Analysis Framework Developer's Guide

Analysis Framework Developer's Guide

There are four steps required to add an Analysis to the Analysis Framework. Before these steps, you need to have a working MAEviz development environment. Also, if the new analysis uses any new data schemas, you must define them following the steps of Creating new Dataset schemas. When following these steps, it will be helpful to refer to the example of the standard Bridge Damage analysis in the `ncsa.maeviz.bridges` plugin. For each step, follow the link to the section below for the low-level details.

Step One: Create the #Analysis Description

First, one must create a new Analysis Description file for the Analysis. This will require knowledge of the parameters, outputs, and runtime requirements of the Analysis to be implemented. Generally, this file is placed in a folder called `descriptions` which sits in the root of the defining plugin, and is named to match the analysis name, such as `BridgeDamage.xml`. See the #Analysis Description section for detail of how to write this file, and the syntax used within.

Step Two: Create the #Task

Second, one must implement the Analysis as a Task. Pick the appropriate base class and implement the required methods. Remember that the keys given to the parameters in the Analysis Description must match the `set` methods in this class. Also, the column names given to the outputs must match the values given in the schema for the specific dataset type. The #Task section below gives details of how to extend the base class, and what java methods you must define.

Step Three: Register with the `#ncsa.analysis.newAnalyses` extension point

Third, register this extension. Remember that the id here must match the id given in the Analysis Description and the tag must match the tag in the `#ncsa.tools.ogrescript.ogreTasks` extension point.

Step Four: Register with the `#ncsa.tools.ogrescript.ogreTasks` extension point.

Lastly, register the Task with this extension point. Remember that the tag here must match the tag given in the `ncsa.analysis.newAnalyses` extension point above.

Analysis Description

The Analysis Description file provides detailed information about the various sections of an Analysis. It is defined by using the following tags:

`<analysis-description>`

Attributes

NAME	DEFAULT VALUE	DESCRIPTION
id	(required)	This id MUST match the id given to the Analysis in the `ncsa.analysis.newAnalyses` extension point.
help-context	(optional - no default)	Assigns a help context id to this analysis.

Elements

NAME	REQUIRED	CARDINALITY	DESCRIPTION
<analysis-type>	(required)	1	
<custom-script>	(optional)	0-1	
<groups>	(required)	1	
<parameter>	*(optional)**	*0*-many*	
<output>	(optional)	0-many	

Text

This element has no text.

`<analysis-type>`

Attributes

NAME	DEFAULT VALUE	DESCRIPTION
type	(required)	Defines how this analysis is to be executed, currently supports `simpleIteration`

Elements

NAME	REQUIRED	CARDINALITY	DESCRIPTION
<property>	(optional)	0-many	a `ncsa.tools.common.Property` object. Additional properties required by the type of iterator.

Text

This element has no text.

Example

```
<analysis-type type="simpleIteration">
    <property name="iteratingDatasetKey" value="bridgeDamage" />
</analysis-type>
```

`<custom-script>`

Attributes

This element has no attributes.

Elements

This element has no children.

Text

Defines a location for a custom OgreScript to use instead of auto-generating one. The format for this script will be defined on a separate page. This location is relative to the bundle in which the analysis is shipped.

Example

```
<custom-script>scripts/ogrescript-bridgeFunc.xml</custom-script>
```

`<groups>`

The `<groups>` elements are currently unused by the analysis system. To place parameters in groups, use `group="groupName"` in the parameter element.

`<parameter>`

A `parameter` element with key `<outputKey>.resultName` is required. This is the only way to change a `resultName` for a given output.

```
<parameter key="mappingResult.resultName"  phylum="string" cardinality="single" friendly-name="Result Name" />

...

<output friendly-name="Mapping Result" key="mappingResult" phylum="dataset">
...
```

Attributes

NAME	DEFAULT VALUE	DESCRIPTION
group	(unused)	a string which must match a member of `<groups>` above --**currently unused**

format	*shapefile*	the format of whatever phylum of parameter this is. For datasets, indicates what type of dataset (mapping, shapefile,etc)
phylum	(required)	the type of the parameter, currently supports `string`, `dataset`, or boolean
cardinality	(required)	how many of this type, currently supports `single` or `multiple`
key	(required)	name of property for which value should be added
friendly-name	{required)	name of property for which value should be added
optional	**false**	A value of **`true`** denotes that this parameter need not have a value
advanced	**false**	A value of **`true`** denotes that this is an advanced parameter

Elements

NAME	REQUIRED	CARDINALITY	DESCRIPTION
<types>	(optional)	0-many	A list of types that are accepted by this `<parameter>`.
<description>	(optional)	1	A textual description of the parameter. Mostly used to generate tooltips in the UI.

