**Implementation Guide**
SAP Manufacturing
Document Version: 1.0 – 2016-11-04

**CUSTOMER**

# Enhanced Method Processing (EMP) in Plant Connectivity 15.1

## Implementation Guide

# Typographic Conventions

| Type Style | Description |
| --- | --- |
| *Example* | Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options.<br>Textual cross-references to other documents. |
| **Example** | Emphasized words or expressions. |
| `EXAMPLE` | Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE. |
| `Example` | Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools. |
| **`Example`** | Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation. |
| **`<Example>`** | Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system. |
| `EXAMPLE` | Keys on the keyboard, for example, `F2` or `ENTER`. |

# Document History

| Version | Date | Change |
| --- | --- | --- |
| 1.0 | 2015-11-04 | Updates for PCo 15.1 (SP03) |
| | | |
| | | |
| | | |

# Table of Contents

# 1 Disclaimer

Document classification for SAP Library: Customer

## 1.1 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.

## 1.2 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.

## 1.3 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.

# 2 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.

Enhanced Method Processing (EMP) enables you to flexibly add methods with own implementation to any PCo OPC UA 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 server logic.

# 3 Prerequisites

## 3.1 Technical Prerequisites

The following technical prerequisites are necessary in order to build a customer-owned enhancement for PCo 15.2:

- .NET Framework 4.5.2 or higher
- .NET development environment, for example, Microsoft Visual Studio 2015 Professional or higher. Microsoft Visual Studio Express editions are not supported.

## 3.2 Required Knowledge and Skills

- .NET development knowledge, preferably C#
- Knowledge in building and creating PCo agents
- Knowledge in connecting OPC UA methods

# 4 Architectural Overview

## 4.1 Main Building Blocks

The following diagram shows the main building blocks of Plant Connectivity for a notification process and illustrates how the EMP is embedded into the architecture.

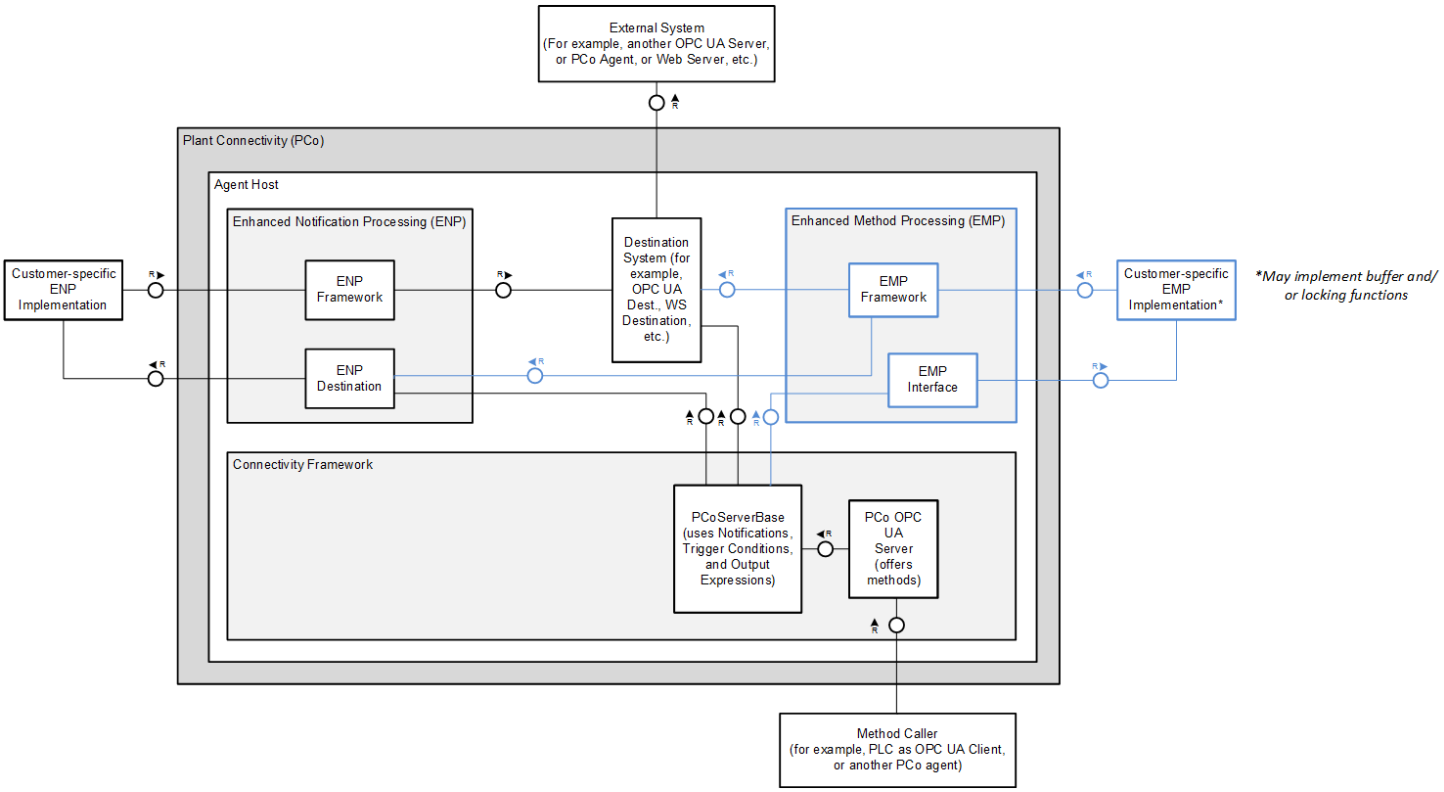



Figure 1: Integration of the EMP Enhancement Implementation into Plant Connectivity 15.2

An external method caller calls an OPC UA method on a running OPC UA server (configured within a PCo agent). This request is routed through the PCo Server Base class an reaches the Enhanced Method Processing Framework. The EMP itself is able to communicate with the customer specific implementation because a potential implementer has implemented the EMP interface provided by the PCo standard.

The PCo management consoles allows the definition of OPC UA server methods on an OPC UA server which runs in the context of a PCo agent. In case of EMP you have the opportunity to import method definitions from your dll.

The PCo management console provides screens for the configuration of the method parameters which may be necessary for a successful method call with own implementation.

The EMP framework is also linked to the destination system. This link enables you to call any PCo destination from your implementation. The implementer decides when the destination call should happen but the actual call itself is handled by the connectivity framework. Destination calls can by synchronous and asynchronous depending on the PCo configuration of the EMP methods.

To call a destination from an own implementation it can be necessary to deal with internal variables within a custom implementation. Those variables are called "EMP variables".

The PCo management console provides functionality for the mapping of EMP variables to destination variables which are needed for a successful web service call.

## 4.2 Interfaces

In the architectural overview, the ENP was shown in the overall context of an EMP scenario. Below is a drill down into the EMP. The diagram below shows a class diagram of the `MethodProcessingFramework` project and its relationship to the enhancement implementation.

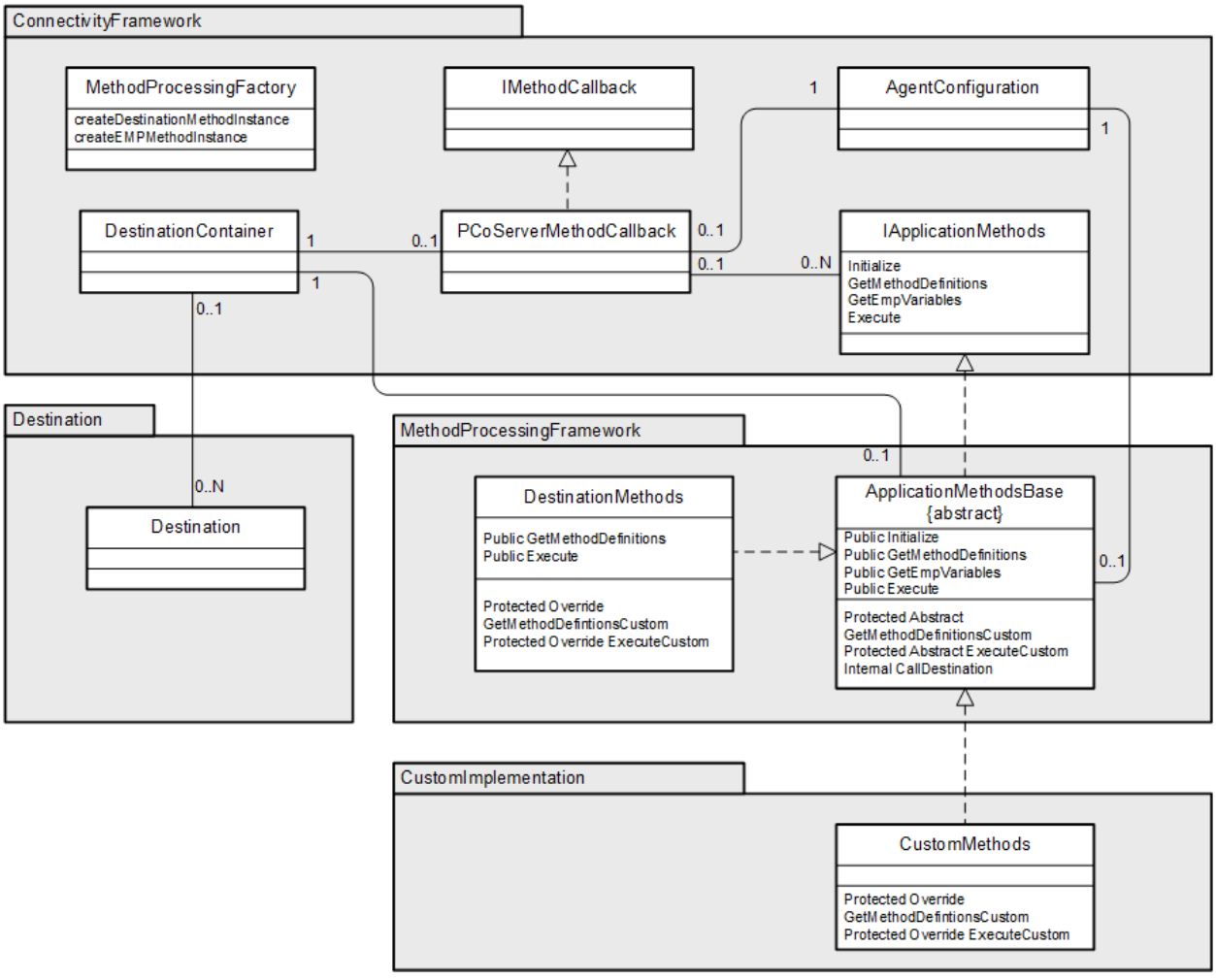



Figure 2: Class Diagram of the MethoProcessingFramework Project

The main part of the EMP is handled by the `MethodProcessingFramework`. All custom implementations derive from the class `ApplicationMethodBase`. Beside an initialization method that the `PCoServerMethodCallback` needs to deal with EMP method and destination instances at runtime the class provides the method

`GetMethodDefinitions.` This method is used from the PCo management console for the import of method metadata from a foreign dll. Internally it calls the method `GetMethodDefinitionsCustom`. The method `GetEMPVariables` is used to transfer variable information from the custom dll to PCo. With this information the connectivity framework is able to call the destination system.

