The Megawidget Framework:

A Reference for Focal Points

Table of Contents

label (empty string)
leftMargin (0)
lines (6)
maxChars (0)
maximumTime (no practical limit)
maximumVisibleTime
maxValue (no practical limit)
minimumInterval (0)
minimumTime (0 (indicating Jan 1, 1970 00:00 GMT))
minimumTimeInterval (1 minute)
minimumVisibleTime
minValue (no practical limit)
numColumns (1)
numericOnly (false)
pages
    numColumns (1)
    pageName
    pageFields
precision (1)
preferredWidth (width of text)
promptText (empty string)
rightMargin (0)
scrollable (false)
selectedLabel (empty string)
sendEveryChange (true)
showAllNoneButtons (true)
showScale (false)
spacing (0)
spellcheck (false)
timeDescriptors (None)
topMargin (0)
unitChoices
valueEditables (None)
valueIfEmpty (empty string)
valueLabels (None)
values (default value(s) for specific type of megawidget)
valueUnit ("ms")
visibleChars (same as maxChars, unless latter is 0, in which case defaults to 20)
visiblePage (first page)

Overview

The *megawidget* framework provides a set of user interface widgets for displaying and manipulating data within dialog boxes via a common set of interfaces. Megawidgets are themselves made up of SWT controls, but they isolate the user from many of the chores of SWT widget layout and configuration, performing much of the grunt work themselves.

Megawidgets may be used within Hazard Services as follows:

- Recommenders: If desired, a recommender may be written to display a dialog box populated by megawidgets in order to collect information from a user before executing.
- Product Generators: If the process of generating a particular product requires additional information, megawidgets may be displayed to collect said information within the Product Staging Dialog.
- Hazard Event Metadata: Different types of hazard events may include different associated metadata. This metadata changes not only with type, but also with the status of the hazard event. The Hazard Detail View (also known as the Hazard Information Dialog) displays megawidgets allowing event metadata to be viewed and manipulated.

Megawidgets are created by specifying their types and parameters within *definitions* in Python code. (They may be instantiated and used within Java classes as well, but Java-based megawidget manipulation beyond the scope of this document.) A collection of definitions providing the contents of a recommender dialog, product generator panel, or metadata panel, including definitions of any child megawidgets, is referred to hereafter as the *total set*.

Definitions

Definitions hold the various *properties* of the megawidgets to be created. Each such property is expressed as a key-value pair within the dictionary making up the definition. An example of a definition for a button megawidget might read:

```
{
      "fieldName": "reset",
      "fieldType": "Button",
      "label": "Reset"
}
```

This would result in the creation of a Button megawidget. The properties in this case provide, respectively:

- the unique identifier of the megawidget
- the type of the megawidget (a button)

- the text to be displayed on the button face

Properties: Mutable vs. Immutable

As discussed above, each megawidget has a set of properties. When a megawidget is first created, its properties, mutable or immutable, are assigned to the values given by that definition. So, in the Button example above, the property `label` has the value "Make Polygon"; thus, the Button megawidget created by that definition will have this text displayed within its body.

Two important properties that must be provided for each megawidget are `fieldName` and `fieldType`. The former serves as the identifier for that megawidget; it must be unique within the total set, meaning that no other megawidget within the same total set may use that same identifier. The latter is used to specify the type of the megawidget to be created.

Many of these properties are *immutable*, meaning that they cannot be changed once the megawidget is created. For example, a megawidget's `fieldName` property cannot be changed; the megawidget has the same identifier throughout its lifetime.

However, a subset of the properties for a given megawidget are *mutable*. These may be altered during the megawidget's lifetime. An example of a mutable property is the `enable` property: During a megawidget's lifetime, it may be enabled or disabled. Such manipulation is possible via the use of *interdependencies*, discussed later in this document.

General Categories

Each type of megawidget falls into at least one of the following general categories.

Stateful

Most megawidgets are *stateful*, meaning that they hold some sort of value as their state. An example is a Text megawidget, which has the text string found within its text entry field as its state. Likewise, an IntegerSpinner's state is the integer found within its entry field. Each state has an identifier, so that when a dictionary of states is supplied to the total set of megawidgets to set their values, or such a dictionary is provided by the megawidgets, the states may be ascertained by finding the values associated with the identifiers.

Most stateful megawidgets only hold one state at a time. For these, the `fieldName` functions as its state's identifier. Certain types of megawidgets, however, are capable of holding more than one state. The TimeScale may have one or more states; each of a given TimeScale's thumbs provides a timestamp, and each timestamp corresponds to a state. For such megawidgets, the state identifiers are derived from the `fieldName` by treating the latter as a colon-separated list of state identifiers.

Thus, for example, if a TimeScale megawidget definition has the `fieldName` of "impactTime", it will have one thumb associated with a single state, the latter having the identifier "impactTime". If instead the megawidget has as its `fieldName` "riseAbove:crest:fallBelow", it would have three thumbs, each associated with a different state identifier, which would be, in order, "riseAbove", "crest", and "fallBelow".

When stateful megawidgets experience a state change, any interdependency script that has been provided with the total set is executed with those megawidgets' `fieldName` values as the list of trigger identifiers.

Notifier

Some megawidgets function only to trigger interdependency scripts. Unlike stateful megawidgets, they have no state. The Button is the canonical example of such a megawidget. When pressed, any interdependency script that was supplied with the total set is executed with the megawidget's `fieldName` value as the trigger.

Container

Container megawidgets are designed merely to hold other megawidgets as children. Examples include the Group and TabbedComposite. Each such megawidget has either one list of children, or, if it has multiple pages (such as the TabbedComposite), one list of children per page. Such megawidgets are used to visually group related megawidgets together and, for multi-paged ones, to save space by displaying only one page at a time.

Detailed

Detailed megawidgets are those that are themselves stateful, but which allow the option of having child megawidgets function as detail fields for their states. Examples include the CheckBoxes, RadioButtons, TimeRange, and TimeScale.

Decorative

Decorative megawidgets have no state, cannot be invoked, and hold no children; they simply serve to explain or decorate the panel in which they reside. (*Note: at this time, there is only one such type.*) The Label is an example. They are the simplest megawidgets.

Available Types

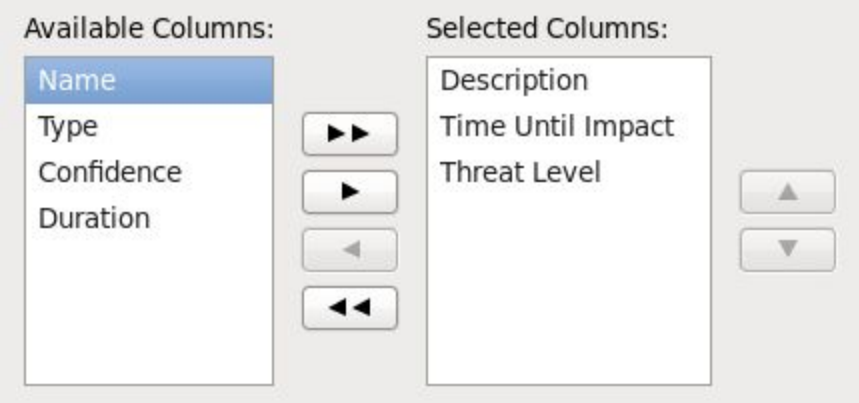
The following types of megawidgets are available. Using the given type as the value for `fieldType` in the definition dictionary indicates that this sort of megawidget is to be created. Each type is annotated with the categories into which it falls, and includes a list of the mutable and immutable properties that it has. Any property that has no default value, and thus must be specified, has its name given in **bold**.

BoundedListBuilder (stateful)

This megawidget is applicable to situations in which there is a bounded set of options from which the user must be able to select zero or more choices, like a CheckBoxes or CheckList; however, unlike said megawidgets, it allows the user to order the choices. It consists of two scrollable lists side by side, one holding the available (unselected) choices, and one holding the currently selected choices, with buttons allowing the reordering of the latter list, and the selecting and deselecting of individual choices. It allows the dragging and dropping of items within the latter list, as well as between the two lists.

Example:

```
{
      "fieldType": "BoundedListBuilder",
      "fieldName": "boundedListBuilder1",
      "label": "Available Columns:",
      "selectedLabel": "Selected Columns:",
      "choices": [ "Name", "Type", "Description", "Threat Level",
      "Time Until Impact", "Confidence", "Duration" ],
      "values": [ "Description",  "Time Until Impact", "Threat
      Level" ],
      "expandHorizontally": True,
      "expandVertically": True
 }
```



Immutable: **fieldName**, **fieldType**, interdependencyOnly, label, lines, selectedLabel, spacing, width

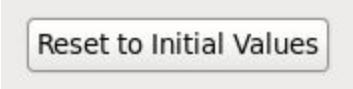
Mutable: **choices**, editable, enable, extraData, values

Button (notifier)

This is a simple command button. Unlike most megawidgets, it has no state, but like stateful megawidgets, it may be used to trigger the execution of an interdependency script.

Example:

```
{
      "fieldType": "Button",
      "fieldName": "button1",
      "label": "Reset to Initial Values"
}
```



Immutable: **fieldName**, **fieldType**, label, spacing, width

Mutable: enable, extraData

CheckBox (stateful)

The CheckBox is a simple toggle, allowing the display of a particular label associated with a boolean value as state.

