it will be impossible to restore these references from the outside.

To solve this problem, Panthea VS has a specialized wrapper **VSObject**, which can store and restore links to any Unity objects (scene objects, components, prefabs, resources). Therefore, if the element code requires a reference to such things, you must use a variable of type **VSObject**. Then, to get a reference to the Unity object directly, you need to use the generic method **Get <T>**, where T is the Unity type. If an object of type T is not available, the method returns null.

**Example**:

```
public VSObject SceneObject;

void MyFunction()
{
	var transform = SceneObject.Get<Transform>();
}
```

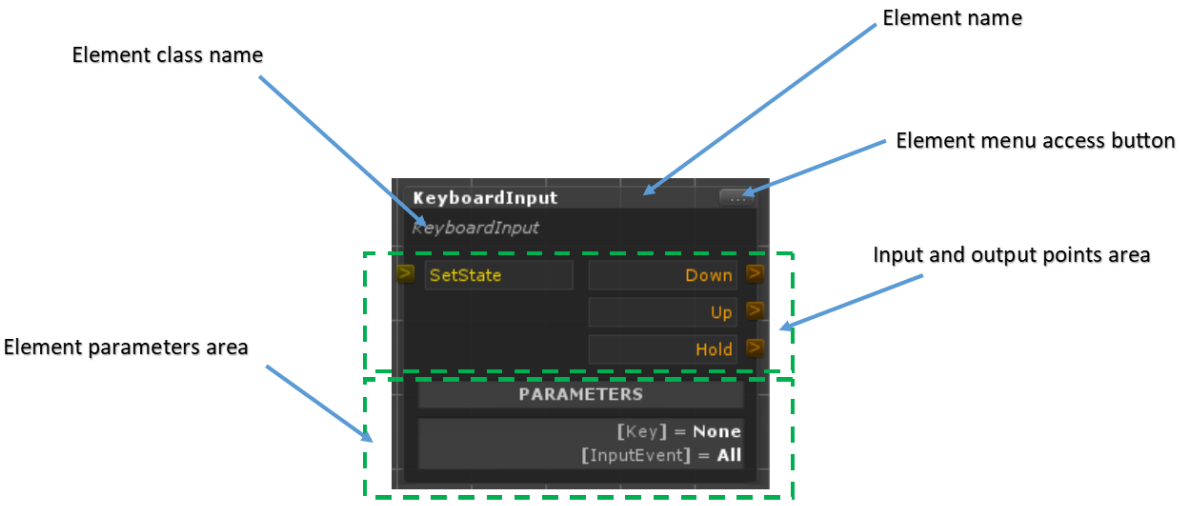
## Programming visual representations of elements

### Introduction

In the current version of the Unity3d development environment, there is an important feature that allows programmers to redefine what the property editor of a component looks like. This mechanism is implemented based on the namespace **UnityEditor**, using **Editor** and **PropertyDrawer** classes. A similar mechanism is also present in Panthea VS framework, which will allow you to redefine the visual representation of the various parts of the element in the diagram, as well as display its properties in the inspector.

### Diagram element

The visual representation of an element is what it looks like when you add it to the diagram display area. For a better understanding of the issue below is a drawing with a description of the parts of which the element consists visually.



