PUBLIC

Enhanced Method Processing (EMP) in Plant Connectivity 15.5

Implementation Guide

THE BEST RUN SAP

TABLE OF CONTENTS

www.sap.com/contactsap

THE BEST RUN SAP®

DISCLAIMER

Coding Samples

Any software coding or code lines/strings ("code") included in this documentation are only examples and are not intended to be used in a productive system environment. The code is only intended to better explain and visualize the syntax and phrasing rules of certain coding. SAP does not warrant the correctness and completeness of the code given herein, and SAP shall not be liable for errors or damages caused by the usage of the code, except if such damages were caused by SAP intentionally or due to gross negligence.

Internet Hyperlinks

The SAP documentation may contain hyperlinks to the Internet. These hyperlinks are intended to serve as a hint where to find supplementary documentation. SAP does not warrant the availability and correctness of such supplementary documentation or the ability to serve for a particular purpose. SAP shall not be liable for any damages caused by the use of such documentation unless such damages have been caused by SAP's gross negligence or willful misconduct

Accessibility

The information contained in the SAP Library documentation represents SAP's current view of accessibility criteria as of the date of publication; it is in no way intended to be a binding guideline on how to ensure accessibility of software products. SAP specifically disclaims any liability with respect to this document and no contractual obligations or commitments are formed either directly or indirectly by this document.

OVERVIEW

With SAP Plant Connectivity (PCo), SAP provides a software component that enables the exchange of data between an SAP system and the industry-specific standard data sources of different manufacturers, for example, process control systems, plant Historian systems, and programmable logic controller (PLC) systems. With PCo, you can receive tags and events from the connected source systems in production either automatically or upon request and forward them to the connected SAP systems.

The Plant Connectivity component supports the following basic processes:

- **Notification process**: The notification process enables you to monitor production facilities and record any sudden, undesired events (such as rule violations or changes in measurement readings) and report them to a destination system.
- **Query process**: This process enables you to query specific source system tags from a destination system (such as SAP MII). This data can then be displayed on a dashboard, for example.
- **PCo as a server**: PCo can act as an OPC UA and Web server to accept requests from clients and process them within PCo or forward them to destination systems.

Enhanced Method Processing (EMP) enables you to flexibly add methods with your own implementation to any PCo OPC UA or Web server (running within the context of an agent instance) of your choice. Therefore, PCo has established a kind of plugin concept that allows you to extend the server functionality without modification. EMP comes with a design time to flexibly import method metadata. During runtime, your implementation will be called by the surrounding PCo OPC UA or Web server logic.

PREREQUISITES

Technical Prerequisites

The following technical prerequisites are necessary to build a customer-owned EMP implementation for PCo 15.5 (SP00):

- .NET Framework 4.8 or higher
- .NET development environment, for example, Microsoft Visual Studio 2019 or higher.

Required Knowledge and Skills

- .NET development knowledge, preferably C#
- Knowledge of building and creating PCo agents
- Knowledge of connecting OPC UA methods and connecting of Web service clients

GENERAL CONCEPTS

The following diagram illustrates how the EMP is embedded into the architecture of PCo.

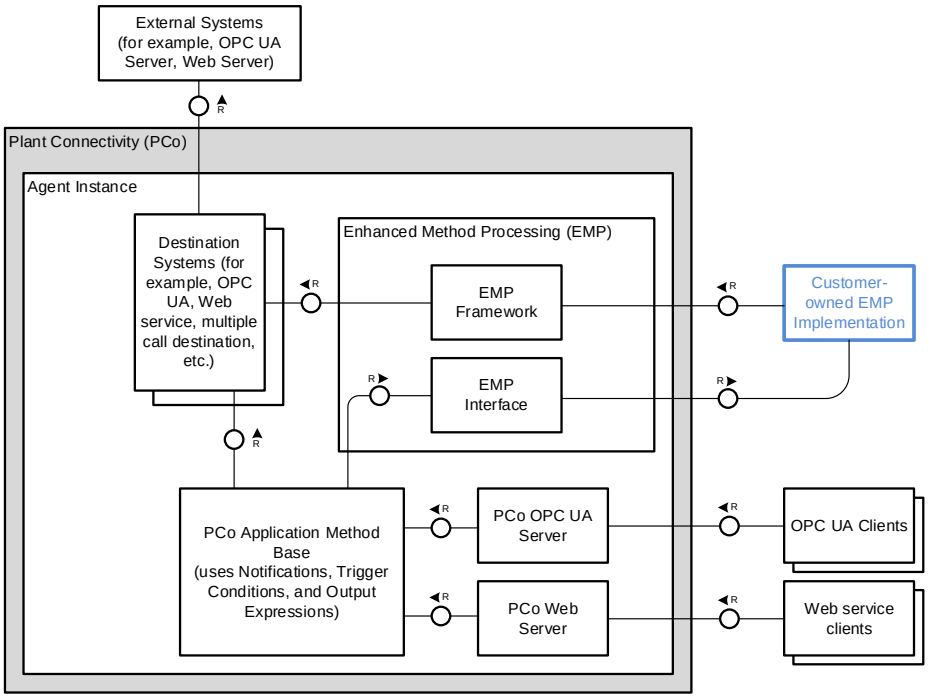

Figure 1: Integration of the EMP implementation into Plant Connectivity 15.5

PCo offers the possibility to configure agent instances acting as OPC UA and Web servers. Users can define input and output parameters of OPC UA methods or Web service operations offered by the PCo agent instance.

Note: Since the procedures for configuring OPC UA methods and Web service operations are almost identical, only the term “method” is used here from now on to describe the concepts of EMP.

The task of a method is defined by assigning it to a destination system. So, for example, if a user wants to define an OPC UA method that retrieves the next operation from SAP Manufacturing Execution (SAP ME), this can be achieved by assigning the OPC UA method to a universal Web service destination through a notification. The OPC UA method would accept a shop floor control (SFC) number as an input parameter and return the information from SAP ME, which is the next operation, to the caller of the OPC UA method as an output parameter. By using trigger conditions and output expressions, the information passed to the OPC UA method can be filtered or converted to adapt to the format required by SAP ME PAPI services. In order to combine several calls to SAP ME or other business systems within the same OPC UA method, a multiple call destination system can be defined that performs the calls in a controlled sequence.

If the standard configuration means of PCo are not sufficient to set up a production scenario, customers can develop their own customer-owned enhanced method processing (EMP) implementation. An EMP implementation is a .NET dynamic link library (DLL) that contains a predefined set of methods and can be deployed to many PCo installations. The EMP methods can be consumed by OPC UA or Web service clients.