Example:

```
{
      "fieldType": "CheckBox",
      "fieldName": "checkBox1",
      "label": "Include surrounding counties",
      "values": True
}
```

Immutable: expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, spacing, width

Mutable: editable, enable, extraData, values

CheckBoxes (stateful, detailed)

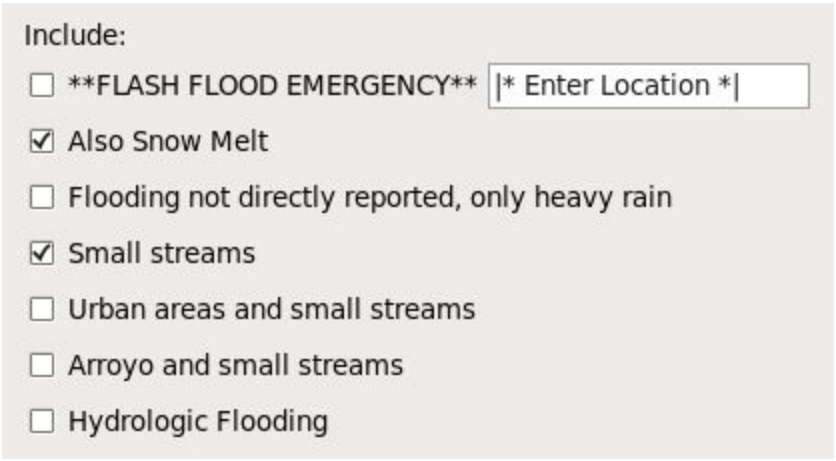
This is comprised of a list of checkboxes, each representing one of the possible choices available to the user for its state. If provided with a label, the latter appears as a descriptive

string above the checkboxes. The megawidget's state is a set of zero or more chosen choices. Like the CheckList, it is used in situations where the user may want to choose multiple (non-mutually-exclusive) options. Unlike a CheckList, it allows the inclusion of zero or more child megawidgets providing detail fields for any of the choices represented by its checkboxes.

Example:

```
{
      "fieldType":"CheckBoxes",
      "fieldName": "checkBoxes1",
      "label": "Include:",
      "choices": [
            {
                  "identifier": "ffwEmergency",
                  "displayString": "**FLASH FLOOD EMERGENCY**",
                  "detailFields": [
                        {
                              "fieldType": "Text",
                              "fieldName": "text1",
                              "expandHorizontally": True,
                              "maxChars": 40,
                              "visibleChars": 20,
                              "values": "|* Enter Location *|"
                        }
                  ]
            },
            {
                  "identifier": "+SM",
                  "displayString": "Also Snow Melt"
            },
            {
                  "identifier": "-FL",
                  "displayString": "Flooding not directly reported,
                  only heavy rain"
            },
            {
                  "identifier": "+SS",
                  "displayString": "Small streams"
            },
            {
                  "identifier": "+US",
                  "displayString": "Urban areas and small streams"
            },
            {
                  "identifier": "+AS",
                  "displayString": "Arroyo and small streams"
            },
```

```
                {
                        "identifier": "+HA",
                        "displayString": "Hydrologic Flooding"
                }
        ],
        "values": [ "+SM", "+SS" ]
}
```



Immutable: **choices**, expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, spacing, width

Mutable: editable, enable, extraData, values

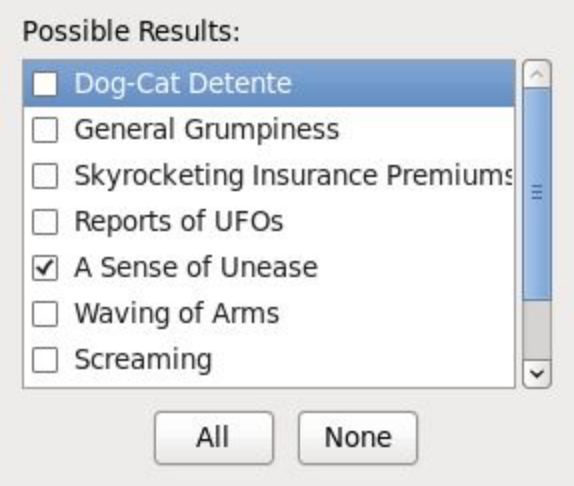
CheckList (stateful)

This is a scrollable list containing one or more checkable choices. If provided with a label, the latter appears as a descriptive string above the list box. It is identical in purpose and state to the CheckBoxes type, except that it may be used in situations where the number of choices is too great for the vertical space available, and thus scrolling is required. An advantage over the CheckBoxes megawidget is that its list of choices is mutable.

Example:

```
{
        "fieldType": "CheckList",
        "fieldName": "checkList1",
        "label": "Possible Results:",
        "choices": [
                "Dog-Cat Detente",
                "General Grumpiness",
                "Skyrocketing Insurance Premiums",
                "Reports of UFOs",
```

```
                "A Sense of Unease",
                "Waving of Arms",
                "Screaming",
                "Talking to Neighbors",
                "Existential Crises"
        ],
        "values": [ "A Sense of Unease" ],
        "expandHorizontally": True,
        "expandVertically": True
}
```



Immutable: **`fieldName`**, **`fieldType`**, `interdependencyOnly`, `label`, `lines`, `showAllNoneButtons`, `spacing`, `width`

Mutable: **`choices`**, `editable`, `enable`, `extraData`, `values`

ComboBox (stateful)

This is a drop-down selector containing one or more choices, and allowing a single choice to be chosen. If provided with a label, the latter appears as a descriptive string to the left of the combo box. Its state is the currently selected choice. Like the RadioButtons, it is used in situations where the user may want to choose exactly one option from a list of choices. One advantage over the RadioButtons megawidget is that its list of choices is mutable; another is that it allows `autocomplete` to be used, helpful when the list of choices is very large.

Example:

```
{
        "fieldType": "ComboBox",
        "fieldName": "comboBox1",
        "label": "Level of Threat:",
```

```
        "choices": [ "Moderate", "Severe", "Extinction" ],
        "values": "Severe",
        "expandHorizontally": True
}
```

Immutable: autocomplete, expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, spacing, width

Mutable: **choices**, editable, enable, extraData, values

Composite (container)

This is a simple undecorated widget with the sole purpose of holding child megawidgets, akin to the SWT composite. It may be constructed with scrollbars that are displayed when the child megawidgets take up too much space to fit if desired. Any label provided is ignored.

Immutable: bottomMargin, columnSpacing, expandHorizontally, expandVertically, **fieldName**, **fields**, **fieldType**, fullWidthOfDetailPanel, leftMargin, numColumns, rightMargin, scrollable, spacing, topMargin, width

Mutable: extraData

DetailedComboBox (container, detailed, stateful)

Like the ComboBox, this is a drop-down selector containing one or more choices, and allowing a single choice to be chosen. Its state is the currently selected choice. However, it differs from the ComboBox in that the drop-down selector, and its optional label, are superimposed over an SWT Group, acting as the latter's visual header instead of the label usually found atop a group.

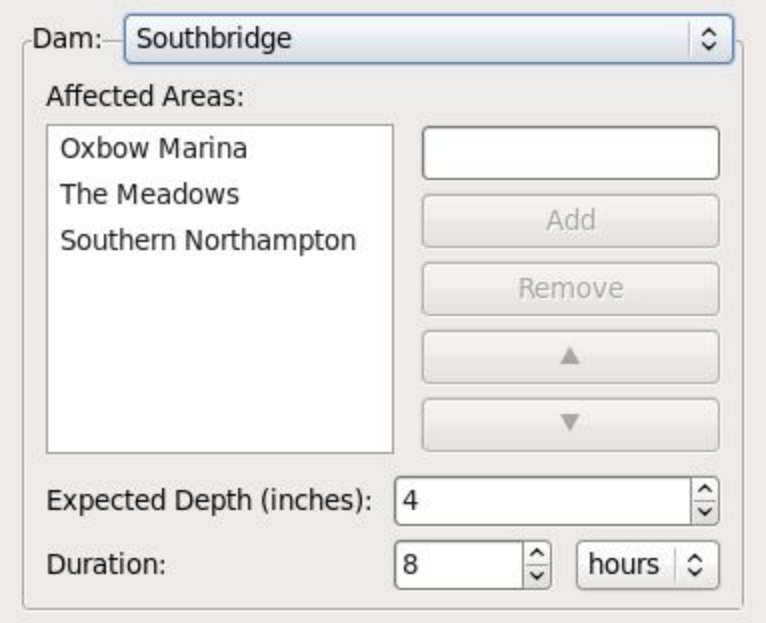
The group displays within itself any detail fields that are specified as associated with the currently selected choice within the combo box, optionally with scrollbars if the detail fields take up too much room to fit. This may be preferable to a detail-field-equipped RadioButtons when the detail fields for at least one of the choices will take up significant amounts of space, since only a single choice's details are shown at any given time within the group. Unlike the ComboBox, the list of choices is not mutable.