At runtime the method `Execute` is used to execute the method call. This means that either a destination system or an EMP method is called. For the execution of an EMP method the `Execute` method calls `ExecuteCustom` internally.

The internally called methods are all implemented within the EMP dll (within the foreign code).

### 4.2.1 IApplicationMethods Interface (Controller Interface)

The `IApplicationMethods` interface contains the definition of how an enhancement implementation has to be implemented. The interface provides an initialization method, two design time methods for the configuration of an enhancement implementation, and one runtime method for processing notification messages.

The methods `GetMethodDefinitions, GetEMPVariables, Initialize and Execute` are explained within the previous chapter, already.

#### 4.2.1.1 Initialization

The `Initialize` method initializes the implementing class with references to a `DestinationContainer` and an `AgentConfiguration.`

The `DestinationContainer` is used to bundle the instances of destinations for a better destination handling.

## 4.2.1.2 Configuration Methods

In order to understand the sense of the configuration methods you have to be familiar with the PCo terms and functions when dealing with EMP.

Let's start with an explanation of EMP method processing that doesn't call an additional destination. You use this variant each time when you want to plug in your own logic (and want the connectivity framework to execute it) but you don't want to perform a destination call.

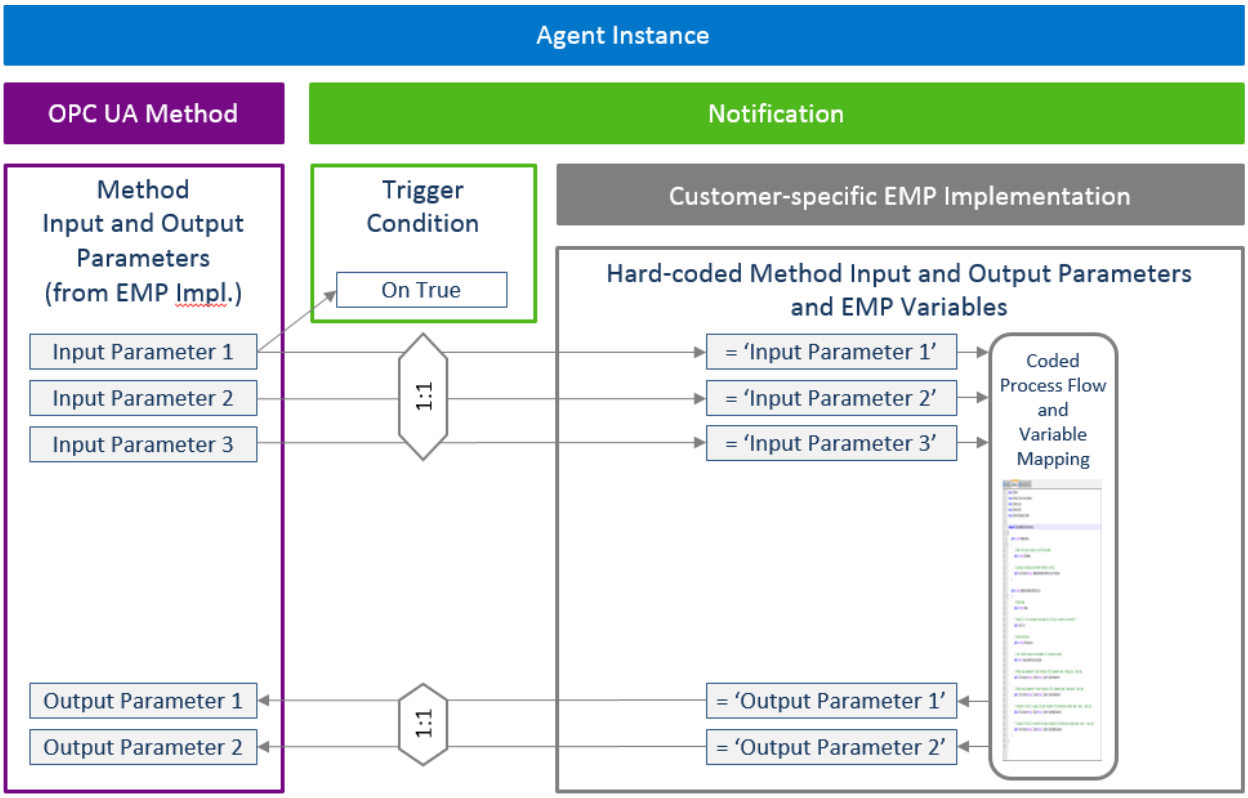



Figure 3 Enhanced Method Processing without destination system call

After methods have been imported from an implementing dll they are available as OPC UA methods on a UA server of an agent instance. Like for any other UA method it is possible to define a method based notification. Like you know it from notifications you can enter a trigger condition if you want to. The input parameters of the UA method are mapped 1:1 to the notification input parameters. From there they are mapped 1:1 to the EMP implementation and are used there within the coding. The output parameter of the EMP implementation have also a 1:1 relationship to the notification output parameters and from there (also 1:1) to the OPC UA method output parameters on the UA server.

Beside the EMP method handling without destination processing it is also possible to call a destination from the EMP implementation. Such a behavior may be useful if you consider the following example:

During the runtime of a PCo process EMP method calls take place in order to invoke the EMP logic. One of the implemented methods has the need to call another system via web service (for example to set an operation to "complete" in ME). For this reason a destination call is implemented inside the EMP coding. In this case the connectivity framework will perform a web service destination call.

The PCo management console fetches method metadata via method `GetMethodDefinitions`.

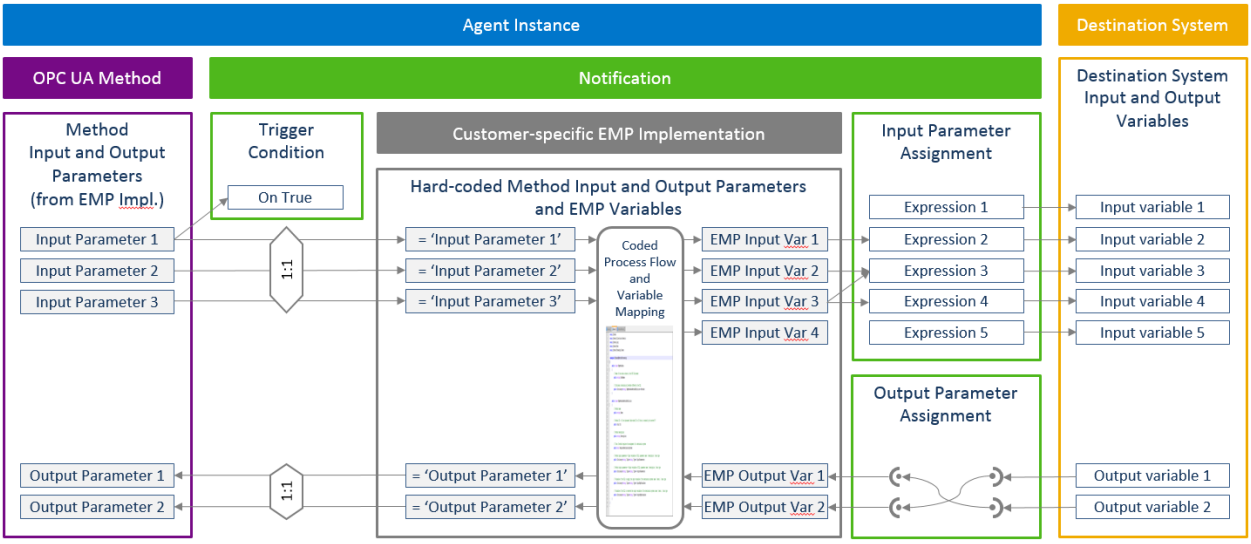



Figure 4 Enhanced Method Processing with destination system call

The above picture shows the enhancement that is necessary when you call a destination from the EMP implementation. Only the implementer knows how to call the destination and which input and output variables have to be filled in order to invoke the destination in the right way. It must not be assumed that there is a direct relation between method parameters and destination variables because the variables for a destination call could also be calculated within the EMP implementation. In this case they don't come from the method and therefore the method has no parameters which hold values for the destination call.

For the above case the EMP variables have been introduced. Figure 4 shows the mapping of the EMP variables. From the UA method to the EMP implementation the mapping is the same as in figure 3. Additionally the implementation uses EMP variables for the call of the destination system.

The PCo management console fetches EMP metadata via method `GetEMPVariables`.