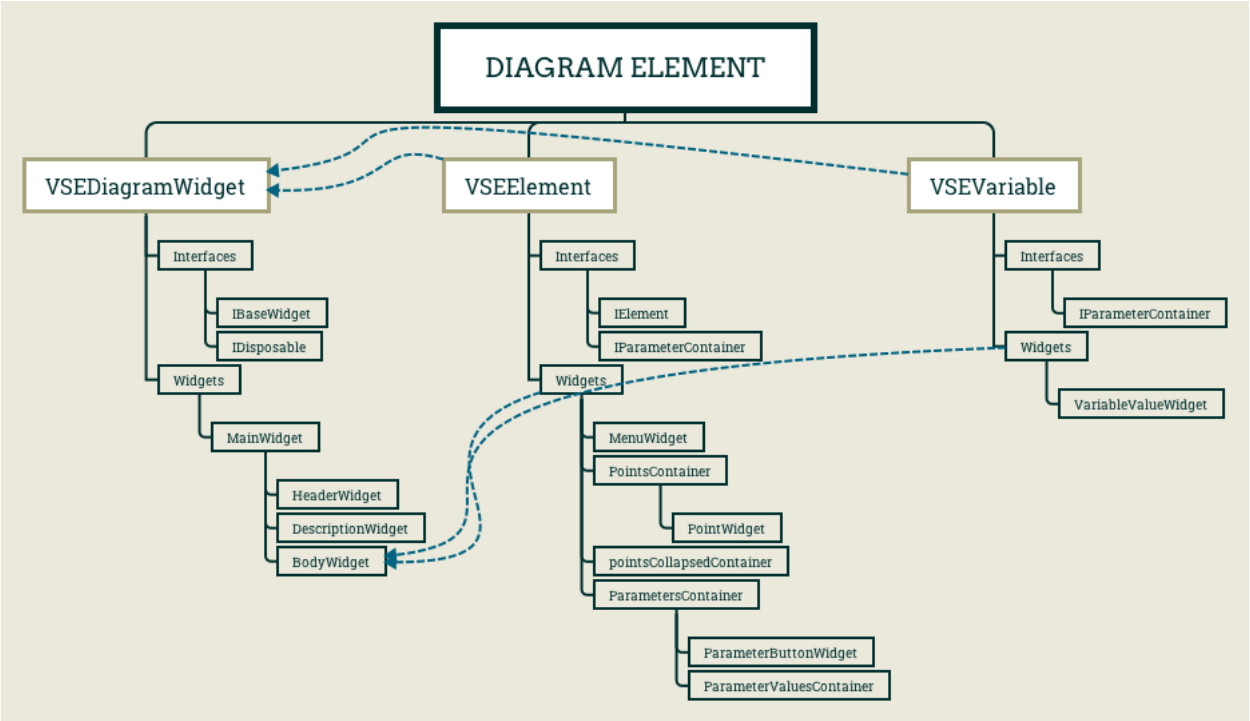

Below, the customization of each of the parts will be described, here are some basic principles first.

To redefine the editor (visual representation) for an element, the same conditions as for creating custom editors in Unity3d are required:

1. The class that defines the visual representation of the element must be in the folder named Editor.
2. For a class to be interpreted as a new editor for an element, it must be marked with a special attribute **VSECustomElementDrawer** specifying the type of the element class for which this editor will be applied.
   **Example**:

```
[VSECustomElementDrawer(typeof(Elements.SendColor))]
public class VariableColorCustomDrawer : VSEVariable
{
}
```

For a better understanding of how the code for drawing elements is arranged, a flowchart is shown below.




Here:

- **VSEDiagramWidget** – a base class for all elements
- Widgets are separate visual parts of the element responsible for strictly defined functionality.
- All widgets implement the **IChildWidget** interface
- All container widgets implement the **IWidgetContainer** interface
- For convenience, there are abstractions that implement the functionality of the container widget (**WidgetElementContainer**) and the widget of the child (**ChildWidget**). All widgets that are described in the schema are their heirs.
- **VSEElement** and **VSEVariable** are inherited from **VSEDiagramWidget**
- All specialized widgets of elements and variables are placed in the **BodyWidget** container.

For more information about classes, see the API documentation section.

In most cases, the developer does not need to deeply interfere with the visual representation of the element, and simpler development methods are used. They will be discussed below.

With a simple way to define an element editor, inherit from the **VSEElement** class is used, this class is the default base class that is used to draw an element in the diagram editor.

**Example of a class declaration:**

```
[VSECustomElementDrawer(typeof(Elements.DebugElement))]
public class MyCustomDrawer : VSEElement
{
	public CustomDrawer(DiagramStorage.ElementsStorage.ElementData elementStorageData)
	: base(elementStorageData)
	{

	}
}
```

## Change the drawing of the element name

To change the appearance of the element name, you need to replace the widget responsible for drawing it. A reference to this widget is provided to the developer via the protected property **headerWidget**.

To define a new widget, you must create a class inherited from the **HeaderWidget** class.

Example:

```
public class CustomHeaderWidget : VSEDiagramWidget.HeaderWidget
{
	public CustomHeaderWidget(IBaseWidget baseWidget) : base(baseWidget)
	{

	}
}
```

Here, **baseWidget** is a reference to an interface that gives access to data on an element from the data storage.