Manually configured methods and methods from one or more EMP implementations can be used within the same agent instance in parallel. EMP implementations are independent of the server type, so that they can be used both for the OPC UA and Web server capabilities of a PCo agent instance, even simultaneously.

Consider the following use case: You want to provide an EMP implementation that connects to a server type for which there are no suitable source or destination systems in PCo standard, like an SQL server. You implement the SQL calls within the EMP DLL. Note that, unlike a customer-owned destination system, an EMP implementation cannot make use of the PCo Management Console to configure the processes running inside the DLL. Therefore, unless the connection data to the SQL server is not hard-coded in the EMP implementation, you would need to read the(?) configuration from a text file or query it from elsewhere.

After you deployed the EMP DLL to the PCo system folder, the methods from the DLL are ready to be used in an OPC UA or Web server of an agent instance. Configuration effort is reduced to activating the method and defining trigger conditions, if necessary.

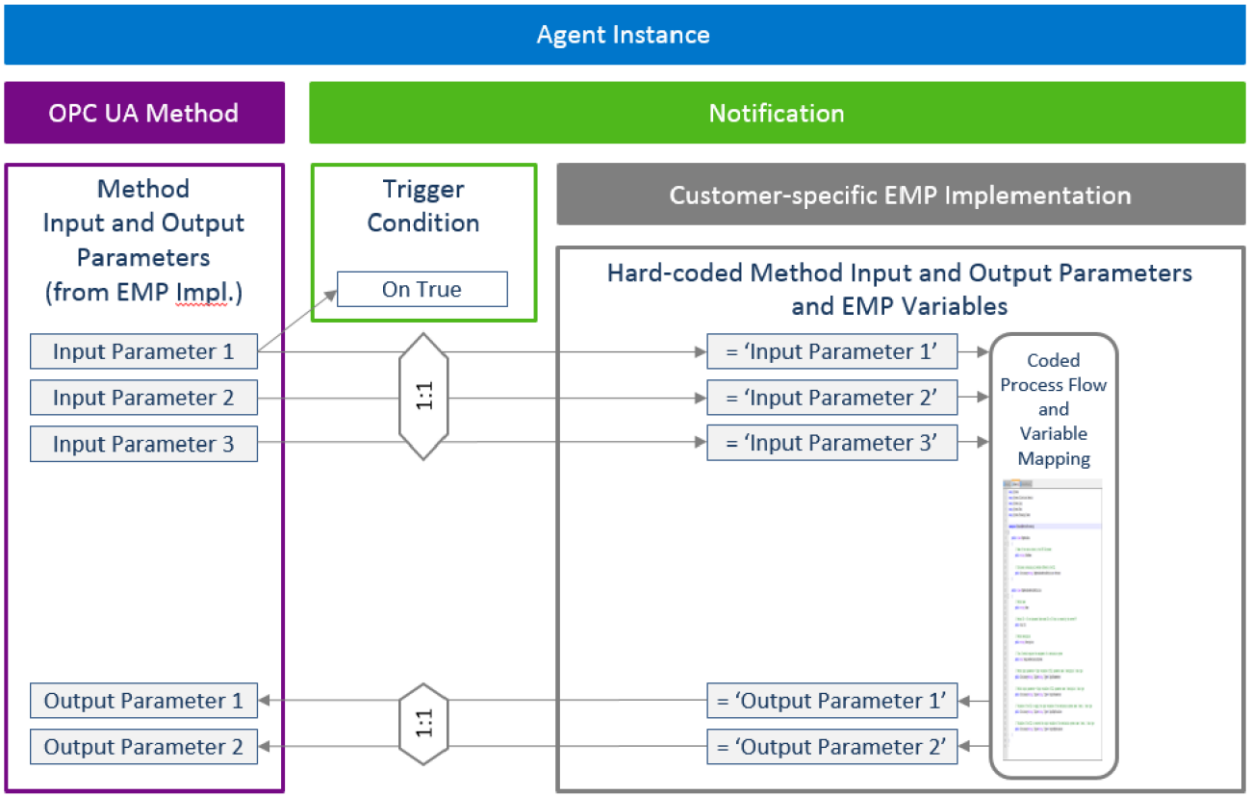

Figure 2: Example of an OPC UA server offering an EMP method without destination system call

EMP methods can also be implemented in a way that they call configured PCo destination systems. Let's assume that an EMP method reads data from an SQL server and forwards it to SAP ME to complete an SFC with the data read from the SQL server. In order to realize this use case, you need to code the call to the SQL server and the call of a PCo destination system within your EMP method implementation. Later, in the PCo Management Console, you configure the universal Web service destination system which performs the actual call to SAP ME.

However, having an EMP method calling a PCo destination system is optional.

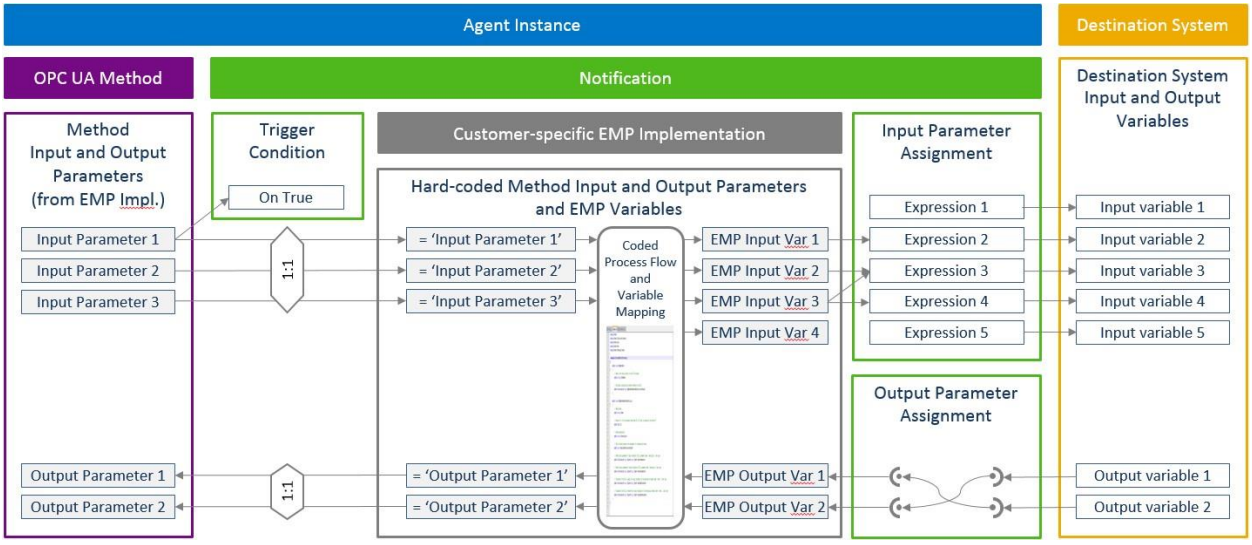

Figure 3: EMP method processing with destination system call

The configuration effort for an EMP method with destination system call involves the selection of an actual PCo destination system and the mapping of destination system input and output variables to the hard-coded EMP input and output variables from the EMP method implementation.