# 5 How to Implement an Enhanced Method Processing Project

## 5.1 Overview of Implementation Steps

The implementation of a customer-owned enhancement consists mainly of two parts:

1. Implementation activities in Microsoft Visual Studio:
   - Create a class that inherits from the class `ApplicationMethodBase`
   - Create implementations of the methods `getMethodDefinitionsCustom` and `executeCustom`
   - Build an EMP DLL (dynamic link library) from your class.
2. Configuration activities in the PCo Management Console:
   - Create destination systems for every destination system that you want to call from the customer-owned enhancement implementation.
   - Maintain the destination system settings and operation configuration in the destination system.
   - Create an agent instance, and link your EMP DLL to the agent instance.
   - Import UA method definitions from the EMP DLL
   - Create a method based notification per imported method in order to invoke the EMP code at runtime
   - Create a mapping for method parameters, notification variables and EMP variables

The following sections describe the individual activities in detail.

## 5.2 Implementation Activities in Microsoft Visual Studio

### 5.2.1 Prerequisites

To create a customer-owned enhancement, you require an integrated development environment for .NET development. SAP recommends using Microsoft Visual Studio 2015 or higher. The following steps are explained by using screenshots and settings from Microsoft Visual Studio 2015.

If you want to use an EMP DLL provided by a third party, you can skip this section and proceed directly to section 5.3. Note that the DLL has to be built for .NET framework 4.0 with platform target *Any CPU*.

## 5.2.2 Create a Solution and an Implementing Class

First create a new solution for your implementation. Make sure you choose *.NET Framework 4.0*. Use the *Class Library* template.

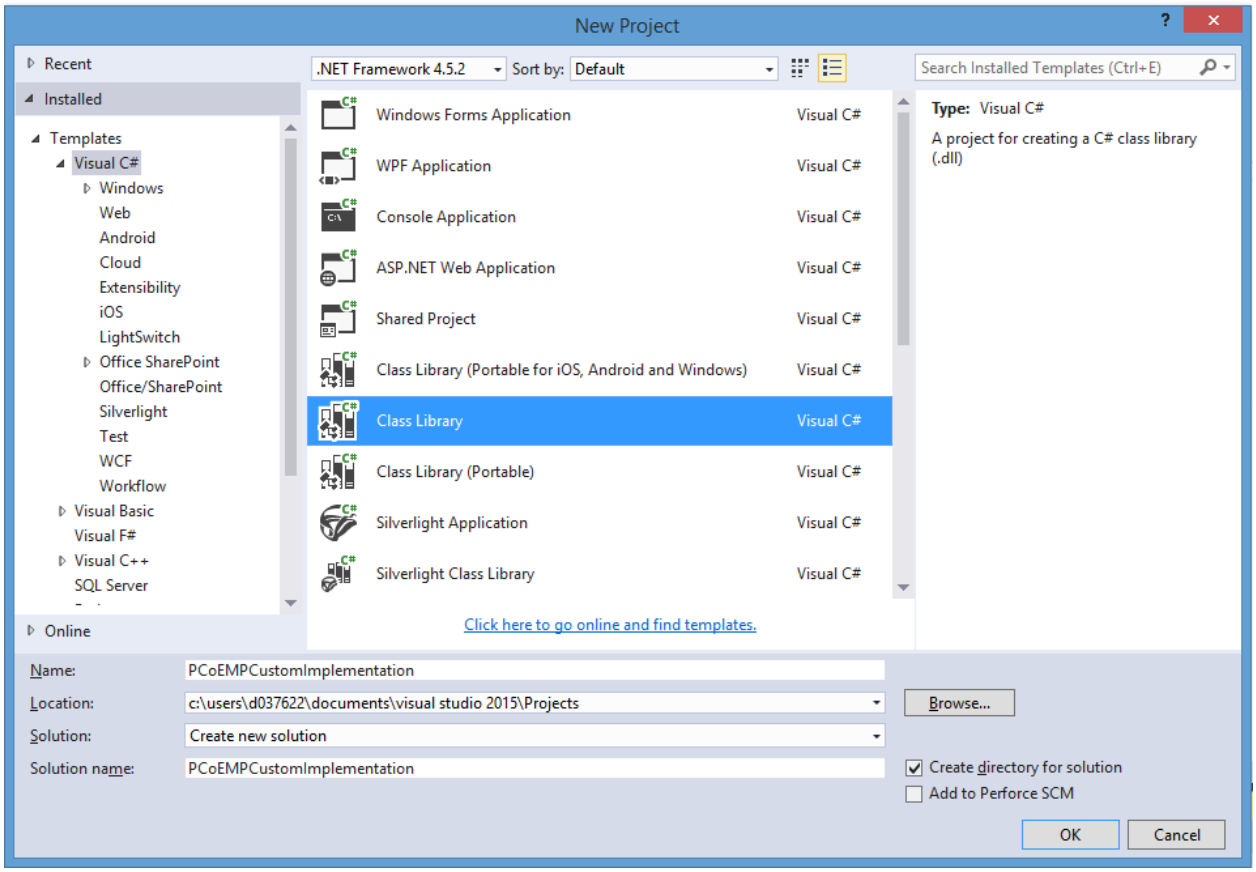



Figure 5: Create a New Solution

Then set the properties of your new project. On the Application Properties tab, check if the target framework is set to *.NET Framework 4.5.2* and the output type is set to *Class Library*.

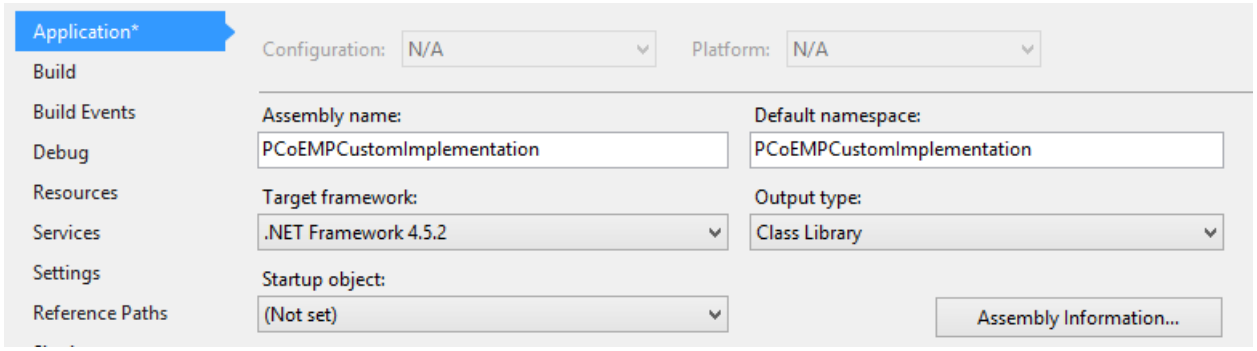



Figure 6: Application Properties

Make sure that the platform target is set to *Any CPU* for all configurations.

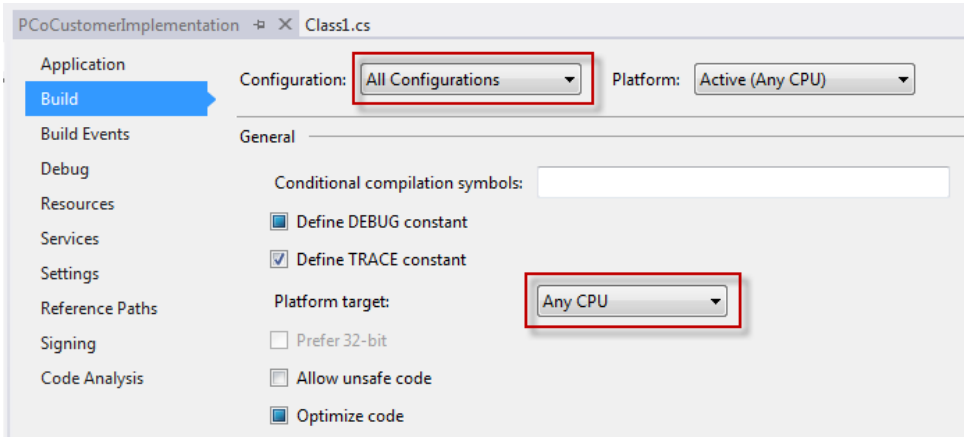



Figure 7: Project Build Configuration

Now create a reference to the PCo `EMP-Method` DLL for your project.

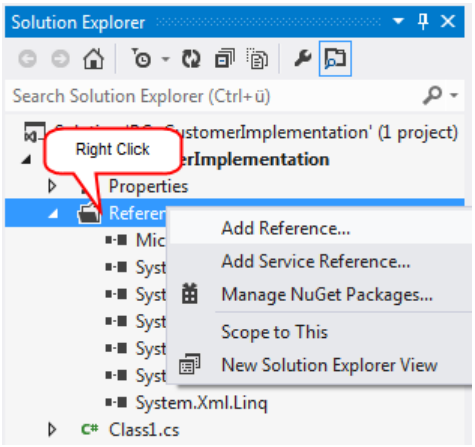



Figure 8: Adding References to Your Project

Choose *Browse* then select the *MethodProcessingFramework.dll* from the *System* subfolder of the PCo program folder. The *References* branch of your project should now contain the reference to the MethodProcessingFramework.dll.