Example:

```
{
        "fieldType": "DetailedComboBox",
```

```
"fieldName": "comboBox1",
"label": "Dam:",
"topMargin": 2,
"bottomMargin": 8,
"leftMargin": 10,
"rightMargin": 10,
"expandHorizontally": True,
"expandVertically": True,
"headerExpandHorizontally": True,
"values": "southbridge",
"choices": [
      {
            "identifier": "oxbow",
            "displayString": "Oxbow Catchment",
            "detailFields": [
                  {
                        "fieldType": "UnboundedListBuilder",
                        "fieldName": "list1",
                        "label": "Affected Areas:",
                        "values": [
                              "Oxbow Marina",
                              "The Meadows",
                              "Southern Northampton"
                        ],
                        "expandHorizontally": True,
                        "expandVertically": True
                  },
                  {
                        "fieldType": "IntegerSpinner",
                        "fieldName": "spinner1",
                        "label": "Expected Depth (inches):",
                        "values": 4,
                        "minValue": 1,
                        "maxValue": 36,
                        "expandHorizontally": True
                  },
                  {
                        "fieldType": "TimeDelta",
                        "fieldName": "timeDelta1",
                        "label": "Duration:",
                        "unitChoices": [ "hours", "days" ],
                        "valueUnit": "hours",
                        "values": 8,
                        "minValue": 4,
                        "maxValue": 96,
                        "expandHorizontally": True
                  }
            ]
```

```
		},
		{
			"identifier": "southbridge",
			"displayString": "Southbridge",
			"detailFields": "oxbow"
		},
		{
			"identifier": "millRiver",
			"displayString": "Mill River",
			"detailFields": [
				{
					"fieldType": "CheckBox",
					"fieldName": "evacuateSmith",
					"label": "Evacuate Smith College",
					"values": True
				}
			]
		},
		{
			"identifier": "sewagePlant",
			"displayString": "Sewage Treatment Plant",
			"detailFields": [
				{
					"fieldType": "Label",
					"fieldName": "yuck",
					"label": "It's going to be gross.",
					"italic": True
				}
			]
		}
	]
}
```



Immutable: autocomplete, bottomMargin, **choices** (*with detail fields reuse*), columnSpacing, expandHorizontally, expandVertically, **fieldName**, **fieldType**, interdependencyOnly, headerExpandHorizontally, label, leftMargin, rightMargin, scrollable, spacing, topMargin, width

Mutable: editable, enable, extraData, values

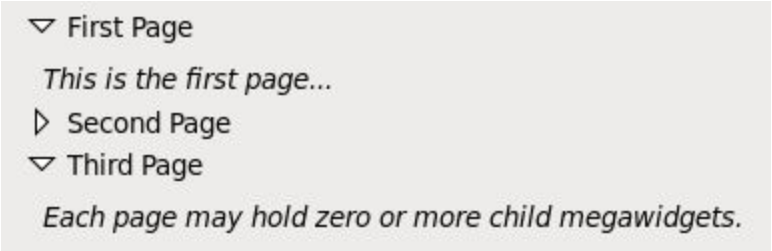
ExpandBar (container)

This is akin to an SWT expand bar, as it holds one or more pages, each accessed via a header label with an arrow next to it. Each page holds child megawidgets. Any label provided is ignored. Clicking a header toggles the visibility of its corresponding page. Unlike a TabbedComposite, multiple pages may be shown at once. A disadvantage, however, is that the area required to show multiple pages could take up a larger amount of vertical space.

Example:

```
{
        "fieldType": "ExpandBar",
        "fieldName": "expandBar1",
        "leftMargin": 10,
        "rightMargin": 10,
        "topMargin": 5,
        "bottomMargin": 5,
        "expandHorizontally": True,
        "expandVertically": True,
        "pages": [
```

```
                {
                        "pageName": "First Page",
                        "pageFields": [
                                {
                                        "fieldType": "Label",
                                        "fieldName": "label1",
                                        "label": "This is the first
                                        page...",
                                        "italic": True
                                }
                        ]
                },
                {
                        "pageName": "Second Page",
                        "pageFields": [
                                {
                                        "fieldType": "Label",
                                        "fieldName": "label2",
                                        "label": "This is the second page.",
                                        "italic": True
                                }
                        ]
                },
                {
                        "pageName": "Third Page",
                        "pageFields": [
                                {
                                        "fieldType": "Label",
                                        "fieldName": "label3",
                                        "label": "Each page may hold zero or
                                        more child megawidgets.",
                                        "italic": True
                                }
                        ]
                }
        ]
}
```



Immutable: bottomMargin, columnSpacing, expandHorizontally,

expandVertically, **fieldName**, **fieldType**, leftMargin, **pages**, rightMargin, spacing, topMargin, width

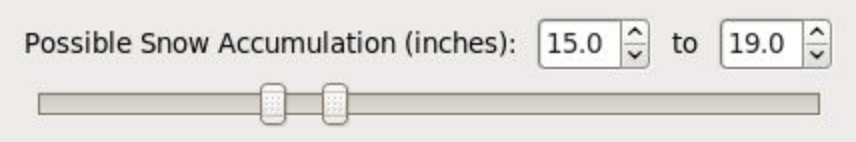
Mutable: enable, expandedPages, extraData

FractionRange (stateful)

This is a pair of spinners allowing the selection of two floating-point values, each within a defined range, that together comprise a range. If provided with a label, the latter appears as a descriptive string to the left of the lower-bound spinner; a between-spinner label may also be provided. It holds the two currently selected fractions as its two states. It may be specified with a scale bar widget below it as an option.

Example:

```
{
	"fieldType": "FractionRange",
	"fieldName": "range1:range2",
	"label": "Possible Snow Accumulation (inches):",
	"betweenLabel": "to",
	"minValue": {
		"range1": 0.0,
		"range2": 0.0
	},
	"maxValue": {
		"range1": 50.0,
		"range2": 50.0
	},
	"values": {
		"range1": 15.0,
		"range2": 19.0
	},
	"precision": 1,
	"expandHorizontally": True,
	"showScale": True
}
```



Immutable: betweenLabel, expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, minimumInterval, precision, sendEveryChange, showScale, spacing, width

Mutable: editable, enable, extraData, incrementDelta, maxValue, minValue, values

FractionSpinner (stateful)

This is a spinner allowing the selection of a single floating-point value within a defined range. If provided with a label, the latter appears as a descriptive string to the left of the spinner. It holds as state the currently selected floating-point value. It may be specified with a scale bar widget below it as an option.

Example:

```
{
        "fieldType": "FractionSpinner",
        "fieldName": "spinner1",
        "label": "Richter Scale:",
        "minValue": 1.0,
        "maxValue": 10.0,
        "values": 5.9,
        "incrementDelta": 1,
        "precision": 1,
        "expandHorizontally": True,
        "showScale": True
}
```

Immutable: expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, precision, sendEveryChange, showScale, spacing, width

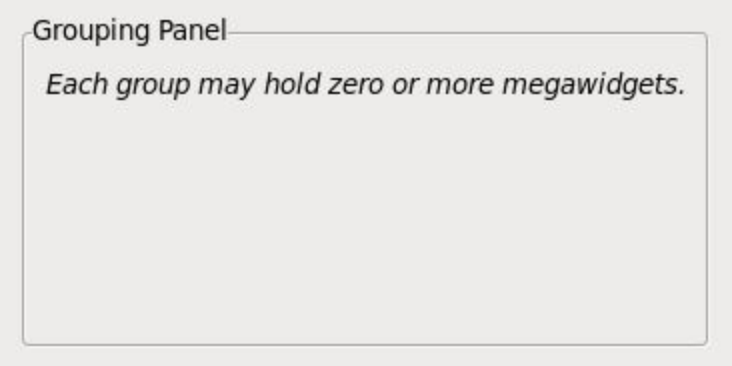
Mutable: editable, enable, extraData, incrementDelta, **maxValue**, **minValue**, values

Group (container)

Like a Composite, this megawidget has one or more columns and child megawidgets, but visually it is identical to an SWT Group in that it has a border around it and its label, if provided, is used as the border's title text. The area holding the children may be given scrollbars for when the child megawidgets are too large to fit within it if desired.

Example:

```
{
      "fieldType": "Group",
      "fieldName": "group1",
      "label": "Grouping Panel",
      "leftMargin": 10,
      "rightMargin": 10,
      "topMargin": 10,
      "bottomMargin": 10,
      "expandHorizontally": True,
      "expandVertically": True,
      "fields": [
            {
                  "fieldType": "Label",
                  "fieldName": "label1",
                  "label": "Each group may hold zero or more
                  megawidgets.",
                  "italic": True
            }
      ]
}
```



Immutable: bottomMargin, columnSpacing, expandHorizontally, expandVertically, **fieldName**, **fields**, **fieldType**, label, leftMargin, numColumns, rightMargin, scrollable, spacing, topMargin, width

Mutable: enable, extraData

HiddenField (stateful)

This megawidget has no visual representation; it is completely invisible within the window. Its purpose is simply to hold any sort of state that should be changed via an interdependency script, as opposed to being manipulated directly by the user. For example, two other

megawidgets could, when experiencing a state change, trigger an interdependency script to change the state of a HiddenField. It may hold as its state any type of Python primitive as a value, as well a list.

Immutable: **fieldName**, **fieldType**

Mutable: extraData, values

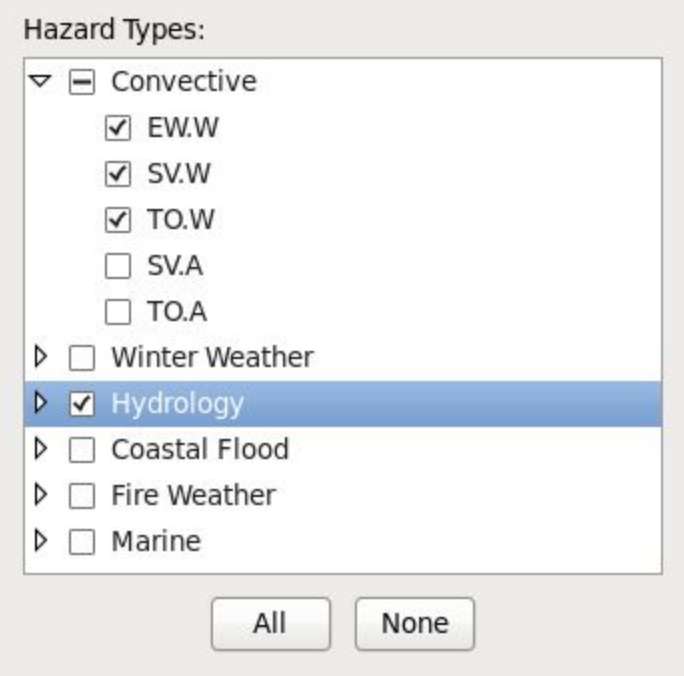
HierarchicalChoicesTree (stateful)

This is comprised of a visual representation of a tree structure, with each node in the tree (whether branch or leaf) being checkable to indicate that it is chosen. If provided with a label, the latter appears as a descriptive string above the tree. This megawidget's state is the tree (hierarchical list) of choices selected by the user, and all "branch" nodes (i.e. choices with child choices) implicitly selected due to the selection of one or more of their descendants. Thus, the state is always a tree structure that is a subset of the tree of choices.