If you want to change the font or color of the element name, then you should override the **RegisterStyle** method, which creates new drawing styles before the first drawing call.

Example:

```
public override void RegisterStyle()
{
	if(headerStyle == null)
	{
		headerStyle = new GUIStyle(GUI.skin.label);
		headerStyle.fontStyle = FontStyle.BoldAndItalic;
		headerStyle.fontSize = 14;
		headerStyle.alignment = TextAnchor.MiddleCenter;
	}
}
```

If you need to completely customize the drawing, then you should override the function **Draw**.

Example:

```
public override void Draw()
{
	EditorGUI.MaskField(widgetRect, 0, options);
}
```

To apply the changes to the element name widget, you must replace the default widget class with the help of protected generic **ReplaceWidget** method. This must be done in the overridden method **PrepareChildWidget**, which is responsible for creating widgets.

**An example of a class that changes the name of a Debug element:**

```
[VSECustomElementDrawer(typeof(DebugElement))]
public class MyCustomDrawer : VSEElement
{
    public class CustomHeaderWidget : VSEDiagramWidget.HeaderWidget
    {
        public CustomHeaderWidget(IBaseWidget baseWidget) : base(baseWidget)
        {
        }

        public override void RegisterStyle()
        {
            if (headerStyle == null)
```

```
            {
                headerStyle = new GUIStyle(GUI.skin.box);
                headerStyle.fontStyle = FontStyle.BoldAndItalic;
                headerStyle.fontSize = 14;
                headerStyle.alignment = TextAnchor.MiddleCenter;
            }
        }
    }

    public MyCustomDrawer(DiagramStorage.ElementsStorage.ElementData elementStorageData)
: base(elementStorageData)
    {
    }

    protected override void PrepareChildWidget()
    {
        ReplaceWidget<HeaderWidget, CustomHeaderWidget>();

        base.PrepareChildWidget();
    }

}
```

### Change the drawing of the element class

Changing the appearance of the name of an element class can be done in the same way as the name of the element. To define a new widget, you need to create a class inherited from **DescriptionWidget** class. The class is replaced in the same way as described in the example in the previous paragraph.

### Change the drawing of input and output points

Changing the appearance of the input and output points of the element occurs in the same way as the name of the element. To define a new widget, you must create a class inherited from PointWidget class. The class is replaced in the same way as described in the example above.

### Change the drawing of the element parameters

Use the technics above, but with **ParameterValueWidget** class.

### Change the drawing of references to variables

Use the technics above, but with **VariableLinkWidget** class.

### Change the remaining widgets

You can change the rest of the elements used for drawing in the same way, these include:

- **BodyWidget** – element body widget.
- **ParameterButtonWidget** – widget of the button minimizing the parameters area.
- **ParametersContainer** – container for the entire parameter area.
- **ParameterValuesContainer** – container for a list of parameter values.
- **PointsContainer** – container for input and output points
- **MenuWidget** – element menu widget.

### Configuring a set of input and output points

Previously, it was pointed out that a developer can dynamically create input and output points, depending on the parameters of the element. In addition to dynamically creating points, which we will talk about later, it is possible to configure the points that are created immediately. To do this, it is necessary for the element to implement one or both **IInputPointParse** and **IOutputPointParse** interfaces.