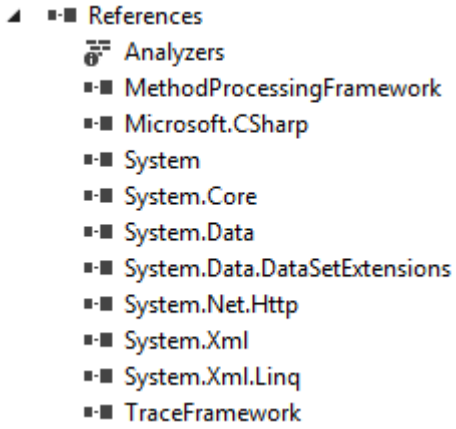



Figure 9: Required Project References

You now have a solution with an almost empty class, for example, `Class1.cs`, which could look as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Xml;
using System.IO;
using SAP.Manufacturing.MethodProcessingFramework;
```

```
namespace PCoCustomerImplementation
{
    public class Class1:ApplicationMethodsBase

    {
    }
}
```

**Note that Class1 has to derive from** `ApplicationMethodsBase`**.**

## 5.2.3 Implement Required Methods

In order to use your implementation as an enhancement implementation in PCo, you have to implement the following methods:

### 5.2.3.1 executeCustom

This method is used to call the desired EMP method/implementation. Therefore the generic method `callEMPMethod` has been introduced to the `ApplicationMethodBase` class. One of the method parameters is the `destinationCallback`. If you want your DLL to call a destination within the one or the other EMP method it is best practice to store it within a class variable. So the entire implementation of the executeCustom method is very small:

```
protected override void executeCustom(Guid methodId, DestinationCallback destinationCallback, Dictionary<string, object> inputVariables, out Dictionary<string, object> outputVariables)
{
    currentDestinationCallback = destinationCallback;

    base.callEMPMethod(methodId, inputVariables, out outputVariables);
}
```

### 5.2.3.2 getMethodDefinitionsCustom

This method is used to expose EMP method metadata to the outside. Method parameters as well as EMP variables are defined here. PCo will be able to read EMP metadata and import the method definitions accordingly.

The base class contains an attribute called `empMetaData`. This attribute has to be returned by this method. As best practice create a helper method (for example createEMPMetaData()). This method creates the EMP metadata content and stores it within the attribute `empMetaData`. Call the helper method from the constructor of your EMP class ("Class1" in the above example).

For each method you'd like to expose you have to generate a unique GUID as identifier. To create a GUID in visual studio chose "Create GUID" from the Tools menu.

As best practice store this GUID within a static attribute of your class. For example like this:

```
private static Guid QuickCompleteMethodId           = new Guid("2D9F7909-22F1-48FC-A5F5-6401EF07EB0A");
```

Create also a GUID for the node id of the node that should be visible on the server. You need this node id to bundle all your EMP methods under a particular server node. As best practice also store this node within a static attribute of your class.

Example: Imagine that you implement EMP methods for a server node called "ME_Buffer". In this case you have to initialize the `empMetaData` attribute in the following way:

```
empMetaData.Id = MEBufferNodeId;
empMetaData.Description = "Buffering Manufacturing Execution Plan Information";
empMetaData.NodeName = "ME_Buffer";
empMetaData.Methods = new Dictionary<Guid, MethodMetaData>();
```

"MEBufferNodeId" is the static attribute that contains the GUID for the server node under which the EMP methods are bundled. In addition you have to create a description and a node name. The example above would create a server node called "ME_Buffer" on the UA server where you import the EMP methods.

The dictionary `empMetaData.Methods` is used to store the metadata information of the EMP methods.

The following section shows an example how to fill the metadata of a method called _QuickComplete.

```
//Method QuickComplete
MethodMetaData quickCompleteMetaData = new MethodMetaData();
quickCompleteMetaData.Id = QuickCompleteMethodId;
quickCompleteMetaData.Name = ME_BufferConstants.methodQuickComplete;
quickCompleteMetaData.Description = ME_BufferResources.QuickCompleteMethodDescription;
quickCompleteMetaData.RequiresDestinationSystem = true;

quickCompleteMetaData.InputParameters    = new Dictionary<string, Tuple<string, Type>>();
quickCompleteMetaData.OutputParameters   = new Dictionary<string, Tuple<string, Type>>();
quickCompleteMetaData.InputEmpVariables  = new Dictionary<string, Tuple<string, Type>>();
quickCompleteMetaData.OutputEmpVariables = new Dictionary<string, Tuple<string, Type>>();

//Input Parameters
quickCompleteMetaData.InputParameters.Add(ME_BufferConstants.inSite, new Tuple<string, Type>(ME_BufferResources.site, Type.GetType("System.String")));
quickCompleteMetaData.InputParameters.Add(ME_BufferConstants.inSfcNumber, new Tuple<string, Type>(ME_BufferResources.sfcNumber, Type.GetType("System.String")));
quickCompleteMetaData.InputParameters.Add(ME_BufferConstants.inOperation, new Tuple<string, Type>(ME_BufferResources.operation, Type.GetType("System.String")));
quickCompleteMetaData.InputParameters.Add(ME_BufferConstants.inResource, new Tuple<string, Type>(ME_BufferResources.resource, Type.GetType("System.String")));
quickCompleteMetaData.InputParameters.Add(ME_BufferConstants.inCallDestination, new Tuple<string, Type>(ME_BufferResources.callDestinationIdentifier, Type.GetType("System.Boolean")));

//Output Parameters
quickCompleteMetaData.OutputParameters.Add(ME_BufferConstants.outNextOperation, new Tuple<string, Type>(ME_BufferResources.nextOperation, Type.GetType("System.String")));

//EMP Input Variables (= input variables mapped against the web service input parameters)
quickCompleteMetaData.InputEmpVariables.Add(ME_BufferConstants.inSfcRef, new Tuple<string, Type>(ME_BufferResources.sfcRef, Type.GetType("System.String")));
quickCompleteMetaData.InputEmpVariables.Add(ME_BufferConstants.inOperationRef, new Tuple<string, Type>(ME_BufferResources.operationRef, Type.GetType("System.String")));
quickCompleteMetaData.InputEmpVariables.Add(ME_BufferConstants.inResourceRef, new Tuple<string, Type>(ME_BufferResources.resourceRef, Type.GetType("System.String")));

//EMP Output Variables (= output variables mapped against the web service response)
quickCompleteMetaData.OutputEmpVariables.Add(ME_BufferConstants.outStepId, new Tuple<string, Type>(ME_BufferResources.nextOperation, Type.GetType("System.String")));

empMetaData.Methods.Add(QuickCompleteMethodId, quickCompleteMetaData);
```

Therefore use the structure `MethodMetaData` which is defined in the MethodProcessingFramework. Assign the static attribute with the GUID to the attribute "Id" of the `MethodMetaData`. Then assign method name and method description to the fields "Name" and "Description" of `MethodMetaData`. The method of the above example has input and output parameters as well as input and output EMP variables. In this case you have to create four empty dictionaries as shown in the text above. Then start with the definition if parameters. When you add an entry to one of the parameter dictionaries the parameter of the "Add" method are defined as follows:

ParameterName, Tuple<string parameterDescription, Type parameterType>.

## 5.2.4 Implement the actual method(s)

As mentioned already the method identifier (GUID) is part of the parameters of the `execute` method which in turn is called at runtime to invoke the actual implementation. According to the example of the last chapter and its metadata definition you need a method within your class that holds the actual implementation of your functionality. According to the example above you would have the need to implement a method "_QuickComplete" like this:

```
public Dictionary<string, object> _QuickComplete(string inSite, string inSFC, string inOperation, string inResource, bool callDestination)
{
```

... here comes your own logic .....

... and a potential destination call....

The interface of this method contains the real parameter that you need for within the method implementation.

Best Practice: The above method is one of a type that uses the possibility to call an external service. If you configure your system the external service (like a web service for example) might not be available all the time. To avoid unnecessary errors or delays include an indicator (in this case "callDestination") that switches the destination call on or off depending on its setting. So when you configure your choreography in PCo you can switch the external service call off if you want to.

The following paragraph shows an example on how you might call the external service within the "_QuickComplete" implementation:

```
//call further destination if required
if (callDestination == true)
{
    //prepare the input variables for the ME service call
    Dictionary<string, object> destinationInput = new Dictionary<string, object>();
    destinationInput.Add(this.empMetaData.Methods[QuickCompleteMethodId].InputEmpVariables.ElementAt(0).Key, "SFCBO:" + inSite + "," + inSFC);
    destinationInput.Add(this.empMetaData.Methods[QuickCompleteMethodId].InputEmpVariables.ElementAt(1).Key, "OperationBO:" + inSite + "," + inOperation + ",#");
    destinationInput.Add(this.empMetaData.Methods[QuickCompleteMethodId].InputEmpVariables.ElementAt(2).Key, "ResourceBO:" + inSite + "," + inResource);