Example:

```
{
      "fieldType": "HierarchicalChoicesTree",
      "fieldName": "tree1",
      "label": "Hazard Types:",
      "lines": 10,
      "choices": [
            {
                  "displayString": "Convective",
                  "children": [ "EW.W", "SV.W", "TO.W", "SV.A",
                  "TO.A" ]
            },
            {
                  "displayString": "Winter Weather",
                  "children":  [ "BZ.W", "BZ.A", "ZR.Y", "IS.W",
                  "LE.W", "LE.Y", "LE.A", "WC.W", "WC.Y", "WC.A",
                  "WS.W", "WS.A", "WW.Y" ]
            },
            {
                  "displayString": "Hydrology",
                  "children": [ "FF.A", "FF.W.Convective",
                  "FF.W.NonConvective", "FA.Y", "FA.A", "FA.W",
                  "FL.A", "FL.W", "FL.Y", "HY.S", "HY.O" ]
            },
            {
                  "displayString": "Coastal Flood",
                  "children": [ "CF.Y", "CF.A", "CF.W", "CR.S",
                  "LS.A" ]
```

```
                },
                {
                        "displayString": "Fire Weather",
                        "children": [ "FW.A", "FW.W" ]
                },
                {
                        "displayString": "Marine",
                        "children": [ "SE.A", "SE.W", "BW.Y", "GL.A",
                        "GL.W", "HF.W", "HF.A", "LO.Y", "MA.S", "MA.W",
                        "MH.Y", "MH.W", "MF.Y", "MS.Y", "SI.Y", "SC.Y",
                        "SW.Y", "RB.Y", "SR.A", "SR.W", "UP.A", "UP.Y",
                        "UP.W" ]
                }
        ],
        "expandHorizontally": True,
        "expandVertically": True
}
```



Immutable: **`fieldName`**, **`fieldType`**, `interdependencyOnly`, `label`, `lines`, `showAllNoneButtons`, `spacing`, `width`

Mutable: **`choices`**, `editable`, `enable`, `extraData`, `values`

IntegerRange (stateful)

This is a pair of spinners allowing the selection of two integers, each within a defined range, that together comprise a range. If provided with a label, the latter appears as a descriptive string to

the left of the lower-bound spinner; a between-spinner label may also be provided. It holds the two currently selected integers as its two states. It may be specified with a scale bar widget below it as an option.

Example:

```
{
	"fieldType": "IntegerRange",
	"fieldName": "range1:range2",
	"label": "Wind Speed (mph):",
	"betweenLabel": "to",
	"minValue":  {
		"range1": 0,
		"range2": 0
	},
	"maxValue": {
		"range1": 50,
		"range2": 50
	},
	"values": {
		"range1": 5,
		"range2": 25
	},
	"expandHorizontally": True,
	"showScale": True
}
```

Immutable: betweenLabel, expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, minimumInterval, sendEveryChange, showScale, spacing, width

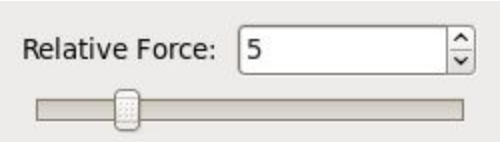
Mutable: editable, enable, extraData, incrementDelta, maxValue, minValue, values

IntegerSpinner (stateful)

This is a spinner allowing the selection of a single integer within a defined range. If provided with a label, the latter appears as a descriptive string to the left of the spinner. It holds as state the currently selected integer. It may be specified with a scale bar widget below it as an option.

Example:

```
{
        "fieldType": "IntegerSpinner",
        "fieldName": "spinner1",
        "label": "Relative Force:",
        "minValue": 1,
        "maxValue": 20,
        "values": 5,
        "expandHorizontally": True,
        "showScale": True
}
```



Immutable: expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, sendEveryChange, showScale, spacing, width

Mutable: editable, enable, extraData, incrementDelta, **maxValue**, **minValue**, values

Label (decorative)

The Label is a simple megawidget with no state, used only to display descriptive text.

Example:

```
{
        "fieldType": "Label",
        "fieldName": "label1",
        "label": "This is a bold and italic label.",
        "bold": True,
        "italic": True
}
```

This is a bold and italic label.

Immutable: bold, color, **fieldName**, **fieldType**, italic, label, preferredWidth, spacing, width, wrap

Mutable: enable, extraData

MenuButton (notifier)

This is a command button that, when selected, displays a dropdown menu of commands. Like the Button megawidget, it has no state, but like stateful megawidgets, its menu commands may be used to trigger the execution of an interdependency script. Such invocations will result in the trigger identifier for the script being of the form:

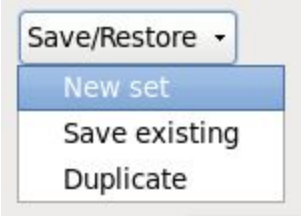
```
<megawidgetIdentifier>.<commandIdentifier>
```

For example, the first command choice in the example provided below would, when invoked, generate the trigger identifier "menuButton1.New set".

At first glance, the MenuButton would appear to be very similar in form to the ComboBox, but they are intended for fundamentally different tasks. The latter has state, and is thus used to track the choice a user has made. It is not intended to be used to issue commands. The MenuButton, in contrast, issues commands through its menu choices. In the example shown here, it would be used to fire off one of three different commands, indicating the user wishes to create something new, save existing information, or duplicate existing information.

Example:

```
{
      "fieldType": "MenuButton",
      "fieldName": "menuButton1",
      "label": "Save/Restore",
      "choices": [ "New set", "Save existing", "Duplicate" ]
}
```



Immutable: **fieldName**, **fieldType**, label, spacing, width

Mutable: **choices**, enable, extraData

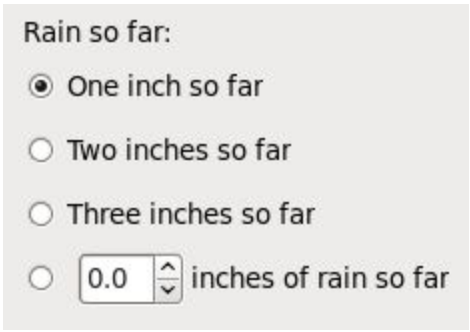
RadioButtons (stateful, detailed)

This is comprised of a list of radio buttons, each representing one of the possible choices

available to the user for its state. If provided with a label, the latter appears as a descriptive string above the radio buttons. It is identical in purpose and state to the ComboBox type, except that it is only appropriate for use in situations where the number of choices is small enough to fit within the vertical space available. Unlike a ComboBox, it allows the inclusion of zero or more child megawidgets providing detail fields for any of the choices represented by its radio buttons.

Example:

```
{
	"fieldType": "RadioButtons",
	"label": "Rain so far:",
	"fieldName": "radioButtons1",
	"choices": [
		"One inch so far",
		"Two inches so far",
		"Three inches so far",
		{
			"identifier": "N inches so far",
			"displayString": "",
			"detailFields": [
				{
					"fieldType": "FractionSpinner",
					"fieldName": "spinner1",
					"minValue": 0,
					"maxValue": 99,
					"incrementDelta": 1,
					"precision": 1
				},
				{
					"fieldType": "Label",
					"fieldName": "label1",
					"label": "inches of rain so far"
				}
			]
		}
	]
}
```



Immutable: **choices**, expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, spacing, width

Mutable: editable, enable, extraData, values

SwtWrapper

This is a special megawidget that is simply a wrapper for SWT widgets. As such, it allows the mixing of arbitrary SWT widgets into a megawidget-constructed graphical user interface. Any label provided is ignored. (*Note: This megawidget is only useful if one is accessing it in Java code to in order to add SWT widgets to it, and is thus never used for GUIs created by users.*)

Immutable: expandHorizontally, expandVertically, **fieldName**, **fieldType**, fullWidthOfDetailPanel, spacing, width

Mutable: extraData

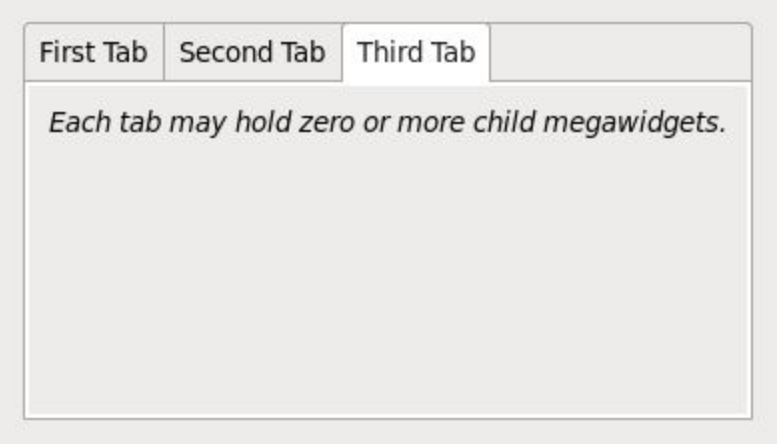
TabbedComposite (container)

This is akin to an SWT tab folder, as it holds one or more pages, each accessed via a tab and in turn holding child megawidgets. It may be configured to display scrollbars in its pages whenever the child megawidgets are too large to fit. Any label provided is ignored. Unlike an ExpandBar, only one page may be shown at a time, but this also limits the size of the area required to display this megawidget and its children, which may be advantageous for some applications.

