Plug-ins

In Quaestor you can make use of externally implemented functions via plug-ins.

Calling a function defined in a plug-in is done via the intrinsic functions RUNPLUGIN (returning a value) and RUNPLUGIN$ (returning a string).

Knowledge Engineer Side

Place the plug-in (the .dll file) in the knowledgebase folder under Applic\Plugins\

The KE has to define the return type, the name, the version and the input parameters for the function.

The return type can be a value or string, in accordance to the Quaestor parameter that will be assign to.

Return type example in a Quaestor relation

```
sumTestPlugin = RUNPLUGIN("sum", "", A, B)                          // this will return a value
sumTstString$ = RUNPLUGIN$("sumstr", "", firstString$, secondString$)       // this will return a string
```

Identifying the plug-in

The first 2 input parameters are fixed:

- the first one is a unique name that identifies the function (it can be different from the plug-in .dll file)
- the second one is the version of the function (optional; if not used, it should be an empty string)

The KE is responsible for the management of the plug-in functions that are used in a knowledge base. Each function needs to have an unique name.

Both these parameters are case sensitive.

The number of input parameters for the function is unlimited.

The type of the input parameters can be only value or string.

The KE has to decide and communicate to the plug-in developer the type, order and meaning of each input parameter. There can be any combination of regarding the types of the parameters.

In the previous example, these are:

- 2 value parameters (A and B), for the first relation
- 2 string parameters (firstString$ and secondString$), for the second relation

Plug-in side

Requirements

This functionality is implemented using the Managed Extensibility Framework

The plug-in has to be implemented using .NET framework 4.0.

The communication with Quaestor is done by implementing the interface IQFunctionPlugin and using the attributes defined in the IQPluginMetadata interface. Both of them are defined in the QInterfaces.dll . This library is available on the MARIN nuGet server.

IQFunctionPlugin

```
    /// <summary>
    /// Defines a method to be called by Quaestor.
    /// </summary>
    public interface IQFunctionPlugin
    {
        /// <summary>
        /// Defines a function to be executed by Quaestor.
        /// The KE must inform the developer about the return type and the order and types of the input
values
        /// </summary>
        /// <param name="errorMessage"></param>
        /// <param name="inputValues">Contains the input values; it can contain only string and double.
        /// The order and type are specified by the KE.</param>
        /// <returns>The result of the calculation. It can be only string or double.</returns>
        object Compute(out string errorMessage, List<object> inputValues);
    }
```

Implementation

Steps in implementing a Quaestor plug-in:

1. Create a class library project in Visual Studio
2. Add reference to the QInterfaces.dll (available via the MARIN nuGet server)
3. Add reference to System.ComponentModel.Composition
4. Create a class that will implement the method called by Quaestor
5. Add the Export and ExportMetadata attributes to the class
6. (optional) Create as many classes as needed to be called by Quaestor; create a GUI project to test the classes

Each new function needs to be implemented in a separate class that inherits from the IQFunctionPlugin interface.

Each class containing a function needs to be decorated with the Export and ExportMetadata attributes, as in the following example:

Example of a plug-in implementation

```
    [Export(typeof(MARIN.QInterfaces.IQPlugins.IQFunctionPlugin))]
    [ExportMetadata("Name", "sum")]
    [ExportMetadata("Version", "1")]
    public class FirstMethodExample : IQFunctionPlugin
    {
        #region IQuaestorPlugin Members
        public object Compute(out string errorMessage, List<object> input)
        {
            double first = Convert.ToDouble(input[0]);
            double second = Convert.ToDouble(input[1]);

            errorMessage = null;

            return first + second;
        }
        #endregion
    }
```

The "Name" provided in the ExportMetadata attribute needs to be unique for each function.

The input parameters are organized in a List<object> object. This allows for flexibility in defining them, with the note that the type can be only string or double.

The KE decides the order and the type of each of the list elements.

The first parameter is used for defining an error message that will be displayed to the user, if it is the case.

The return value can be either a string or a double, as requested by the KE. Quaestor will convert from object to string or double at its end.