    //call ME service
    Dictionary<string, object> destinationResult = currentDestinationCallback.callDestination(destinationInput, true);
}
```

Within the above code you first of all create a dictionary as input for the destination call. Then you add input parameters for the destination call. Therefore you use the parameter names from the EMP metadata input parameters. The actual call of the destination is done via the method `callDestination` of the `destinationCallback`. At this point you are able to see that it is useful to store the `destinationCallback` within a class attribute (here: currentDestinationCallback) in order to have it available here.

You might have been realized that the definition of the EMP metadata used EMPOutput-Variables. But in the particular case of the example there was no need to assign the destination output. Therefore nothing is done with the destination result.

Create an implementation like this for each method you have defined within the metadata section. If your method doesn't use EMP variables only assign its input- and/or output parameters.

## 5.2.5 Build the EMP DLL

After you have finished the implementation, you have to 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.

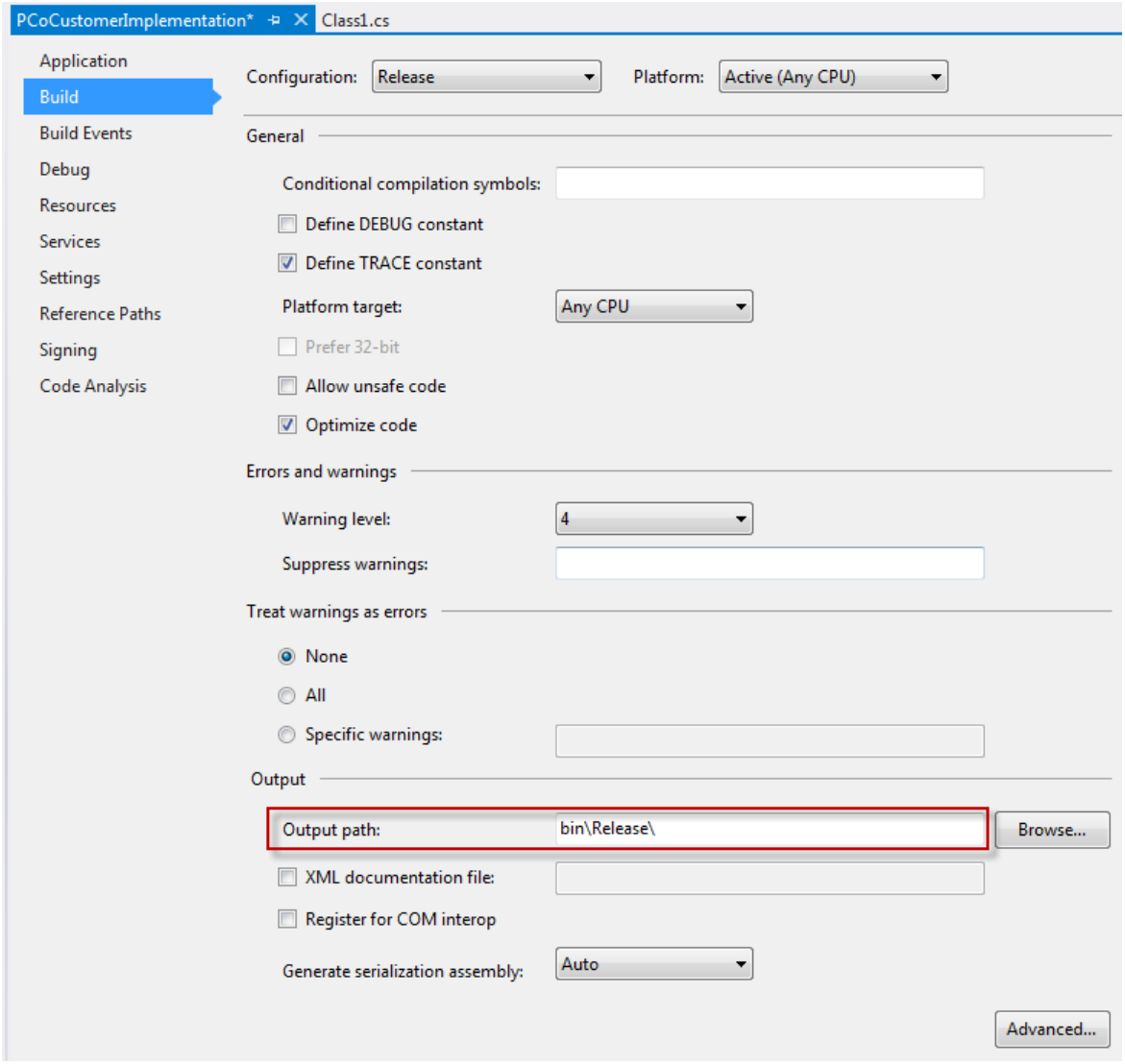



Figure 10: Check the Output Path for the ENP DLL

The resulting DLL is named `<Project Name>.DLL`, for example, `PCoMECustomImplementation.dll`

It is recommended that you copy the DLL into the System folder of the PCo installation. On Windows 32-Bit installation, this folder is named *<Installation Drive>:\Program Files\SAP\Plant Connectivity\System*, in 64-Bit systems *<Installation Drive>:\Program Files (x86)\SAP\Plant Connectivity\System*. If the EMP DLL resides in a different folder, the dependent PCo DLLs are duplicated into this folder once the EMP DLLs are loaded during the runtime of the Management Console.

## 5.3 Configuration Activities in the PCo Management Console

### 5.3.1 Prerequisites

Before you can configure the EMP, you create the EMP DLL as described in the previous section 5.2; in particular all of the methods mentioned in section 5.2.3 have to be implemented. Alternatively, you could use a third-party DLL that inherits from the `ApplicationMethodBase` and is compiled for .NET framework 4.5.2 and platform target *Any CPU*.

Furthermore, you have to define an agent instance with a source system of your choice. The source system is not important for the matter of EMP. Within the server configuration of the agent an OPC UA server has to be defined in order to be able to import EMP method definitions.

### 5.3.2 Import EMP method definitions

Press the "Add" Button to create new method definitions within the PCo UA server.

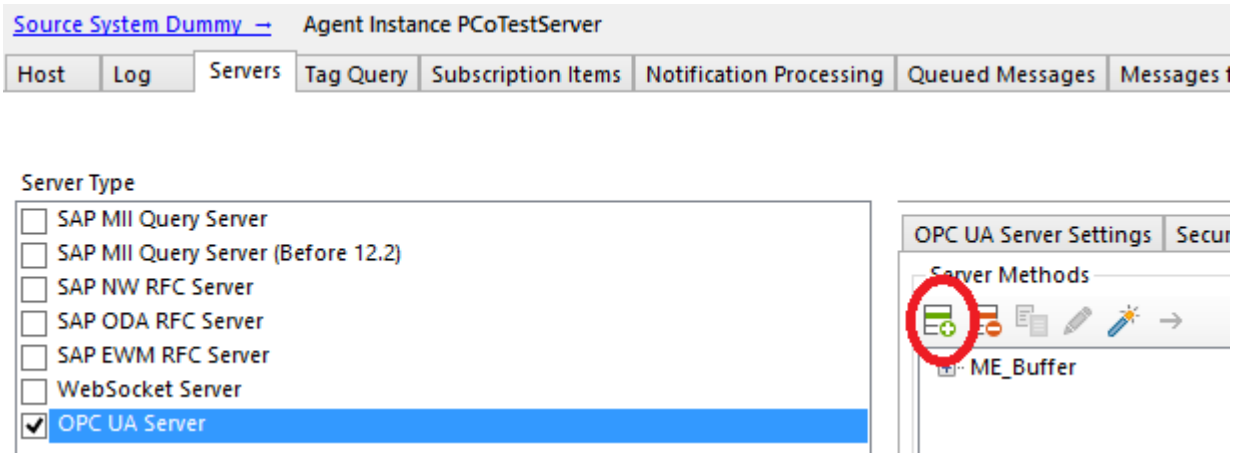



Figure 11 Add Method Button

Choose the option to load the method definitions from a foreign assembly.

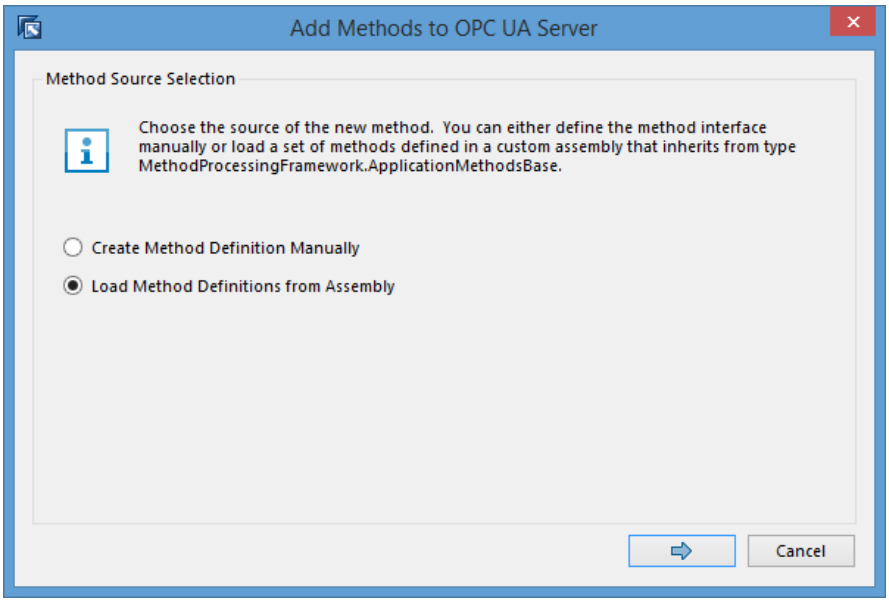



Figure 12 Load Method Definitions from Assembly

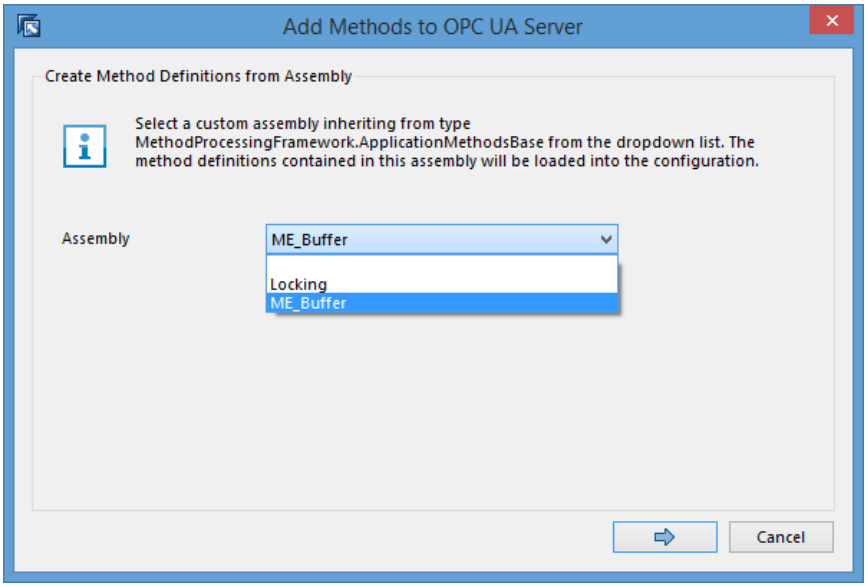



Figure 13 Selection of Assembly