**Example**:

```
public INPUT_POINT<int> SetExternal = new INPUT_POINT<int>();
public INPUT_POINT SetInternal = new INPUT_POINT();
public INPUT_POINT Get = new INPUT_POINT();

public OUTPUT_POINT CompleteSet = new OUTPUT_POINT();
public OUTPUT_POINT<int> ReturnValue = new OUTPUT_POINT<int>();

public HideSetGetFlags HideFlag;

public int InternalValue;

public override void Constructor()
{
	SetExternal.Handler = HandlerExternalSet;
	SetInternal.Handler = HandlerInternalSet;
	Get.Handler = HandlerGet;
}


private void HandlerExternalSet(int data)
{
	SetValue(data);

	CompleteSet.Execute();
}

private void HandlerInternalSet()
{
	SetValue(InternalValue);

	CompleteSet.Execute();
}

private void HandlerGet()
{
	ReturnValue.Execute(GetValue());
}

public Dictionary<string, object> GetInputPoints()
{
	var returnInputPoint = new Dictionary<string, object>();

	switch (HideFlag)
	{
		case HideSetGetFlags.HideSet:
			returnInputPoint.Add("Get", Get);
			break;
		case HideSetGetFlags.HideGet:
			returnInputPoint.Add("SetExternal", SetExternal);
			returnInputPoint.Add("SetInternal", SetInternal);
			break;
		case HideSetGetFlags.None:
			returnInputPoint.Add("SetExternal", SetExternal);
			returnInputPoint.Add("SetInternal", SetInternal);
			returnInputPoint.Add("Get", Get);
			break;
	}

	return returnInputPoint;
}

public Dictionary<string, object> GetOutputPoints()
{
	var returnOutputPoint = new Dictionary<string, object>();
```

```
        switch (HideFlag)
        {
                case HideSetGetFlags.HideSet:
                        returnOutputPoint.Add("ReturnValue", ReturnValue);
                        break;
                case HideSetGetFlags.HideGet:
                        returnOutputPoint.Add("CompleteSet", CompleteSet);
                        break;
                case HideSetGetFlags.None:
                        returnOutputPoint.Add("ReturnValue", ReturnValue);
                        returnOutputPoint.Add("CompleteSet", CompleteSet);
                        break;
        }

        return returnOutputPoint;
}
```

As for the dynamic creation of points, in this case, it is also necessary to implement interfaces, but the implementation of its functions will be more complex.

**Example**:

```
private Dictionary<string, object> _inputPoints = new Dictionary<string, object>();

public override void Constructor()
{
        _inputPoints.Clear();

        for (var i = 0; i < NumberOfInputPoints; i++)
        {
                var inputPoint = new INPUT_POINT<int>();

                inputPoint.Handler = (value) =>
                {
                        Debug.Log(value);
                };

                _inputPoints.Add("Value {0}".Fmt(i + 1), inputPoint);
        }
}

public Dictionary<string, object> GetInputPoints()
{
#if UNITY_EDITOR
        Action fillInputPoints = () =>
        {
                for (var i = 0; i < NumberOfInputPoints; i++)
                {
                        _inputPoints.Add("Value {0}".Fmt(i + 1), new INPUT_POINT<int>());
                }
        };

        if (!UnityEditor.EditorApplication.isPlayingOrWillChangePlaymode)
        {
                _inputPoints.Clear();

                fillInputPoints();
        }
        else
        {
                if(_inputPoints.Count == 0)
                {
                        fillInputPoints();
                }
        }
```

```
#endif
        return _inputPoints;
}
```

The template of this example is mandatory, otherwise errors may occur when debugging a diagram in the editor. To avoid such errors the UNITY_EDITOR wrapper is used, which takes into account the various working conditions of the diagram editor.

## Diagram variable

A diagram variable is a simplified version of an element. It does not contain input and output points, and essentially contains only one additional widget that displays its value. Since the type of the variable can be as complex as you like, you may need to modify the widget to display it correctly. This is done in a similar way, as for an element that is not a variable. The only condition is that you also need to override the method updating the value of the variable.

**An example of a class that changes the drawing of a variable of type Color:**

```
[VSECustomElementDrawer(typeof(Elements.VariableColor))]
public class VariableColorCustomDrawer : VSEVariable
{
        public class VariableColorValueWidget : VariableValueWidget
        {
                public VariableColorValueWidget(VariableValueData valueData) :
        base(valueData) { }

                public override void RegisterStyle() { }
                public override void UpdateView() { }
                public override void Draw()
                {
                        Handles.BeginGUI();
                        Handles.DrawSolidRectangleWithOutline(valueAreaRect,
(Color)valueData.Value, Color.white);
                        Handles.EndGUI();
                }
        }

        Public VariableColorCustomDrawer(VSCore.DiagramStorage.ElementsStorage.ElementData
elementStorage) : base(elementStorage) { }

        protected override void PrepareChildWidget()
        {
                ReplaceWidget<VariableValueWidget, VariableColorValueWidget>();

                base.PrepareChildWidget();
        }

        public override void UpdateParameterValues()
        {
                var propertyInfo = VSEditorUtils.GetVariable(InstanceType);

                valueData.Value = propertyInfo.GetValue(Instance, null);
        }

}
```

## Modify the element properties editor

To change the element properties editor, the same mechanism is used as in Unity3d, i.e. creating a class inherited from **PropertyDrawer**. Read more in the Unity3d documentation.