Text

This element has no text.

Example

```
<parameter group="Required" format="dataset" cardinality="single" key="functionalityTable" friendly-name="
Functionality Table">
    <types>
        <type>bridgeFunctionality</type>
    </types>
</parameter>
```

Syntax for various parameter widget types

A list of the various parameter widget types available, and an example for each, can be found on the Parameter Widget Examples page.

`<output>`

An `<output>` of type `dataset` requires two `<property>` elements.

- `base-dataset-key` - The key of the `<parameter>` which is the base for this new Dataset
- `schema` - The id of the schema that this Dataset implements.

Attributes

NAME	DEFAULT VALUE	DESCRIPTION
format	(required)	the format of the parameter, currently supports `string` or `dataset`
key	(required)	name of property for which value should be added. No spaces allowed.
friendly-name	{required)	name of property for which value should be added

Elements

NAME	REQUIRED	CARDINALITY	DESCRIPTION
<property>	(optional)	0-many	a `ncsa.tools.common.Property` object. Additional properties required by the `<output>`.

Text

This element has no text.

Example

```
<output friendly-name="Bridge Functionality" key="bridgeFunctionality" format="dataset">
    <property name="base-dataset-key" value="bridgeDamage" />
    <property name="schema" value="ncsa.maeviz.schemas.bridgeFunctionalityResults.v1.0" />
</output>
```

Task class

Each Analysis MUST implement a class which extends `ncsa.analysis.maeviz.ogrescript.tasks.core.SimpleFeatureTask` or `ncsa.analysis.maeviz.ogrescript.tasks.core.SimpleFeatureCollectionTask`. If the Task class generate more then one feature, the later class must be used. In the future a choice of base class based on the specific implementation required will be available.

There are two required abstract methods.

```
protected abstract void preProcess() throws ScriptExecutionException;
protected abstract void handleFeature( IProgressMonitor monitor ) throws ScriptExecutionException;
```

Requirements

For each `<parameter>` there must be a corresponding `set` method which corresponds to the `key` attribute in the `<parameter>`.

Example:

```
<parameter group="Required" format="dataset" cardinality="single" key="functionalityTable" friendly-name="
Functionality Table" />

public void setFunctionalityTable( Dataset d );
```

The `handleFeature` method is responsible for two things. First is computing the values that are to be added to the new Feature. Second is to populate the `resultMap`.

Example:

```
resultMap.put( COL_LS_SLIGHT, dmg[0] );
resultMap.put( COL_LS_MODERATE, dmg[1] );
```

Note: if the class extends SimpleFeatureCollectionTask, the member, `resultMapList` which is a LinkedList of `resultMap`, must be used to store the result of each feature in feature colleciton.

As a best practice, add `public final static` constants for each column in the Feature. These column names MUST match the fields as defined in the `gisSchema` for the created dataset.

`ncsa.analysis.newAnalyses` extension point

Each Analysis must register an extension with the `ncsa.analysis.newAnalyses` extension point. This registration allows the Analysis Framework to find all Analyses automatically.

NAME	DEFAULT VALUE	DESCRIPTION
id	(required)	This id MUST match the id given in the `<analysis-description>`
name	(required)	This is the "friendly name" of the Analysis and should be i18n
tag	(required)	The tag MUST match the tag in the `ncsa.tools.ogrescript.ogreTasks` extension point. No spaces allowed.
descriptor	(required)	This points to the descriptor file.

`ncsa.tools.ogrescript.ogreTasks` extension point

Each Analysis must register its implementing class with the `ncsa.tools.ogrescript.ogreTasks` extension point.

NAME	DEFAULT VALUE	DESCRIPTION
id	(required)	This id SHOULD match the fully qualified class name of the task

name	(required)	This is the "friendly name" of the Task and should be i18n
tag	(required)	The tag MUST match the tag in the `ncsa.analysis.newAnalyses` extension point. No spaces allowed.
class	(required)	This points to implementing class.