HOW TO IMPLEMENT AN EMP DLL

Example Implementation

In the following steps, you will implement an EMP DLL named `PCoEmpCustomImplementation`. It will provide three methods:

- Method `Add` that adds two numbers and returns the sum.
- Method `Subtract` that subtracts two numbers and returns the difference.
- Method `AMethodWithDestinationCall` that calls a PCo destination system from within the method implementation and returns two values.

You can find the complete coding in the appendix. The coding is the basis for the later descriptions.

Implementation of the EMP DLL in Microsoft Visual Studio

Prerequisites

To create a customer-owned EMP DLL, you require an integrated development environment for .NET development. SAP recommends using Microsoft Visual Studio 2019 or higher.

If you want to use an EMP DLL provided by a third party, you can skip this section and proceed directly to the configuration section.

Implementation Steps

First, create a new project for your EMP DLL implementation. Select *Class Library (.NET Framework)*.

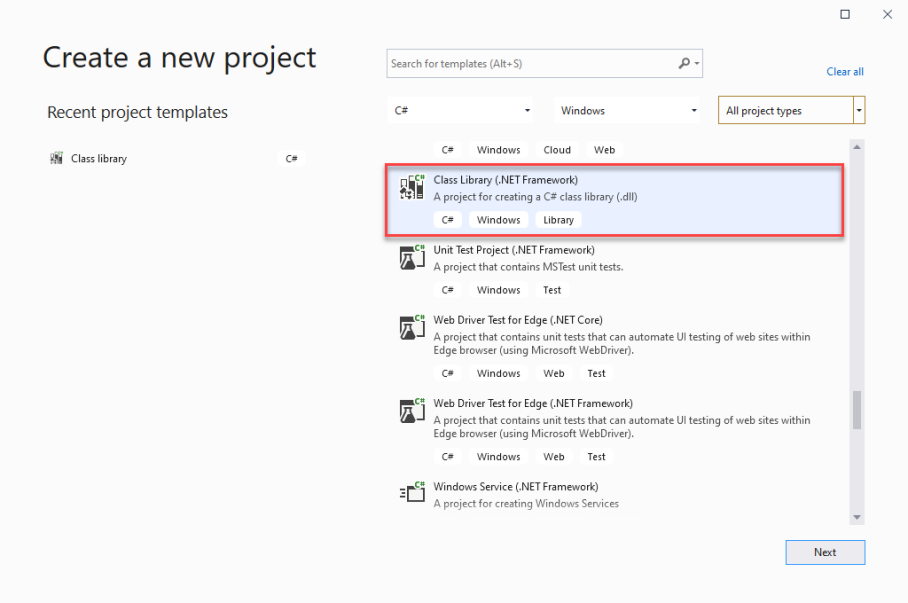

Figure 4: Create a new project

Name the project as `PCoEmpCustomImplementation`. Set the framework to *.NET Framework 4.8.*

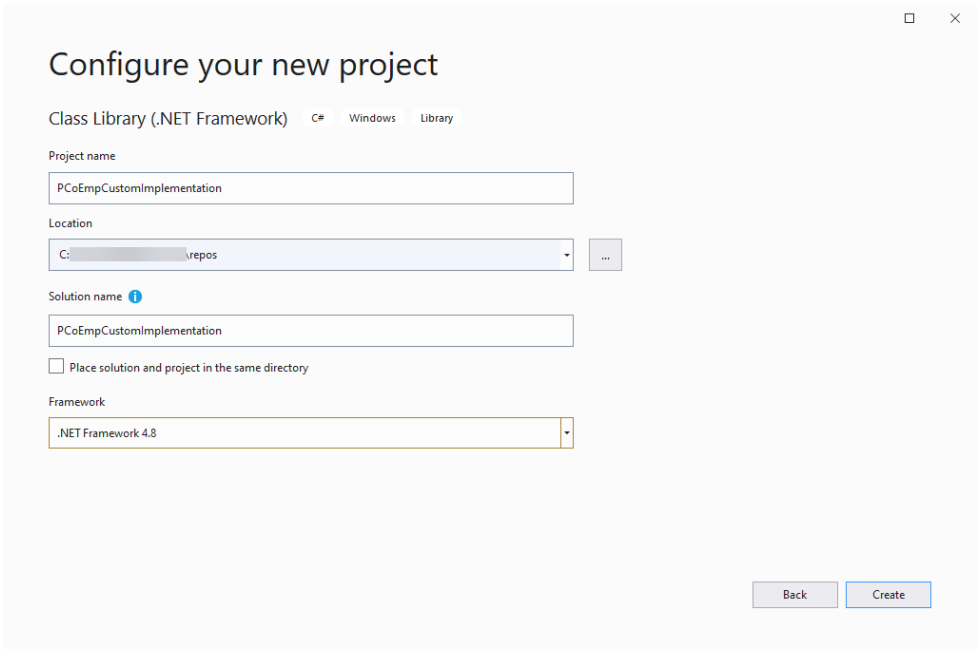

Figure 5:Select project name and framework

Next, add a reference to `MethodProcessingFramework.dll` in the PCo system folder, which is normally `C:\Program Files (x86)\SAP\Plant Connectivity\System`

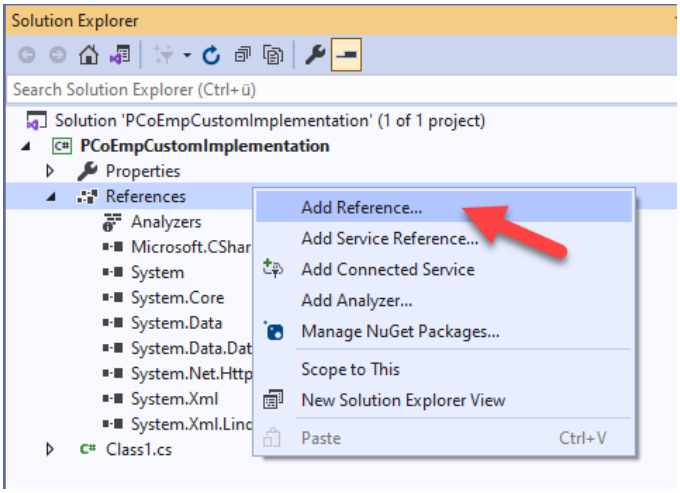

Figure 6: Add a project reference to PCo

As a result, you should find the reference in your project:

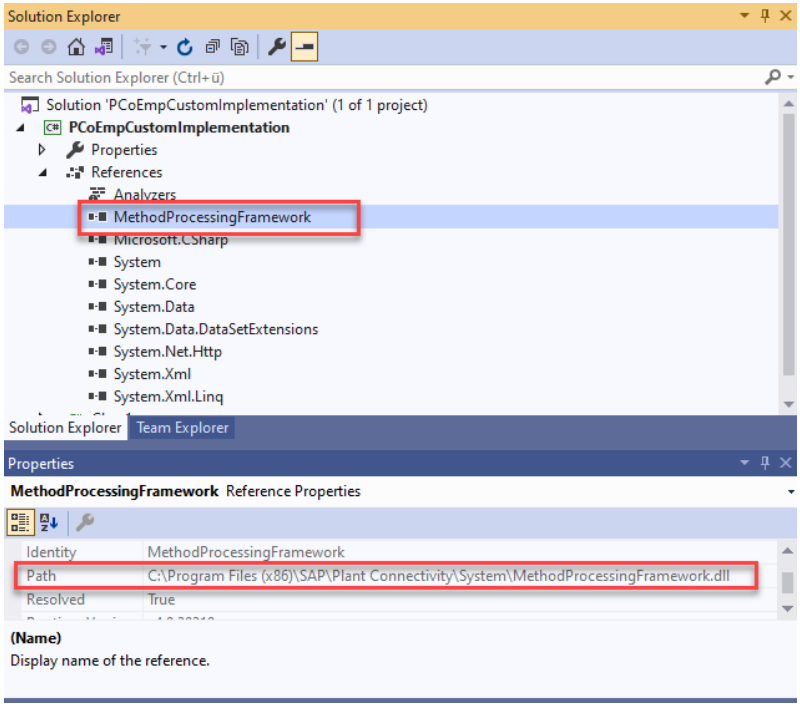

Figure 7: Project references

You now have a solution with an almost empty class, for example, `Class1.cs`. You can rename the class as you wish. Change the class declaration so(?) that it inherits from class `ApplicationMethodsBase`.

```
public class Class1 : ApplicationMethodsBase
```

Visual Studio will now show some errors. To fix them, you need to add a `using` reference to `SAP.Manufacturing.MethodProcessingFramework`. Furthermore, Visual Studio indicates that you need to implement some methods from `ApplicationMethodsBase`. After fixing the errors, your coding will look as follows:

```
using SAP.Manufacturing.MethodProcessingFramework;
using System;
using System.Collections.Generic;

namespace PCoEmpCustomImplementation
{
    public class Class1 : ApplicationMethodsBase
    {
        protected override void executeCustom(
            Guid methodId,
            DestinationCallback destinationCallback,
            Dictionary<string, object> inputVariables,
            out Tuple<Dictionary<string, object>, object> outputVariables)
        {
            throw new NotImplementedException();
        }

        protected override EmpMetaData getMethodDefinitionsCustom()
        {
            throw new NotImplementedException();
        }
    }
}
```

First, you must define a parameterless constructor. From within the constructor, call a private method `GetEmpMetaData()` which does not yet exist. You will fix this error later.

```
        public Class1()
        {
            this.empMetaData = this.GetEmpMetaData();
```

```
        }
```

In the parameterless constructor, you can implement further functions that are called when the PCo agent instance is being started, for example, read configuration data from a text file. This is not required in our example implementation.

Whenever an OPC UA or Web service client calls a method, the PCo Connectivity Framework invokes the method `executeCustom()`. You can hook your own coding into this method to preprocess or postprocess the actual method call. At least, you have to invoke the base method `callEMPMethod()` from within `executeCustom()`, so that the methods implemented in your DLL will be executed.

```
        protected override void executeCustom(
            Guid methodId,
            DestinationCallback destinationCallback,
            Dictionary<string, object> inputVariables,
            out Tuple<Dictionary<string, object>, object> outputVariables)
        {
            base.callEMPMethod(
                methodId,
                destinationCallback,
                inputVariables,
                out outputVariables);
        }
```

Method `getMethodDefinitionsCustom()` returns the EMP DLL and methods metadata to the PCo Connectivity Framework. This is the same metadata that is requested in the constructor. Therefore, call the same method, which does not yet exist, from `getMethodDefinitionsCustom()`:

```
        protected override EmpMetaData getMethodDefinitionsCustom()
        {
            return this.GetEmpMetaData();
        }
```

In the next step, implement the missing private method that returns the EMP metadata:

```
        private EmpMetaData GetEmpMetaData()
        {[…]}
```

The method returns an object instance of type `EmpMetaData`:

```
namespace SAP.Manufacturing.MethodProcessingFramework
{
    public struct EmpMetaData
    {
        public Guid Id;
        public string Description;
        public Dictionary<Guid, MethodMetaData> Methods;
    }
}
```

Every EMP DLL must have a unique ID. Make sure that you do not copy and paste IDs from other EMP implementations. You can use Visual Studio to generate a GUID. The description is displayed in the PCo Management Console when the user selects an EMP DLL.

The EMP methods offered by the DLL are defined in the dictionary `Methods`.

Every EMP method must have a unique ID, which is a GUID. Never reuse GUIDs, otherwise the PCo application method frame cannot identify a method. The structure `MethodMetaData` has the following properties:

```
namespace SAP.Manufacturing.MethodProcessingFramework
{
    public struct MethodMetaData
    {
        public string Name;
        public Guid Id;
```

```
        public string Description;
        public bool RequiresDestinationSystem;
        public bool asynchronous;
        public Dictionary<string, Tuple<string, Type>> InputParameters;
        public Dictionary<string, Tuple<string, Type>> OutputParameters;
        public Dictionary<string, Tuple<string, Type>> InputEmpVariables;
        public Dictionary<string, Tuple<string, Type>> OutputEmpVariables;
    }
}
```

`Name`: Define a method name that is unique within the EMP DLL.

`Id`: Use the same GUID that you used to add the method definition to the dictionary `Dictionary<Guid, MethodMetaData> Methods`.

`Description`: A text that describes the method.