Example:

```
{
        "fieldType": "TabbedComposite",
        "fieldName": "tabComposite1",
        "leftMargin": 10,
        "rightMargin": 10,
        "topMargin": 10,
        "bottomMargin": 10,
        "expandHorizontally": True,
```

```
        "expandVertically": True,
        "pages": [
              {
                    "pageName": "First Tab",
                    "pageFields": [
                          {
                                "fieldType": "Label",
                                "fieldName": "label1",
                                "label": "This is the first tab's
                                page...",
                                "italic": True
                          }
                    ]
              },
              {
                    "pageName": "Second Tab",
                    "pageFields": [
                          {
                                "fieldType": "Label",
                                "fieldName": "label2",
                                "label": "... and this is the second
                                tab's page.",
                                "italic": True
                          }
                    ]
              },
              {
                    "pageName": "Third Tab",
                    "pageFields": [
                          {
                                "fieldType": "Label",
                                "fieldName": "label3",
                                "label": "Each tab may hold zero or
                                more child megawidgets.",
                                "italic": True
                          }
                    ]
              }
        ]
  }
```



Immutable: bottomMargin, columnSpacing, expandHorizontally, expandVertically, **fieldName**, **fieldType**, leftMargin, **pages**, rightMargin, scrollable, spacing, topMargin, width

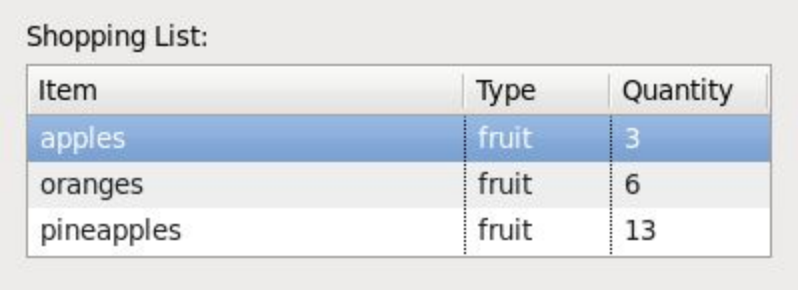
Mutable: enable, extraData, visiblePage

Table (stateful)

This provides a table for displaying tabular data. Its state consists of a list of sublists, with each of the latter constituting a row within the displayed table; the sublist elements are the items shown in the cells of that row. At this time, this megawidget cannot have its state modified via direct user manipulation; as with other stateful megawidgets, it may be changed via an interdependency script, however.

Example:

```
{
      "fieldType": "Table",
      "fieldName": "table1",
      "label": "Shopping List:",
      "lines": 6,
      "columnHeaders": [ "Item", "Type", "Quantity" ],
      "values": [ [ "apples", "fruit", "3" ],
                  [ "oranges, "fruit", "6" ],
                  [ "pinapples", "fruit", "13" ] ]
}
```



Immutable: **`columnHeaders`**, **`fieldName`**, **`fieldType`**, `interdependencyOnly`, `label`, `lines`, `spacing`, `width`

Mutable: `enable`, `extraData`, `values`

Text (stateful)

This is a text entry field. If provided with a label, the latter appears as a descriptive string to the left of the field if single-line, or above if multi-line. Its state consists of the text currently entered into the field.

Example:

```
{
      "fieldType": "Text",
      "fieldName": "text1",
      "label": "Additional Info:",
      "visibleChars": 40,
      "lines": 6,
      "expandHorizontally": True
}
```

Immutable: `expandHorizontally`, **`fieldName`**, **`fieldType`**,

interdependencyOnly, label, lines [1], maxChars, numericOnly, promptText, sendEveryChange, spacing, spellcheck, valueIfEmpty, visibleChars, width

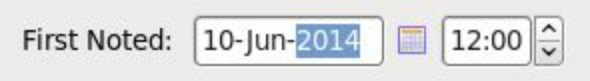
Mutable: editable, enable, extraData, values

Time (stateful)

The Time megawidget allows the selection of a single epoch time in milliseconds, via a calendar/time field pair. It is a simplified version of TimeScale, in that it does not allow the selection of multiple times, has no detail fields, and does not display a scale widget.

Example:

```
{
      "fieldType": "Time",
      "fieldName": "time1",
      "label": "First Noted:",
      "values": 1402401600000
}
```



Immutable: expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, sendEveryChange, spacing, width

Mutable: editable, enable, extraData, values

TimeDelta (stateful)

This is a spinner allowing the selection of a single time delta within a defined range, and optionally a drop-down selector allowing the selection of the time unit to be used. If provided with a label, the latter appears as a descriptive string to the left of the spinner. Its state is the currently selected time delta, expressed as an integer in the unit given by the `valueUnit` parameter.

Example:

```
{
      "fieldType": "TimeDelta",
      "fieldName": "timeDelta1",
      "label": "Time Until Impact:",
      "minValue": 10,
```

```
        "maxValue": 36000,
        "values": 3600,
        "unitChoices": ["seconds", "minutes", "hours"],
        "valueUnit": "seconds",
        "currentUnit": "hours",
        "expandHorizontally": True
}
```

Immutable: expandHorizontally, **fieldName**, **fieldType**, interdependencyOnly, label, sendEveryChange, spacing, **unitChoices**, valueUnit, width

Mutable: currentUnit, editable, enable, extraData, **maxValue**, **minValue**, values

TimeRange (stateful, detailed)

This megawidget consists of two time selectors, with an optional two-thumbed scale and accompanying time "ruler" showing what values on the scale correspond to in time. Each thumb is associated with a different state; thus, the megawidget must have two states. The state identifiers are taken as the colon-separated substrings of its megawidget identifier.

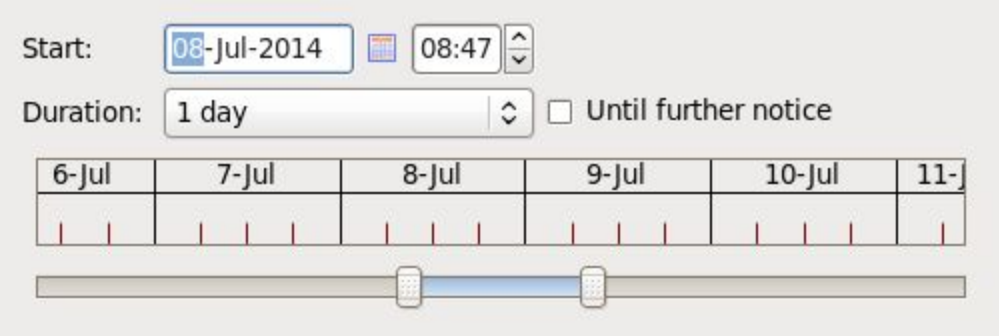
Each thumb is used to select a time value, specified as a long integer describing epoch time in milliseconds. The time selector for the lower boundary is a pair of calendar/time fields, while the one for the upper boundary is a drop-down selector allowing the choosing of a particular offset (time delta) from the lower boundary. The upper boundary must be greater than the lower boundary at all times. If provided with a label, the latter appears as a descriptive string above all of the other component widgets.

Note that if the upper boundary value's distance from the lower boundary is not equal to one of the time deltas provided as choices, the megawidget will allow the two scale thumbs to be moved independently of one another, and the drop-down selector is changed to hold only one value, "N/A". If the delta between the lower and upper boundaries becomes equal to one of the time delta choices, the drop-down selector is changed to hold the list of time delta choices, and the appropriate one is selected. Furthermore, the scale thumbs are locked together, so that moving one moves the other so as to keep the time delta between them the same as the current selection in the drop-down selector.

Note that by default, the two-thumbed scale and its accompanying ruler do not actually get shown; the time selectors are the only ones displayed. The megawidget may be specified with an option to show the scale and ruler.

Example:

```
{
	"fieldName": "start:end",
	"fieldType": "TimeRange",
	"showScale": True,
	"valueLabels": {
		"start": "Start:",
		"end": "Duration:"
	},
	"durationChoices": [
		"12 hours",
		"1 day",
		"2 days",
		"3 days",
		"5 days"
	],
	"detailFields": {
		"end": [
			{
				"fieldType": "CheckBox",
				"fieldName": "endTimeUntilFurtherNotice",
				"label": "Until further notice"
			}
		]
	},
	"values": {
		"start": 1404291600000,
		"end": 1404464400000
	}
}
```



Immutable: detailFields, **fieldName**, **fieldType** interdependencyOnly,, label, sendEveryChange, showScale, spacing, valueLabels, width

Mutable: **durationChoices**, editable, enable, extraData, maximumTime, **maximumVisibleTime**, minimumTime, **minimumVisibleTime**, valueEditables, values

TimeScale (stateful, detailed)

This megawidget has one or more time selectors (each comprised of a pair of fields, one holding a date selector, the other a time of day selector), with an optional scale with one thumb per time selector and accompanying time "ruler" showing what values on the scale correspond to in time. Each time selector and associated thumb is associated with a different state; thus, the megawidget may have one or more states. The state identifiers are taken as the colon-separated substrings of its megawidget identifier.