**Note**: In addition, standard Unity3d attributes applied to fields and properties, such as Header, Tooltip, Range, etc., are supported.

### Displaying the parameters of an element in the diagram editor

For some element parameter (public property or field) to be displayed in the diagram editor, it must be marked with a special attribute **ViewInDiagram**.

**Example**:

```
[ViewInDiagram]
public int Param;
```

In case the parameter of the element is a complex data structure that needs to be displayed in the diagram editor, for correct display it is necessary that the structure (class) implement the **IWrapperData** interface.

**Example**:

```
[Serializable]
public class DiagramMessageDefinition : IWrapperData
{
	public int Id;

	[SerializeField]
	private string _name;

	public KeyValuePair<string, string> GetInfo()
	{
		return new KeyValuePair<string, string>("Message", _name);
	}
}
```

### Hiding references to variables from the diagram editor

By default, if an element contains references to diagram variables, they will automatically be displayed in the diagram editor. In certain cases, this will not be necessary. To hide the reference, just apply the **HideInDiagram** attribute to the field.

**Example**:

```
[HideInDiagram]
public VARIABLE_LINK<int> Variable = new VARIABLE_LINK<int>();
```

### Advanced programming

If the developer needs to completely change the drawing of the element, there are two options for this:

1. Direct inheritance from **VSEDiagramWidget** and manual implementation of drawing input and output points, element parameters, menus, etc.
2. A more complicated version is the direct implementation of the **IBaseWidget** and **IElement** interfaces, in which case all the code for the implementation of the element drawing and interaction with the Panthea VS editor should be written manually.

## Communication between a diagram and an external code.

During the development of applications, you may need to connect the logic implemented in the diagrams and external code written by programmers. This connection can be a notification, either as a transfer or receipt of any data.

In the simplest version, you can do it using a special element that sends messages with the identifiers set (**SendMessageElement** and **ReceiveMessageElement**, for details refer to the documentation for the elements). In this case, if you need to send data or notification to the diagram, then in the code you need to use the **DiagramMessage** class.

**Example**:

```
private void SendDataMessage (int messageId, object value)
{
	DiagramMessage.Call(new DiagramMessage(messageId, value));
}
```

To retrieve data from the diagram, you must provide a subscription to a message of type **DiagramMessage**.

**Example**:

```
[GlobalEvent.HandlerEvent]
private void OnDiagramMessage(DiagramMessage ev)
{
}
```

Examples above use the global events (read the API documentation for details). The same mechanism can be used for own events to link a diagram and an external code. It will be necessary to write the code of the element that will work with such events (messages).

And as another option, it is possible to write a special element that will link a diagram and code without using global events, for example through singletons, delegates or through direct links to MonoBehaviour scripts.

## Launching a diagram from code

During the development of applications, you may need to be able to launch a diagram from external code. To do this, use the **RunDiagramExternal** method from the **DiagramController** class (see API reference for details). This method takes as a parameter a link to the diagram storage, which you can get by downloading a diagram from a text asset using the **LoadDiagram** method from the **DiagramStorage** class (see API reference for details).

**Example of loading a diagram from AssetBundle**:

```
void Start()
{
    StartCoroutine(LoadDiagramFromUrl("MyAssetBundleUrl", "ExternalDiagram"));
}

IEnumerator LoadDiagramFromUrl(string url, string diagramName)
{
    var www = new WWW(url);

    yield return www;

    if (string.IsNullOrEmpty(www.error))
    {
        var diagramBinaryData = www.assetBundle.LoadAsset<TextAsset>(diagramName);

        if (diagramBinaryData != null)
        {
            var storage = DiagramStorage.LoadDiagram(diagramBinaryData);

            DiagramController.Instance.RunDiagramExternal(storage);
        }
    }
}
```

In case you need to launch a diagram as an instance, you should use the **RunDiagramInstance** method from the **DiagramController** class (see API reference for details). If the diagram does not

automatically call the command about its completion or its completion is tied to the event that is processed by the code, in this case, the diagram is stopped and resources are cleared using the **StopDiagramInstance** method from the **DiagramController** class (see API referencefor details).