`RequiresDestinationSystem`: If true, the method calls a destination system from within. In this case, you may have to define `InputEmpVariables` and `OutputEmpVariables`.

`asynchronous`: If true, the method is executed as an asynchronous method. This means that the caller receives an empty response immediately, while the method processing happens in a parallel background task.

`InputParameters`: Input parameters of the method.

`OutputParameters`: Output parameters of the method.

`InputEmpVariables`: Only used if `RequiresDestinationSystem` is true.

`OutputEmpVariables`: Only used if `RequiresDestinationSystem` is true.

The dictionaries for input parameters, output parameters, input EMP variables, and output EMP variables are all structured in the same way:

- Key of the dictionary is the name of a parameter or variable.
- Value of a dictionary item is a tuple, where `Item1` is the parameter description, and `Item2` is the data type of the dictionary. Note that only the elementary .NET data types that are supported by PCo can be used.

After you have implemented the metadata definitions in method `GetEmpMetaData()`, you must provide an implementation for every defined method. The signature of a method implementation is

```
public Tuple<Dictionary<string, object>, object> [MethodName](
       DestinationCallback destinationCallback,
       [type of input parameter 1] inputParameter1,
       ..,
       [type of input parameter n] inputParamater n)
```

The method name must match the EMP method name exactly. The number sequence, and data types of arguments for the input parameters must match the number, sequence, and data types of the input parameters.

Therefore, for the method `Add` with its two input parameters `inSummand1` and `inSummand2`, each being of type double, the method signature is

```
public Tuple<Dictionary<string, object>, object> Add(
            DestinationCallback destinationCallback,
            double inSummmand1,
            double inSummand2)
```

If the EMP method requires a destination system (see method `AMethodWithDestinationCall`), you must populate the EMP input variables that will be, by configuration, mapped to the input variable of a destination system in the PCo Management Console. Finally, you must implement the call of the destination system

```
Dictionary<string, object> empOutputVariables =
                destinationCallback.callDestination(empInputVariables, false);
```

and the mapping of the returned EMP output variables to the output parameters of the EMP method.

After you have finished the implementation, you create the EMP DLL that is used later in the PCo Management Console. In Visual Studio, choose a configuration (for example, *Debug* or *Release*), then choose *Build Solution*. If the build was successful, you should find the EMP DLL in the output directory defined in the project properties. The resulting DLL is named <Project Name>.DLL, for example, `PCoEmpCustomImplementation.dll`.

You can now deploy the DLL to one or many PCo installations by copying it into the PCo system folder, which is, in most cases

```
C:\Program Files (x86)\SAP\Plant Connectivity\System
```

CONFIGURATION OF AN EMP DLL IN THE PCO MANAGEMENT CONSOLE

Start the PCo Management Console. Choose an agent instance that you want to use as an OPC UA or Web server. Navigate to the *Servers* tab. Enable *OPC UA Server*, *PCo Web Server*, or both. Configure the server endpoints and server security.

Navigate to the *Server Method Definitions* tab. Select *Add New Method*.

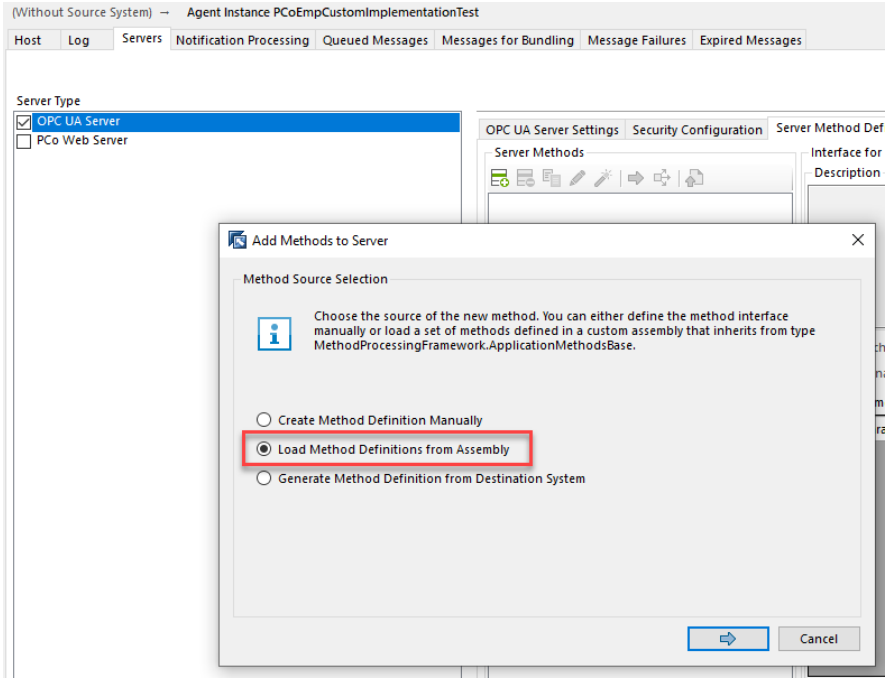

Figure 8: Load method definitions from assembly

Select the name of your EMP DLL from the *Assembly* dropdown list:

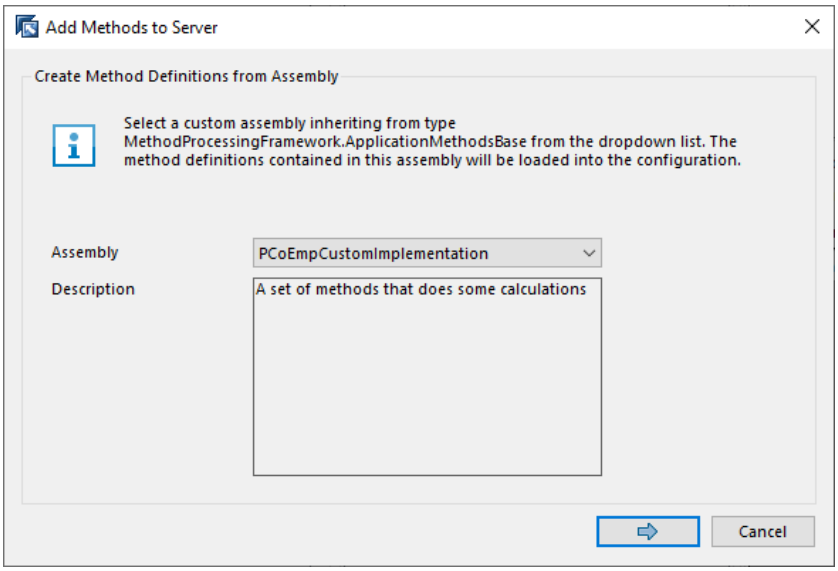

Figure 9: Select EMP DLL

You should now see the assembly description that you defined in the EMP DLL metadata. In the next step, you can choose the methods from the EMP DLL. Select those methods that you want the server to offer.