Per default each of the methods are selected. Al selected method means that PCo will create a method based notification per method automatically. You can also prevent PCo from creating method based notifications by deselecting the respective method(s).

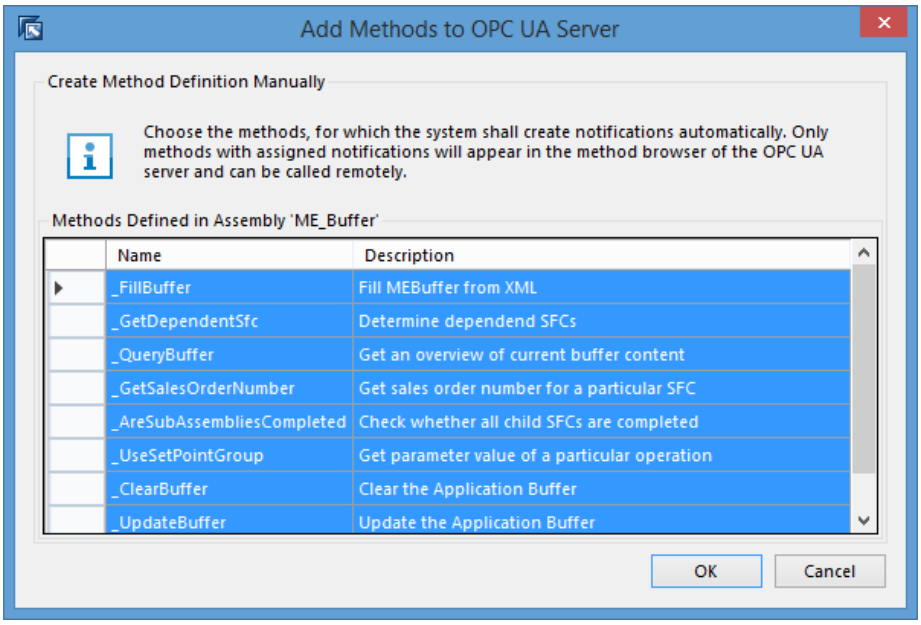



Figure 14 Default Notification Creation

After you have imported the method definitions the respective notifications should have been created

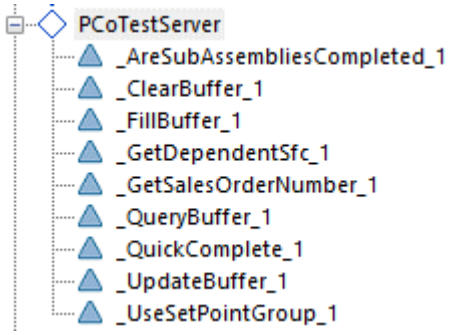



Figure 15 Method Based Notifications

## 5.3.3 Notification Maintenance

For those notifications without EMP variables the method parameters are routed 1:1 to the respective method implementations. This means that the management console doesn't offer an additional parameter mapping in this case.

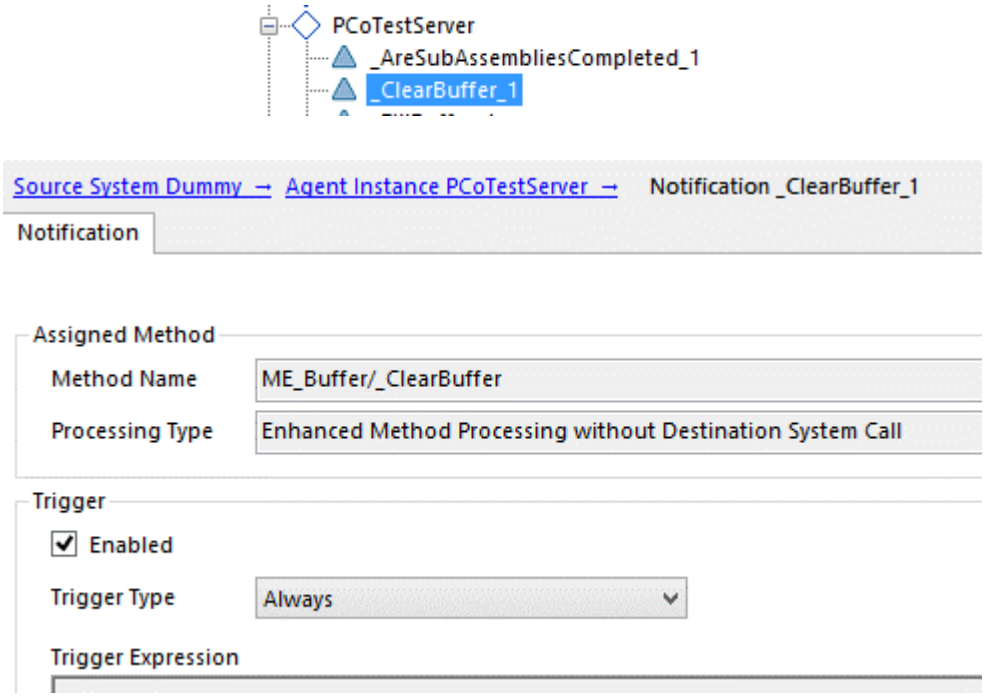



Figure 16 Notification without further parameter mapping

For methods with EMP variable assignment a further parameter mapping is offered by the PCo management console. You will use such methods to call additional services (destinations) by the help of the PCo connectivity framework.

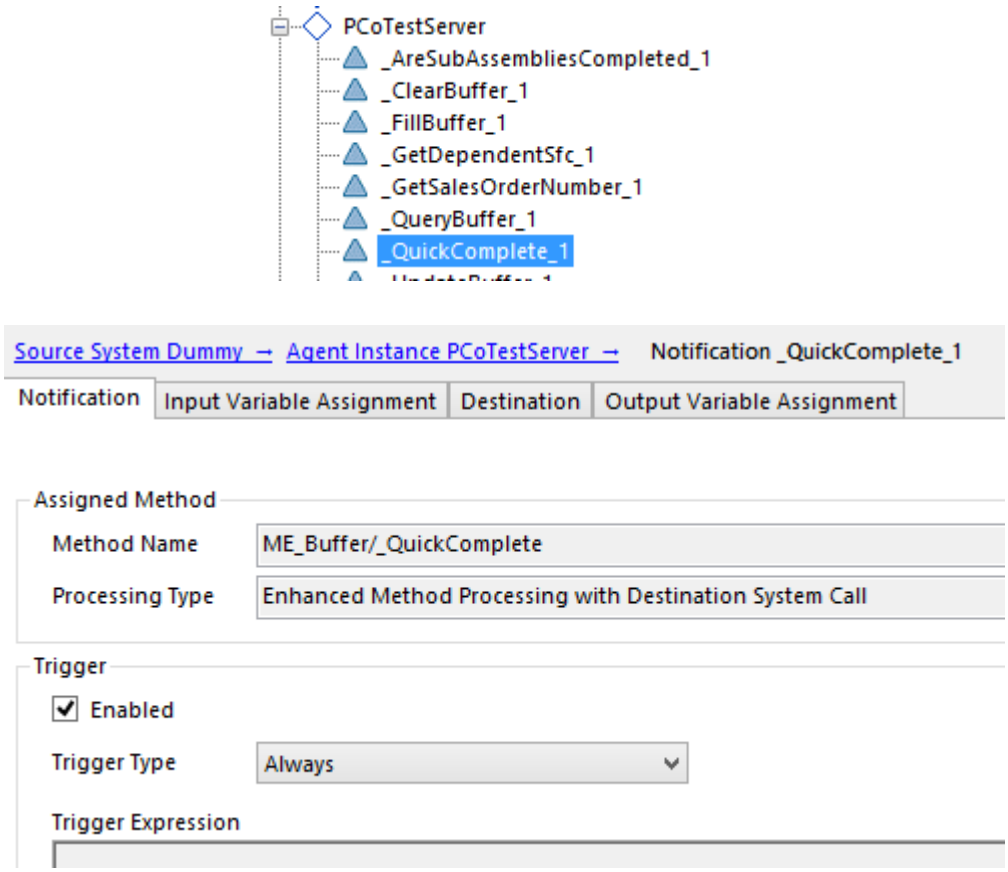



Figure 17 Notification with additional parameter mapping

For the input variable assignments recommendations are generated. They are taken from the EMP variable definitions of the foreign assembly.  If you don't have an urgent need you don't have to change it.

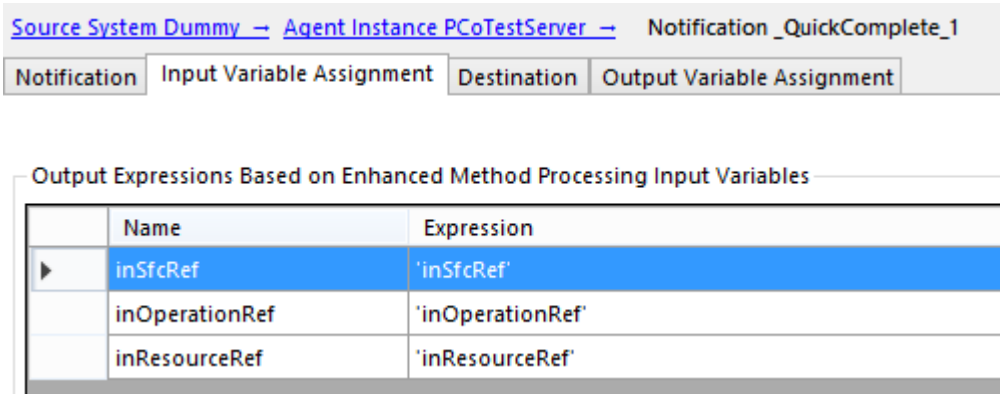



Figure 18 Input Variable Assignment

Within the destination assignment you map service variables (column "Variable") against your notification output from figure 18.

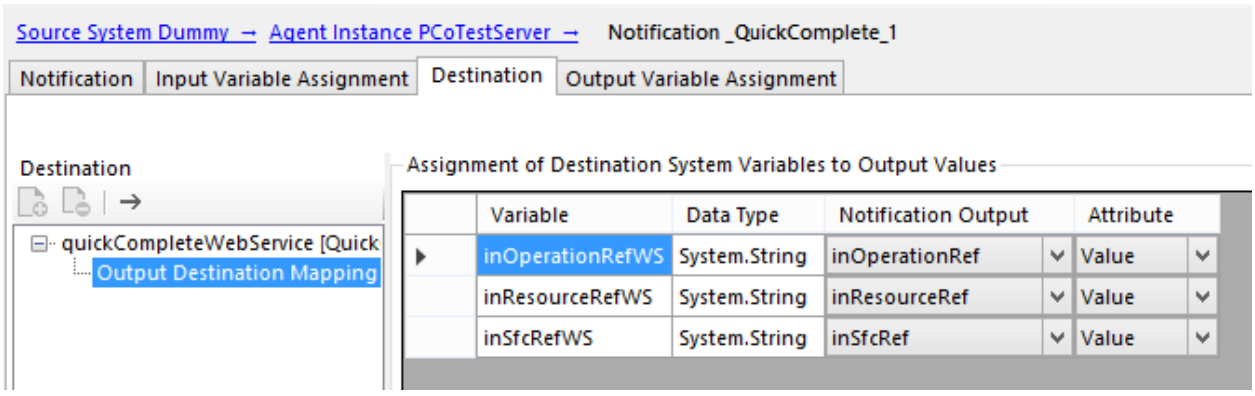



Figure 19 Notification destination variable mapping

Finally you have to assign the output variables

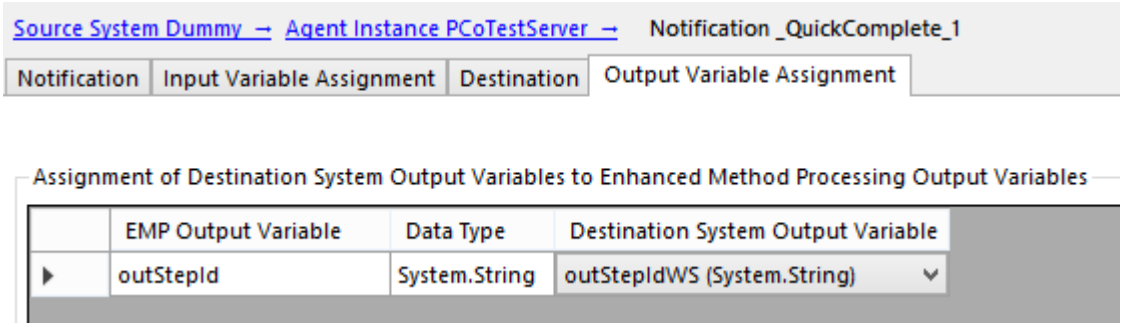



Figure 20 Notification output variable assignment

# 6 Example Implementation (Simple EMP Method)

Within the following chapter it is described how to implement and include an example for an EMP method. The method will simply add two integers and will give back the result.

## 6.1 Creation of a class library

As described earlier we start with the creation of a class library in visual studio 2015.

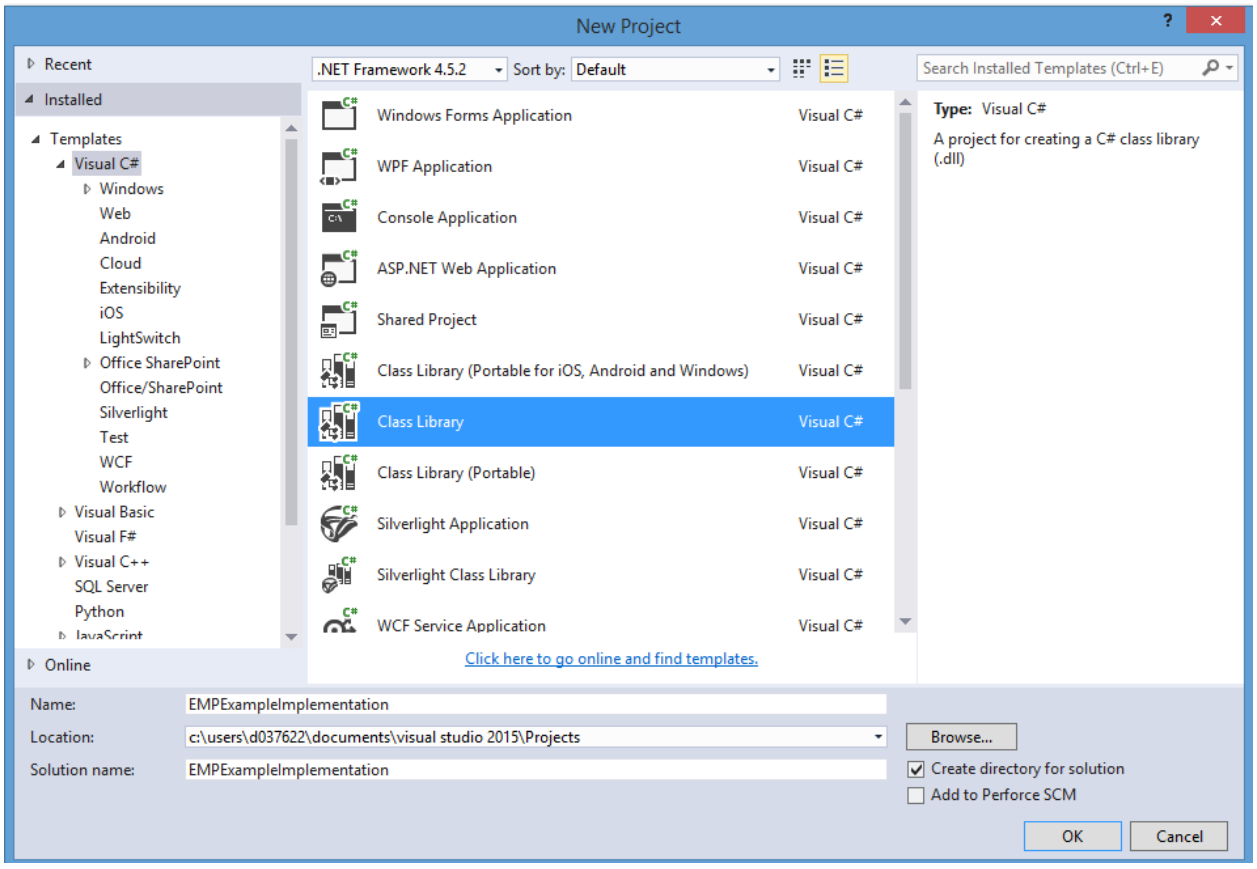



Figure 21 Creation an example class library

## 6.2 Maintain References

Add the reference to the MethodProcessingFramework.dll

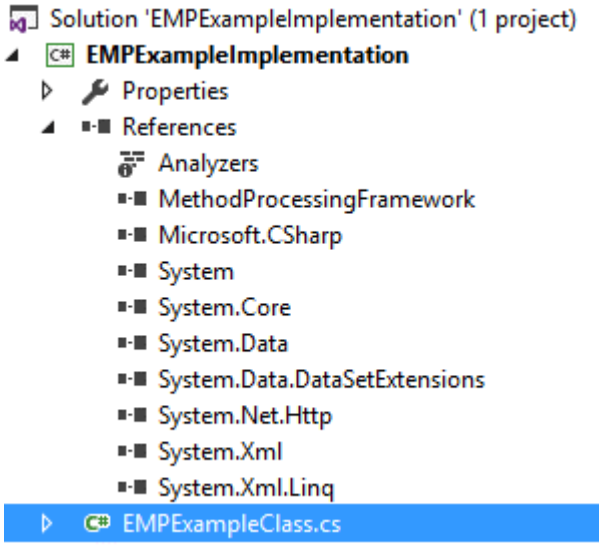



Figure 22 References of example implementation

## 6.3 Create Implementation Class

Within the implementing class inherit from `ApplicationMethodBase and implement all required methods:`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Xml;
using System.IO;
using SAP.Manufacturing.MethodProcessingFramework;

namespace EMPExampleImplementation
{
    public class EMPExampleClass : ApplicationMethodsBase
    {
        protected override void executeCustom(Guid methodId, Destinatior
        {
            throw new NotImplementedException();
        }

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

Figure 23 Raw implementation

Create a new GUID for the server node under which the message should appear. Create a new attribute that stores this GUID in your class.