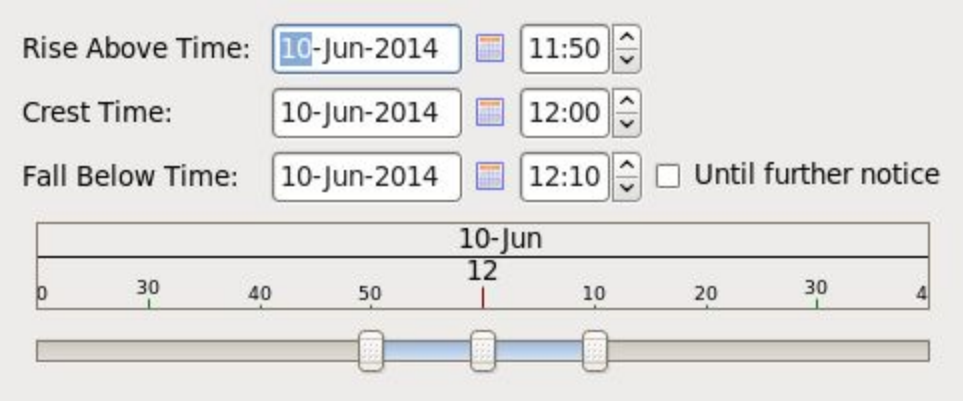
Each time selector and associated thumb is used to select a time value, specified as a long integer describing epoch time in milliseconds. Each thumb must have a greater than or equal value to the previous one at all times. If provided with a label, the latter appears as a descriptive string above all of the other component widgets.

Note that by default, the multi-thumbed scale does not actually get shown; the calendar/time fields are the only ones displayed. The megawidget may be specified with an option to show the scale.

Example:

```
{
      "fieldName": "riseAbove:crest:fallBelow",
      "fieldType": "TimeScale",
      "showScale": True,
      "valueLabels": {
            "riseAbove": "Rise Above Time:",
            "crest": "Crest Time:",
            "fallBelow": "Fall Below Time:"
      },
      "minimumTimeInterval": 60000,
      "detailFields": {
            "fallBelow": [
                  {
                        "fieldType": "CheckBox",
                        "fieldName": "untilFurtherNotice",
                        "label": "Until further notice"
                  }
            ]
      },
      "values": {
```

```
                "riseAbove": 1402401000000,
                "crest": 1402401600000,
                "fallBelow": 1402402200000
        }
}
```



Immutable: detailFields, **fieldName**, **fieldType**, interdependencyOnly, label, minimumTimeInterval, sendEveryChange, showScale, spacing, timeDescriptors, valueLabels, width

Mutable: editable, enable, extraData, maximumTime, **maximumVisibleTime**, minimumTime, **minimumVisibleTime**, valueEditables, values

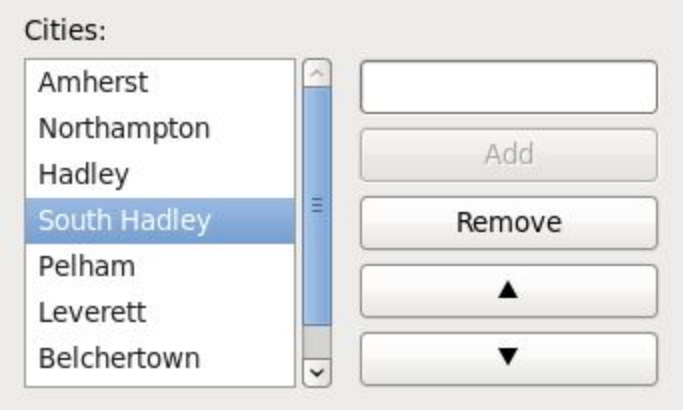
UnboundedListBuilder (stateful)

The unbounded list builder is intended for situations in which the user must be able to build up an ordered list of arbitrary text strings, each unique within the list. It consists of a scrollable list holding the currently selected choices, with buttons allowing the reordering of the list, and the adding of new choices and removal of existing ones. It allows the dragging and dropping of items within the list as well.

Example:

```
{
        "fieldType": "UnboundedListBuilder",
        "fieldName": "unboundedListBuilder1",
        "label": "Cities:",
        "values": [ "Amherst", "Northampton", "Hadley", "South
        Hadley", "Pelham", "Leverett", "Belchertown", "Florence" ],
        "expandHorizontally": True,
        "expandVertically": True
```

```
}
```



Immutable: **`fieldName`**, **`fieldType`**, `interdependencyOnly`, `label`, `lines`, `spacing`, `width`

Mutable: `editable`, `enable`, `extraData`, `values`

Properties

The following list describes all the properties that megawidget definitions may include. The individual megawidget type descriptions indicate which properties are mutable and which are immutable. For those properties that may be optional, the standard default value is given in parentheses following the property name. Note that individual megawidget types may use different default values; in such cases, the defaults will be included in those types' enumerations of their properties.

autocomplete (false)

Boolean indicating whether or not autocomplete should be available for the combo box text field. If true, the user may enter text into the field, and as characters are entered, a list pops up showing any choices that contain the string of characters entered into the field, if any. This should be used with a combo box that contains a large number of choices, as it allows the user to quickly narrow down the choices to a manageable number by typing in a few characters of the desired choice's name.

betweenLabel (empty string)

Text string to be displayed between two visual components of a megawidget; for example, for an IntegerRange this would be the text between the lower and upper value spinners.

bold (false)

Boolean indicating whether or not the label text should be displayed using a bold font.

bottomMargin (0)

Non-negative integer indicating the number of pixels of height to be used for the margin below the panel holding child megawidgets.

choices

List of choices to be allowed. The list must contain at least one choice. Each choice is either a string or a dictionary. If a dictionary, it must contain the following properties:

> ***children (None) [for hierarchical choice megawidgets only]***
>
> Optional list of choices that are children of this choice. Each such list is made up of either strings or dictionaries, just as is the choices property for the megawidget.
>
> ***detailFields (None) [for detailed megawidgets only]***
>
> Optional list of definitions of child megawidgets to be used as detail fields for the choice.
>
> Note that for detailed megawidgets that only display the detail fields associated with at most one choice at any given instant, such as the DetailedComboBox, another option besides a list of child megawidget definitions is the supplying of a choice identifier. The latter must identify a choice from farther up the list of choices.
>
> If such an identifier is supplied, the detail fields associated with the specified identifier are reused for this choice. This allows two or more choices that require the same detail fields to define those fields within the specification of the first such choice, and then reuse them for one or more subsequently specified choices.
>
> Note that CheckBoxes and RadioButtons megawidgets display their detail fields inline.
>
> ***displayString***
>
> Text string describing the choice, to be used to show the choice within the megawidget.
>
> ***identifier (displayString)***
>
> Text string providing the identifier of the choice. The identifier is what is used when determining the state value of the megawidget, if the latter is stateful, or the subcommand that is invoked, if it is a notifier. Identifiers must be unique in the list of choices.

color (None)

Dictionary providing a color to be used in rendering the megawidget. If not given, the default

color is used. The color dictionary should hold the following properties:

red

Floating-point value between 0.0 and 1.0 inclusive, indicating the red component of the color.

green

Floating-point value between 0.0 and 1.0 inclusive, indicating the green component of the color.

blue

Floating-point value between 0.0 and 1.0 inclusive, indicating the blue component of the color.

columnHeaders

List of one or more strings, each one a column header, displayed left to right in the order specified within the list.

columnSpacing (15)

Non-negative integer indicating the number of pixels of spacing to include between each column within the panel holding child megawidgets.

currentUnit (smallest of unitChoices)

Time unit currently being used to specify the state value; must be one of the choices specified by the unitChoices property.

detailFields (None)

Dictionary in which the keys are the state identifiers, and the values are lists of definitions of child megawidgets that are to be used as detail fields for the associated states. Each list of megawidgets is displayed inline.

durationChoices

List of duration choices to be allowed. Each is a string holding a description of a time delta. The description must include one or more of the following substrings. If more than one is included, the substrings must be separated by a space, a comma, or both; furthermore, the substrings have to be ordered by decreasing size of the time unit, i.e. days before hours or minutes, hours before minutes. Note that the unit specifiers may be of any case (lowercase, uppercase, or

mixed), and alternate specifiers for the units are enclosed within parentheses and separated by vertical bars (|):

num (d|day|days)

Number of days, with the days given by *num*.

num (h|hr|hrs|hour|hours)

Number of hours, with the hours given by *num*.

num (m|mn|min|mins|minute|minutes)

Number of minutes, with the minutes given by *num*.