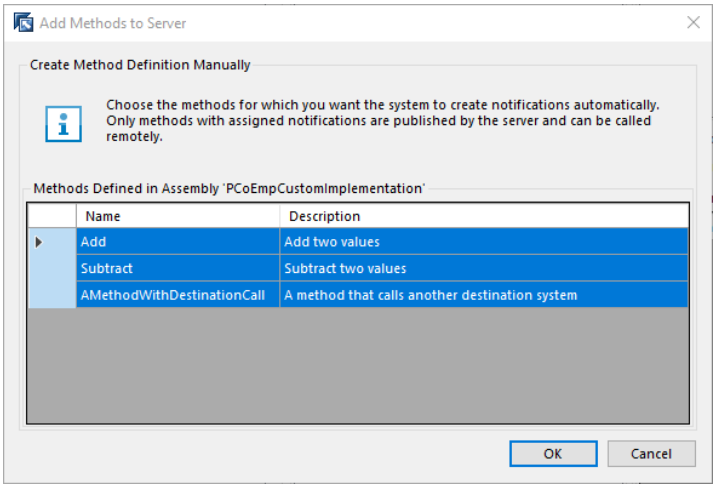

Figure 10: Select EMP methods

The selected methods are now OPC UA methods or Web service operations. By clicking on the server methods, you can see their input and output parameters. You can also check if a method requires a destination system.

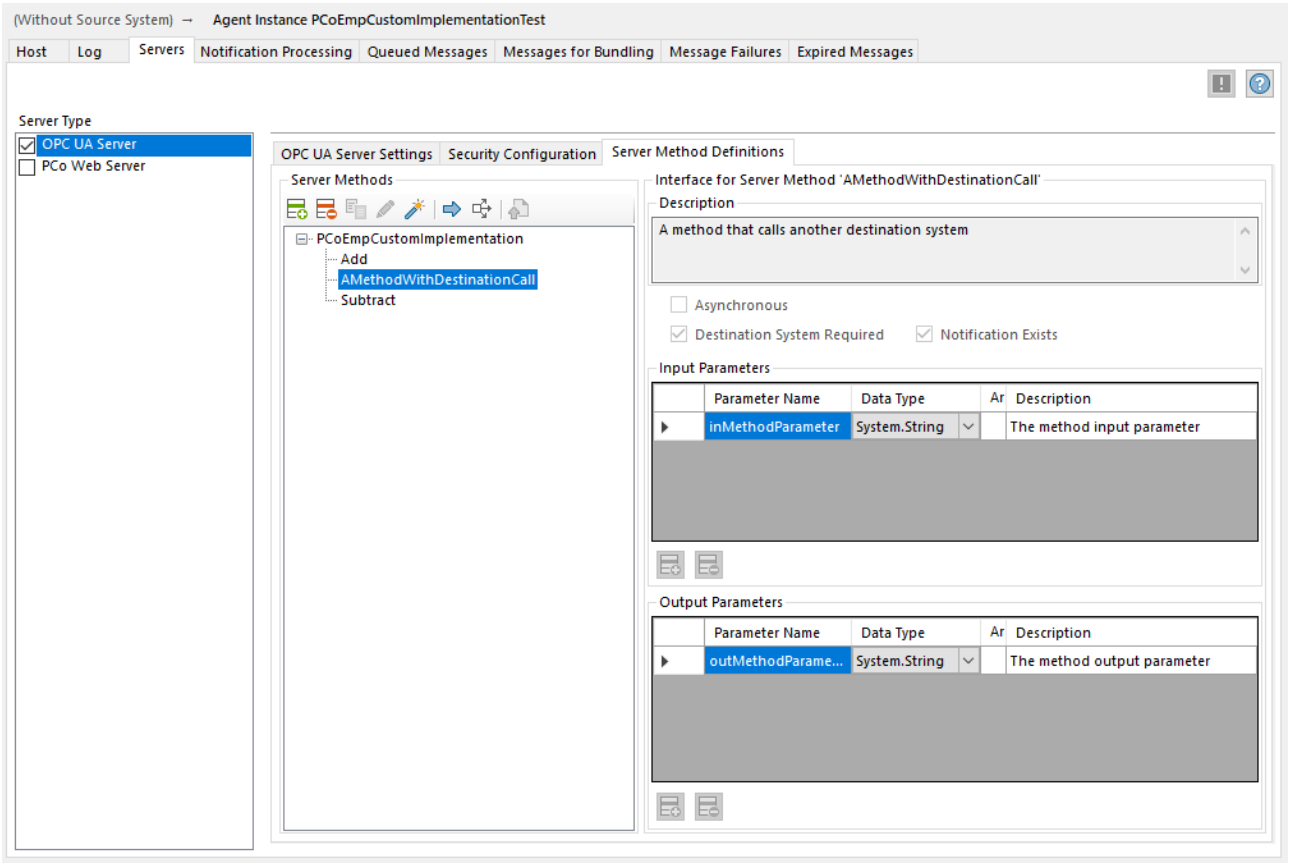

Figure 11:Input and output parameters of an EMP method

The PCo Management Console has created notifications automatically for the selected EMP methods:

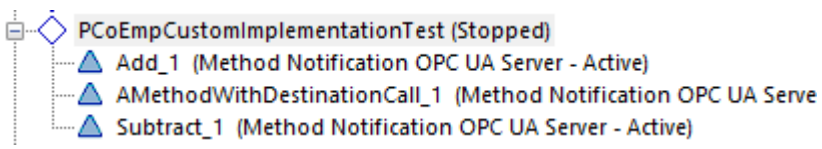

Figure 12: Method notifications

Methods that do not require a destination system such as `Add` and `Subtract` are already functional without further configuration.

For `AMethodWithDestinationCall`, however, a destination system needs to be assigned to the method notification. Navigate to the *Destination* tab of the method notification. Click *Add Destination System*, and select a destination system, for example, a universal Web service destination system that would perform the actual call to SAP ME. The following example shows a multiple call destination system with matching input and output variables.

In the *Destination* tree, expand the root node. Select *Output Destination Mapping*. Map the destination system input variables to the EMP input variables that you hard-coded in the EMP DLL.