```
public class EMPExampleClass : ApplicationMethodsBase
{
    private static Guid empTestNodeId = new Guid("0BB7EF9A-255E-46C8-9011-2AE337FC745C");
```

Figure 24 Method node GUID

Create a GUID for each particular method that you want to expose. In our example we have only one method.

```
public class EMPExampleClass : ApplicationMethodsBase
{
    private static Guid empTestNodeId      = new Guid("0BB7EF9A-255E-46C8-9011-2AE337FC745C");
    private static Guid sumIntegerMethodId = new Guid("57B3C304-CA81-449D-98DE-2345E045F92D");
```

Figure 25 Method GUID

Implement the method `getMethodDefinitionsCustom()` to return the metadata of the method you want to expose.

```
protected override EmpMetaData getMethodDefinitionsCustom()
{
    return empMetaData;
}
```

Figure 26 Inherited Method for metadata definitions

At this point it is enough to just return the structure `empMetaData` that is defined within the father class. We will use a helper method to create the metadata at a later point in time.

Create a constructor, define a helper method for the metadata definitions and call it from the constructor.

```
public class EMPExampleClass : ApplicationMethodsBase
{
    private static Guid empTestNodeId      = new Guid("0BB7EF9A-255E-46C8-9011-2AE337FC745C");
    private static Guid sumIntegerMethodId = new Guid("57B3C304-CA81-449D-98DE-2345E045F92D");

    public EMPExampleClass()
    {
        empMetaData = createEMPExampleMetaData();
    }

    private EmpMetaData createEMPExampleMetaData()
    {
        throw new NotImplementedException();
    }
```

Figure 27 Meta data helper method

Implement the method "createEMPExampleMetaData".

```
private EmpMetaData createEMPExampleMetaData()
{
    EmpMetaData empMetaData = new EmpMetaData();

    empMetaData.Id = empTestNodeId;
    empMetaData.Description = "Test Implementation of EMP method";
    empMetaData.Methods = new Dictionary<Guid, MethodMetaData>();

    //Method SumInt
    MethodMetaData sumIntMetaData = new MethodMetaData();
    sumIntMetaData.Id = sumIntegerMethodId;
    sumIntMetaData.Name = "SumInt";
    sumIntMetaData.Description = "Calculate sum of two integers";
    sumIntMetaData.RequiresDestinationSystem = false;

    sumIntMetaData.InputParameters = new Dictionary<string, Tuple<string, Type>>();
    sumIntMetaData.OutputParameters = new Dictionary<string, Tuple<string, Type>>();


    //Input Parameters
    sumIntMetaData.InputParameters.Add("inIntOne", new Tuple<string, Type>("First Integer", Type.GetType("System.Int32")));
    sumIntMetaData.InputParameters.Add("inIntTwo", new Tuple<string, Type>("Second Integer", Type.GetType("System.Int32")));

    //Output Parameters
    sumIntMetaData.OutputParameters.Add("outSumInteger", new Tuple<string, Type>("Sum of two integers", Type.GetType("System.Int32")));

    empMetaData.Methods.Add(sumIntegerMethodId, sumIntMetaData);

    return empMetaData;
}
```

Figure 28 Meta Data for Integer Calculation Method

Implement the runtime method "executeCustom".