Finally, the time deltas specified by each such string must be unique within the list. Thus, a list with entries for "60 minutes" and "1 hour" is invalid, since they both evaluate to the same time delta.

editable (true)

Boolean indicating whether a stateful megawidget may have its state altered by the user. A non-editable megawidget displays visual cues to indicate its read-only status. Those which are containers also render their nested megawidgets read-only when they are uneditable. (Those that are detailed, however, do not, since it may be desirable to make a megawidget read-only while keeping its detail fields editable.)

enable (true)

Boolean indicating whether or not the megawidget is currently enabled. Disabled megawidgets cannot have their state changed or be invoked by the user, and are grayed out in appearance. Container and detailed megawidgets disable their nested megawidgets when they are disabled.

expandedPages (empty string)

List of zero or more strings, each giving the name of a page that is currently expanded (visible) within the megawidget.

expandHorizontally (false)

Boolean indicating whether or not the megawidget should expand to fill any columns it occupies.

expandVertically (false)

Boolean indicating whether or not the megawidget should expand to fill any rows it occupies.

extraData (None)

Dictionary that, if specified, may hold any arbitrary values associated with any arbitrary non-null keys. This property is not used by the megawidgets themselves; rather, it is intended for use by interdependency scripts as a way to persist values they use internally in between script invocations.

fieldName

Identifier of the megawidget; any string is valid as long as it is unique within the total set of megawidgets and their descendants. Note that stateful megawidgets use their `fieldName` as state identifiers.

fields

List of definitions of child megawidgets.

fieldType

Type of the megawidget being specified by the definition.

fullWidthOfDetailPanel (true)

Boolean indicating whether or not the megawidget should, if it is serving as a detail field for a detailed megawidget, expand to the full width of of the detail panel in which it is embedded.

headerExpandHorizontally (false)

Boolean indicating whether or not the megawidget's header should expand to fill any columns it occupies. The header consists of the components above any grouping panel; for example, for the DetailedComboBox, the header is the label and the combo box at the top of the group.

incrementDelta (minimum that would cause a change in state)

Positive number indicating the increment to be added or subtracted from a value when the megawidget is instructed to change its state as a result of the user paging up or down.

interdependencyOnly (false)

Boolean indicating whether or not the megawidget's state(s) are to be used for the purposes of interdependency scripts only. If true, any state changes the megawidget experiences are not saved as part of the overall state of the entity being edited (be it parameters for a Recommender, or for a Product Staging Dialog, or the metadata of a hazard event).

A value of true is useful for megawidgets that are meant to display state only, and that have the

particulars of such displays be determined by interdependency scripts. An example is the uneditable text fields used to display the rise-above, crest, and fall-below times for FL.A, FL.W, and FL.Y type hazard events within the Hazard Information Dialog.

italic (false)

Boolean indicating whether or not the label text should be displayed using an italic font.

label (empty string)

Text string to be displayed as part of the megawidget; for example, for a Button this would be the text within the button, whereas for a CheckBoxes megawidget this would be the text above the individual checkboxes.

leftMargin (0)

Non-negative integer indicating the number of pixels of width to be used for the margin to the left of the panel holding child megawidgets.

lines (6)

Positive integer indicating the number of lines the megawidget should display in its text field, scrollable list, etc.

maxChars (0)

Non-negative integer indicating the maximum number of characters that the state value can hold. If 0, there is no theoretical limit.

maximumTime (*no practical limit*)

Either a maximum allowable time for all values, given as a long integer representing epoch time in milliseconds, or a dictionary pairing state identifiers with the maximum allowable values they can hold. Note that for the TimeRange megawidget, only the value associated with the first state identifier is governed by this parameter.

maximumVisibleTime

Maximum time, given as a long integer representing epoch time in milliseconds, that is currently visible along the scale bar of the megawidget, that is, the timestamp corresponding to the right edge of the scale bar.

maxValue (*no practical limit*)

Maximum value that the state can take on. If there are multiple states associated with a megawidget, the associated value should be a dictionary pairing state identifiers with the

maximum values they may take on.

minimumInterval (0)

Minimum interval between two state values for a multi-state megawidget; must be a non-negative number of the appropriate type to the megawidget (whether floating-point or integer).

minimumTime (0 (indicating Jan 1, 1970 00:00 GMT))

Either a minimum allowable time for all values, given as a long integer representing epoch time in milliseconds, or a dictionary pairing state identifiers with the minimum allowable values they can hold. Note that for the TimeRange megawidget, only the value associated with the first state identifier is governed by this parameter.

minimumTimeInterval (1 minute)

Non-negative integer indicating the minimum interval of time allowed between adjacent thumbs.

minimumVisibleTime

Minimum time, given as a long integer representing epoch time in milliseconds, that is currently visible along the scale bar of the megawidget, that is, the timestamp corresponding to the left edge of the scale bar.`

minValue (*no practical limit*)

Minimum value that the state can take on. If there are multiple states associated with a megawidget, the associated value should be a dictionary pairing state identifiers with the minimum values they may take on.

numColumns (1)

Positive integer indicating the number of columns that the container megawidget has in which to place child megawidgets.

numericOnly (false)

Boolean indicating whether or not the megawidget should only accept numeric characters (0 through 9) as input.

pages

List of dictionaries defining the pages of the megawidget. Each dictionary should hold the following properties:

numColumns (1)

Positive integer indicating the number of columns the page is to provide its child megawidgets.

pageName

Name of the page, to be displayed as the tab text.

pageFields

List of definitions of child megawidgets.

precision (1)

Positive integer indicating the number of decimals to be shown following the decimal point for a floating-point state value.

preferredWidth (width of text)

Preferrred width of the megawidget, given in units equal to the average size of a character in the font being used. This is intended to be used when the wrap property is set to true.

promptText (empty string)

Text string to be shown as a visual prompt for the user if the field is empty.

rightMargin (0)

Non-negative integer indicating the number of pixels of width to be used for the margin to the right of the panel holding child megawidgets.

scrollable (false)

Boolean indicating whether or not the megawidget is to allow the panel(s) in which its child megawidgets are embedded to scroll. If true, scrollbars are shown whenever the child megawidgets are too large to fit within the available area.

selectedLabel (empty string)

Text string to be used to describe the list of selected choices.

sendEveryChange (true)

Boolean indicating whether or not every state change should result in the interdependency script (if any) being executed. Megawidgets that experience rapid state changes, such as the IntegerSpinner (in which the user may hold down a button to repeatedly increment or decrement

the value) or Text (in which the user may type multiple characters in quick succession) will not execute an interdependency script for every such state change if this is set to false; instead, they will wait for the rapid state change to come to an end before invoking the script.

showAllNoneButtons (true)

Boolean indicating whether or not the megawidget should include two buttons, labeled "All" and "None", that allow the user to select or deselect all of its choices.

showScale (false)

Boolean indicating whether or not the megawidget should show a scale bar (slider with one or more thumbs, as appropriate) below the other components making up its visual state to allow the user another way of viewing and manipulating the current state value(s).

spacing (0)

Non-negative integer indicating the number of pixels separating the megawidget visually from the megawidget above it, or from the top of the enclosing panel if it is the first megawidget in its parent.

spellcheck (false)

Boolean indicating whether or not the megawidget will enable spellchecking on the entered text.

timeDescriptors (None)

Dictionary in which the keys are timestamps in the form of integers giving epoch time (or strings holding the integers, as an alternative, e.g. 1200000 and "1200000" are functionally equivalent keys for this application), and the values are text strings to be used to describe those associated times if they are selected for any of the thumbs.

topMargin (0)

Non-negative integer indicating the number of pixels of height to be used for the margin above the panel holding child megawidgets.

unitChoices

List of time units; must contain one or more of the strings "seconds", "minutes", "hours", and "days". The units so specified are those that are to be offered as choices to the user as to how to specify the state value.

valueEditables (None)

Dictionary holding a key-value pair for each state identifier, which functions as the key, and a

boolean indicating whether or not that state is editable, which is the value.

valueIfEmpty (empty string)

Text string to be used as the state of the megawidget if the latter is empty. This allows a megawidget to be specified as having a particular non-empty value if the user enters no text.

valueLabels (None)

Dictionary holding a key-value pair for each state identifier, which functions as the key, and the text label to be shown, which is the value.

values (default value(s) for specific type of megawidget)

For single-state megawidgets, the current value of the megawidget; for multi-state megawidgets, a dictionary holding a key-value pair for each state identifier, which functions as the key, and the current value for that state, which is the value.

valueUnit ("ms")

Either "ms", "seconds", "minutes", "hours", or "days"; must be the smallest of the units specified by the unitChoices property, or something smaller. This indicates in what units the time delta state is provided.

visibleChars (same as maxChars, unless latter is 0, in which case defaults to 20)

Number of characters that should be displayable on a line without scrolling.

visiblePage (first page)

String giving the name of the page that is currently visible within the megawidget.

width (1)

Positive integer indicating the number of columns that the megawidget should span within its parent. Note that this value may only be greater than 1 if the megawidget is a child of a container megawidget, and the latter has multiple columns.

wrap (false)

Boolean indicating whether or not the label text should, if too long for the existing horizontal space, wrap to the next lines as necessary.

Positioning and Size of Megawidgets

Inline Detail Fields

Some detailed megawidgets lay out their detail fields *inline*. This means that each such field may be either to the right of whatever component with which it is associated, or below it. For a CheckBoxes megawidget, they are associated with a particular choice's checkbox component; for a TimeScale megawidget, they are associated with a particular state identifier's components. How a particular list of detail fields are laid out is determined as follows.

First, the detail field megawidgets are classified as to whether they take up the full width of a detail field panel (the area adjacent to and/or below the associated component, where detail fields are laid out) or not. Any megawidgets that generally stretch across a parent's column are considered to take the full width; examples include large megawidgets such as CheckList, containers such as ExpandBar, and so on. Smaller megawidgets, such as CheckBox and ComboBox, do not. The Composite megawidget can be configured to either take up the entire width of the detail field or to not do so, via its fullWidthOfDetailPanel property.

Next, the detail field megawidgets are inspected in order of specification. Each field that does not take up the full width is placed:

- to the right of the associated component, if it is the first detail field for that component;
- to the right of the previous detail field, if the latter does not take up the full width; or
- in a new row below the previous detail field, if the latter does take up the full width.