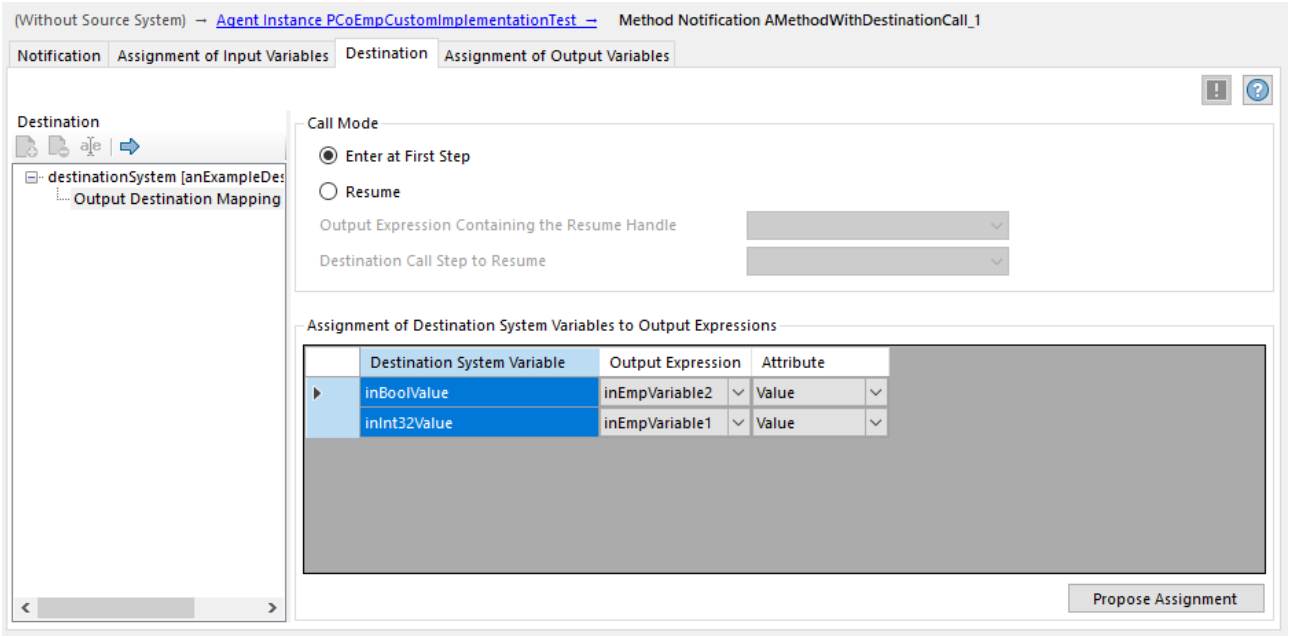

Figure 13: Assignment of EMP input variables to destination system input variables

Next, navigate to the *Assignment of Output Variables* tab. Map the destination system output variables to the EMP output variables from the EMP DLL.

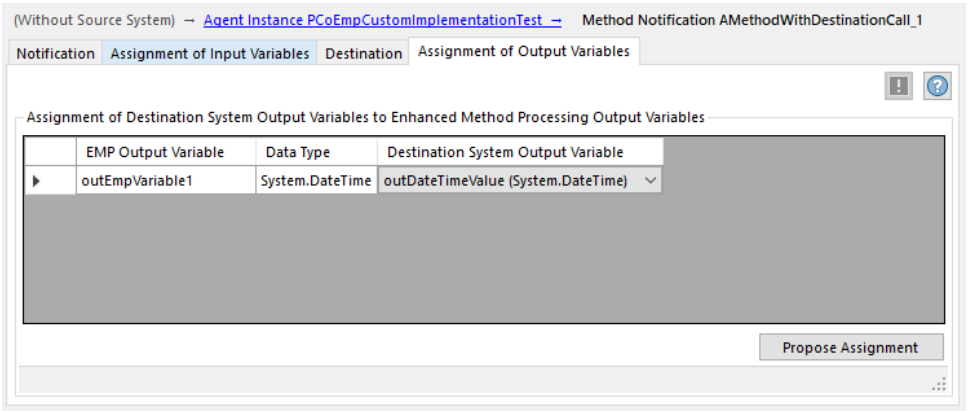

Figure 14: Assignment of EMP output variables to destination system output variables

Your methods from the EMP DLL are now ready to be tested.

TROUBLESHOOTING

If the EMP DLL is not shown in the PCo Management Console, even though you copied it into the PCo system folder, first restart the PCo Management Console. If the problem persists, check for implementation errors in method `getMethodDefinitionsCustom()` of your DLL. Any implementation errors in the metadata declarations would cause the PCo Management Console not to show the DLL.

If you want to debug your DLL, make sure that you copy the `pdb` file of your project along with the `dll` file into the PCo system folder. You can debug the configuration process in the PCo Management Console by attaching the Visual Studio debugger to the process `ManagementConsole.exe`. If you want to trace bugs in the method implementations while the agent instance is running, attach the debugger to the process `PCoSvcHost.exe`.