In contrast, any field that does take up the full width is placed:

- in a new row beneath the associated component, if it is the first detail field for that component; or
- in a new row beneath the previous detail field.

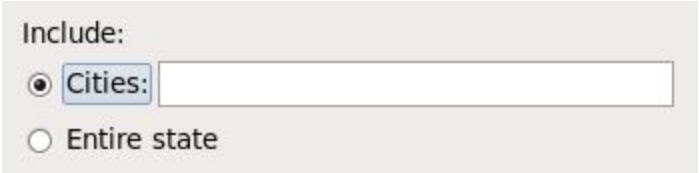
Thus, a list of specified inline detail fields may end up occupying the space to the right of the associated component, in one or more rows below it, or both.

For example, consider the following megawidget definition:

```
[
	{
		"fieldType":"RadioButtons",
		"fieldName": "include",
		"label": "Include:",
		"choices": [
			{
				"identifier": "cities",
				"displayString": "Cities:",
				"detailFields": [
```

```
                        {
                                "fieldType": "Text",
                                "fieldName": "cities",
                                "expandHorizontally": True,
                                "maxChars": 40,
                                "visibleChars": 20
                        }
                ]
        },
        {
                "identifier": "entireState",
                "displayString": "Entire state"
        }
    ]
}
]
```

This yields the following visual layout of the megawidget:



The text field is to the right of the "Cities" choice because single-line Text megawidgets do not take up the full width of a detail panel in which they are placed. If the intent is to position the text field below the "Cities" choice, it may be embedded within a Composite megawidget which has its fullWidthOfDetailPanel property set to True, as follows:

```
[
        {
                "fieldType":"RadioButtons",
                "fieldName": "include",
                "label": "Include:",
                "choices": [
                        {
                                "identifier": "cities",
                                "displayString": "Cities:",
                                "detailFields": [
                                        {
                                                "fieldType": "Composite",
                                                "fieldName": "citiesContainer",
                                                "fullWidthOfDetailPanel": true,
                                                "expandHorizontally": true,
                                                "fields": [
```

```
                                        {
                                              "fieldType": "Text",
                                              "fieldName": "cities",
                                              "expandHorizontally": True,
                                              "maxChars": 40,
                                              "visibleChars": 20
                                        }
                                  ]
                            }
                      ]
                },
                {
                      "identifier": "entireState",
                      "displayString": "Entire state"
                }
          ]
    }
]
```

Because the Composite is configured to take the full width of the detail panel, it is placed on a new row beneath the associated component, and thus any megawidgets the Composite contains are also within this row:

Megawidget Interdependencies

In simpler dialog boxes, megawidgets have no need to affect one another's state or other mutable properties. For example, perhaps a dialog box provides two megawidgets, one a Text megawidget allowing the user to enter a name, the other a ComboBox allowing the choice of one item from a list of possibilities. It is probably not necessary for the two megawidgets to interact; regardless of the text entered in the former, the latter does not change, and vice versa.

However, suppose that the total set of megawidgets for a River Flood Recommender consists of two megawidgets: A RadioButtons megawidget, allowing the choice of the forecast type, and an IntegerSpinner megawidget that is to be disabled unless the forecast type is set to "Set Confidence", in which case it is enabled to allow the selection of the confidence level. The definitions for these megawidgets are as follows:

```
[
      {
            "fieldType": "RadioButtons",
            "fieldName": "forecastType",
            "label": "Type:",
            "choices": [ "Watch", "Warning", "Set Confidence:" ]
      },
      {
            "fieldType": "IntegerSpinner",
            "fieldName": "forecastConfidence",
            "label": "Forecast Confidence (%)",
            "minValue": 0,
            "maxValue": 100,
            "incrementDelta": 10,
            "showScale": True
      }
]
```

In this case, the megawidgets can be said to require interdependencies, that is, when they experience state changes, they must be able to change the mutable properties of one another. In this case, the RadioButtons must be able to change the current state value and enabled/disabled status of the IntegerSpinner when the former has its selection changed. The changing of one megawidget's mutable properties as a result of another's invocation is known as the application of *interdependencies*.

To allow for such relationships, an *interdependency script* may be provided for a set of megawidgets whenever the latter are specified. This script must be within the Python script providing the megawidgets themselves (e.g. a metadata generation script); the latter must include a function named `applyInterdependencies()`. This function must accept two parameters:

- `triggerIdentifiers`: List of one or more megawidgets that underwent a state change, or (for Button megawidgets) were invoked, as a result of user action. If `None` is provided, this means that the script is being run just as the megawidgets are first being displayed.
- `mutableProperties`: Mutable properties of all the megawidgets, given as a dictionary. Each megawidget from the set that has mutable properties will have an entry within the dictionary, keyed by its identifier. The value associated with each key is itself a dictionary, holding key-value pairs for all the mutable properties of that megawidget; again, each key is the name of a mutable property, and each value is the current value of that property.

The function must return a dictionary identical in structure to the `mutableProperties`

parameter, but with a difference: It should only include those mutable properties that were changed as a result of the interdependency script's execution.

The script therefore, in essence, may look at all the mutable properties of all the megawidgets, and may indicate via its return value which properties, if any, should be altered.

Continuing with the example above involving the RadioButtons and IntegerSpinner, the function would, if called as the result of the RadioButtons identifier experiencing a state change, or of initialization occurring, return a dictionary with a single entry. The latter's key would be the IntegerSpinner identifier the corresponding value being another dictionary holding the mutable properties that need changing, in this case either one or two key-value pairs, one indicating whether or not the IntegerSpinner should be enabled, and if it should not, a second indicating what its state value should now be. If called with the IntegerSpinner as the sole trigger identifier, it would return nothing, since changes to the IntegerSpinner's state do not need to precipitate any other changes. The script may be written as follows:

```
def applyInterdependencies(triggerIdentifiers, mutableProperties):

      if triggerIdentifiers == None or \
                  "forecastType" in triggerIdentifiers:

            if mutableProperties["forecastType"]["values"] == "Watch":
                  return {
                        "forecastConfidence": {
                              "enable": False,
                              "values": 50
                        }
                  }

            elif mutableProperties["forecastType"] \
                        ["values"] == "Warning":
                  return {
                        "forecastConfidence": {
                              "enable": False,
                              "values": 80
                        }
                  }

            else:
                  return {
                        "forecastConfidence": {
                              "enable": True
                        }
                  }

      else:
```

```
            return None
```

Persisting Arbitrary Data between Script Executions

Suppose that an interdependency script needs to know something about what previous executions of the same script had done. Since mutable properties are the only data elements that may be set in one execution and then accessed in a later one, any sort of knowledge of previous executions' results must come from mutable properties.

In order to enable scripts to store information for access by later executions of the same script, the mutable property `extraData` is available for all megawidgets. This property allows a script to save data of an arbitrary type and value under a particular name, and then subsequently access that data by querying the value of `extraData` during a later execution of the script.

Thus, for example, the state change of a CheckBox with the `fieldName` "useDefault" could cause the editability and value of another megawidget, a Text with the `fieldName` of "description" , to change. Suppose that when the CheckBox is checked, the Text is to be rendered read-only, and have its value to set to a certain default value. When the CheckBox is unchecked, the Text should be made editable, and have its value restored to what it was before the checking of the CheckBox. This may be accomplished by using `extraData` associated with the Text to store the old value while the CheckBox is checked. The script might be written as follows:

```
def applyInterdependencies(triggerIdentifiers, mutableProperties):

      if triggerIdentifiers == None or \
                  "useDefault" in triggerIdentifiers:

            editable = True
            if "useDefault" in mutableProperties and "values" in \
                        in mutableProperties["useDefault"]:
                  editable = not \
                              mutableProperties["useDefault"]["values"]

            defaultText = "NOTHING TO TO SEE HERE"
            if editable == False and \
                        mutableProperties["description"]["values"] != \
                        defaultText:

                  lastValue = mutableProperties["description"]["values"]
                  return {
                        "description": {
                              "editable": False,
                              "extraData": { "lastValue": lastValue },
                              "values": defaultText
                        }
```

```
                }

            elif editable == True and \
                        mutableProperties["description"]["values"] == \
                        defaultText:
                  if "extraData" in mutableProperties["description"] \
                              and "lastValue" in mutableProperties \
                              ["description"]["extraData"]:
                        value = mutableProperties["description"] \
                                    ["extraData"]["lastValue"]
                  else:
                        value = ""
                  return {
                        "description": {
                              "editable": True,
                              "values": value
                        }
                  }

            else:
                  return {
                        "description": {
                              "editable": editable
                        }
                  }

      else:
            return None
```

Debugging and Flushing Buffers

Debugging an interdependency script is generally done the old-fashioned way: inserting print statements within the script to determine what values arbitrary variables have. Note that due to the way Python scripts interact with the Hazard Services core, any output statements (e.g. calls to `sys.stderr.write()`, `sys.stdout.write()`, etc.) must be followed up by flushing the appropriate buffer (`sys.stderr.flush()` or `sys.stdout.flush()`, respectively, for the two previous write statements) in order to ensure that the output is indeed printed out in the console.

Performance Concerns

An interdependency script should execute very quickly. If it takes more than a very small fraction of a second to run, then it will noticeably impact the performance of Hazard Services and of CAVE. *Thus, potentially long-running algorithms, queries to databases, etc. must be avoided. If such scripts are required for hazard type metadata applications of megawidgets, a metadata reload is the most appropriate approach.*