APPENDIX

```
using SAP.Manufacturing.MethodProcessingFramework;
using System;
using System.Collections.Generic;

namespace PCoEmpCustomImplementation
{
    public class Class1 : ApplicationMethodsBase
    {
        public Class1()
        {
            this.empMetaData = this.GetEmpMetaData();
        }

        protected override void executeCustom(
            Guid methodId,
            DestinationCallback destinationCallback,
            Dictionary<string, object> inputVariables,
            out Tuple<Dictionary<string, object>, object> outputVariables)
        {
            base.callEMPMethod(
                methodId,
                destinationCallback,
                inputVariables,
                out outputVariables);
        }

        protected override EmpMetaData getMethodDefinitionsCustom()
        {
            return this.GetEmpMetaData();
        }

        /// <summary>
        /// Implementation of EMP method "Add"
        /// </summary>
        /// <param name="destinationCallback">Destination callback instance</param>
        /// <param name="inSummmand1">Summand 1</param>
        /// <param name="inSummand2">Summand 2</param>
        /// <returns>Sum</returns>
        public Tuple<Dictionary<string, object>, object> Add(
            DestinationCallback destinationCallback,
            double inSummmand1,
            double inSummand2)
        {
            double sum = inSummmand1 + inSummand2;

            Dictionary<string, object> outputParameters = new Dictionary<string, object>();
            outputParameters.Add("outSum", sum);
            return new Tuple<Dictionary<string, object>, object>(outputParameters, null);
        }

        /// <summary>
        /// Implementation of EMP method "Subtract"
        /// </summary>
        /// <param name="destinationCallback">Destination callback instance</param>
        /// <param name="inMinuend">Minuend</param>
        /// <param name="inSubtrahend">Subtrahend</param>
        /// <returns>Difference</returns>
        public Tuple<Dictionary<string, object>, object> Subtract(
            DestinationCallback destinationCallback,
            double inMinuend,
            double inSubtrahend)
        {
            double difference = inMinuend - inSubtrahend;

            Dictionary<string, object> outputParameters = new Dictionary<string, object>();
            outputParameters.Add("outDifference", difference);
            return new Tuple<Dictionary<string, object>, object>(outputParameters, null);
        }

        /// <summary>
        /// Example of an EMP method that calls a PCo destination system
        /// </summary>
        /// <param name="destinationCallback">Destination callback instance</param>
        /// <param name="inMethodParameter">Method input parameters</param>
        /// <returns>Method output parameters</returns>
        public Tuple<Dictionary<string, object>, object> AMethodWithDestinationCall(
```

```
    DestinationCallback destinationCallback,
    string inMethodParameter)
{
    // Populate the EMP input variables
    Dictionary<string, object> empInputVariables =
        new Dictionary<string, object>();

    if (string.IsNullOrEmpty(inMethodParameter))
    {
        empInputVariables.Add("inEmpVariable1", 0);
        empInputVariables.Add("inEmpVariable2", false);
    }
    else
    {
        empInputVariables.Add("inEmpVariable1", 42);
        empInputVariables.Add("inEmpVariable2", true);
    }

    // Call the destination system
    Dictionary<string, object> empOutputVariables =
        destinationCallback.callDestination(empInputVariables, false);

    DateTime empOutputVariableTimeStamp =
        (DateTime)empOutputVariables["outEmpVariable1"];

    // Populate the method output parameters
    Dictionary<string, object> outputParameters = new Dictionary<string, object>();

    outputParameters.Add(
        "outMethodParameter",
        empOutputVariableTimeStamp.ToLocalTime().ToString());

    return new Tuple<Dictionary<string, object>, object>(outputParameters, null);
}

private EmpMetaData GetEmpMetaData()
{
    // Dll information
    EmpMetaData empMetaData = new EmpMetaData();
    empMetaData.Id = new Guid("3E7EC6A6-D48F-4CD4-8B84-D327FCF0BE62");
    empMetaData.Description = "A set of methods that does some calculations";
    empMetaData.Methods = new Dictionary<Guid, MethodMetaData>();

    // ------------------------------------
    //Method "Add"
    MethodMetaData addMetaData = new MethodMetaData();
    addMetaData.Id = new Guid("B3637E9B-D52E-41E2-B840-A5D4C3657485");
    addMetaData.Name = "Add";
    addMetaData.Description = "Add two values";
    addMetaData.RequiresDestinationSystem = false;
    addMetaData.InputParameters = new Dictionary<string, Tuple<string, Type>>();
    addMetaData.OutputParameters = new Dictionary<string, Tuple<string, Type>>();

    // Input parameters
    addMetaData.InputParameters.Add(
        "inSummand1",
        new Tuple<string, Type>(
            "Summand 1",
            typeof(double)));
    addMetaData.InputParameters.Add(
        "inSummand2",
        new Tuple<string, Type>(
            "Summand 2",
            typeof(double)));

    // Output parameters
    addMetaData.OutputParameters.Add(
        "outSum",
        new Tuple<string, Type>(
            "The sum of both values",
            typeof(double)));

    empMetaData.Methods.Add(addMetaData.Id, addMetaData);

    // ------------------------------------
    //Method "Subtract"
    MethodMetaData subtractMetaData = new MethodMetaData();
    subtractMetaData.Id = new Guid("912894F5-2653-49D6-8EFE-76750B75A772");
```

```
subtractMetaData.Name = "Subtract";
subtractMetaData.Description = "Subtract two values";
subtractMetaData.RequiresDestinationSystem = false;
subtractMetaData.InputParameters = new Dictionary<string, Tuple<string, Type>>();
subtractMetaData.OutputParameters = new Dictionary<string, Tuple<string, Type>>();

subtractMetaData.InputParameters.Add(
    "inMinuend",
    new Tuple<string, Type>(
        "Minuend",
        typeof(double)));
subtractMetaData.InputParameters.Add(
    "inSubtrahend",
    new Tuple<string, Type>(
        "Subtrahend",
        typeof(double)));

// Output parameters
subtractMetaData.OutputParameters.Add(
    "outDifference", new
    Tuple<string, Type>(
        "The difference between the two values",
        typeof(double)));

empMetaData.Methods.Add(subtractMetaData.Id, subtractMetaData);

// -----------------------------------
//Method "AMethodWithDestinationCall"
MethodMetaData destinationCallMetaData = new MethodMetaData();
destinationCallMetaData.Id = new Guid("5543531A-2AB9-4875-BE2E-7BBB5837EE76");
destinationCallMetaData.Name = "AMethodWithDestinationCall";
destinationCallMetaData.Description =
    "A method that calls another destination system";
destinationCallMetaData.RequiresDestinationSystem = true;
destinationCallMetaData.asynchronous = false;
destinationCallMetaData.InputParameters =
    new Dictionary<string, Tuple<string, Type>>();
destinationCallMetaData.OutputParameters =
    new Dictionary<string, Tuple<string, Type>>();
destinationCallMetaData.InputEmpVariables =
    new Dictionary<string, Tuple<string, Type>>();
destinationCallMetaData.OutputEmpVariables =
    new Dictionary<string, Tuple<string, Type>>();

// Input parameters
destinationCallMetaData.InputParameters.Add(
    "inMethodParameter",
    new Tuple<string, Type>(
        "The method input parameter",
        typeof(string)));

// Output parameters
destinationCallMetaData.OutputParameters.Add(
    "outMethodParameter",
    new Tuple<string, Type>(
        "The method output parameter",
        typeof(string)));

// EMP input variables (input variables of destination system call)
destinationCallMetaData.InputEmpVariables.Add(
    "inEmpVariable1",
    new Tuple<string, Type>(
        "The EMP input variable 1",
        typeof(int)));
destinationCallMetaData.InputEmpVariables.Add(
    "inEmpVariable2",
    new Tuple<string, Type>(
        "The destination system input variable 2",
        typeof(bool)));

// EMP output variables (output variable of destination system call)
destinationCallMetaData.OutputEmpVariables.Add(
    "outEmpVariable1",
    new Tuple<string, Type>(
        "The EMP output variable",
        typeof(DateTime)));

empMetaData.Methods.Add(destinationCallMetaData.Id, destinationCallMetaData);
```

```
            return empMetaData;
        }
    }
}
```