```
protected override void executeCustom(Guid methodId, DestinationCallback dest
{
    currentDestinationCallback = destinationCallback;

    base.callEMPMethod(methodId, inputVariables, out outputVariables);
}
```

Figure 29 executeCustom implementation

At this point it is a good style if you copy the current destination callback instance into a private class variable. If you have implemented a method that calls a destination you will need a reference to the current destination callback within the implementing method.

Then call the method "callEMPMethod" of the base class. This will trigger the actual call of your method implementation.

Within the meta data implementation you have defined a name for the method that is assigned to a method identifier (GUID) and you have stored this information within the empMetaData Dicationary. The method "callEMPMethod" of the base class resolves this dependency and calls the method in a generic way.

The last step (the actual implementation) is missing. So you have to create the method "SumInt" within your implementation. Therefore have an eye on using the exact parameters that you have specified within the meta data section.

```
public Dictionary<string, object> SumInt(int inIntOne, int inIntTwo)
{
    Dictionary<string, object> output = new Dictionary<string, object>();

    output.Add(empMetaData.Methods[sumIntegerMethodId].OutputParameters.ElementAt(0).Key, inIntOne + inIntTwo);

    return output;
}
```

Figure 30 Actual EMP method implementation

In this case you could pack the entire code of the method into the return statement. But to make things more clear a dictionary is defined that holds the output variables and their values. The first output parameter for which the name is defined within the "empMetaData" dictionary is assigned to the sum of the input parameters of the method.

This concludes the implementation steps.

## 6.4 Build DLL

Next step is to build the DLL within a folder of your choice. Within this example we want to build the DLL in the bin/Release folder of the solution directory.

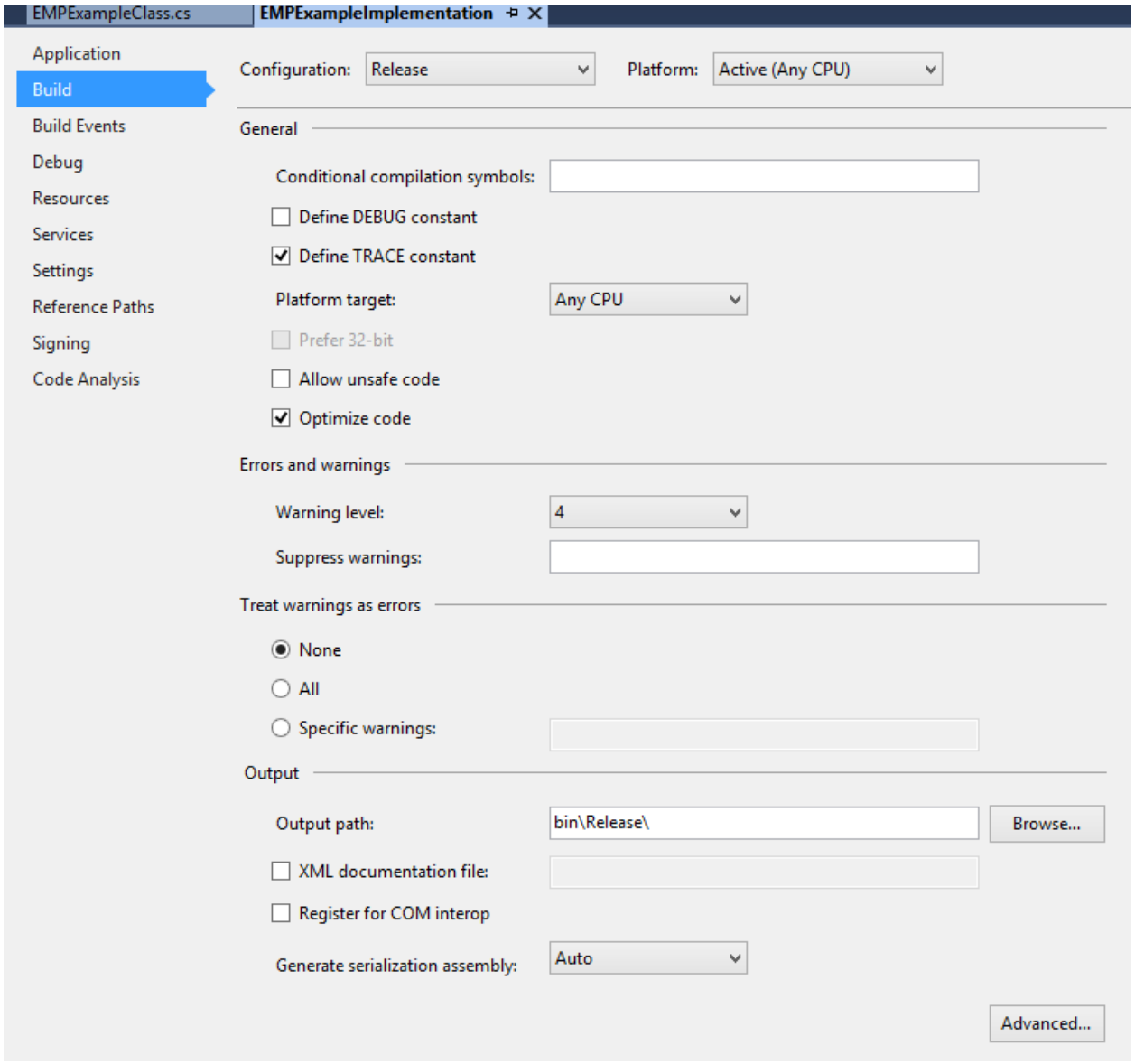
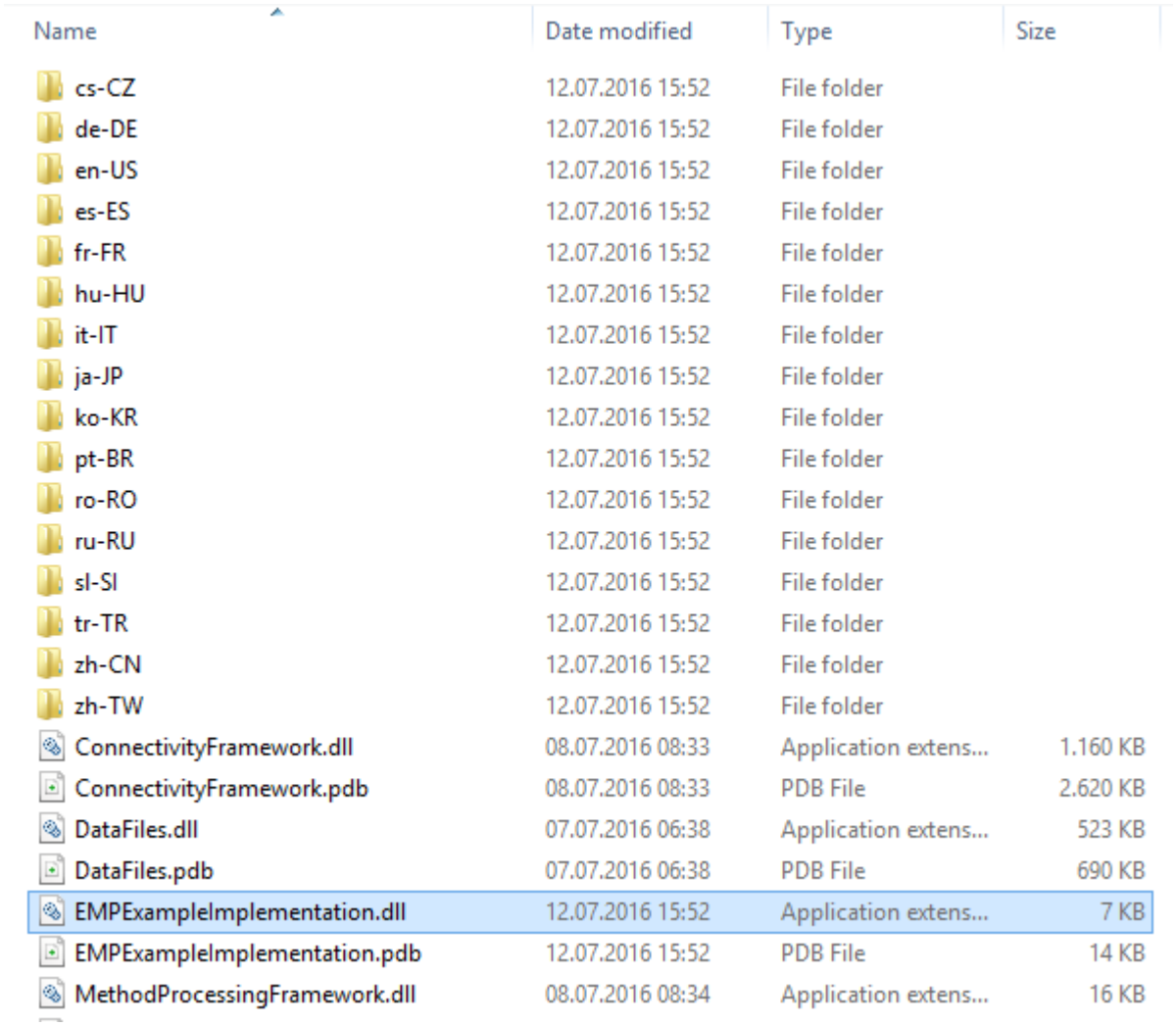



Figure 31 Build Configuration

The result should look like this:



| Name | Date modified | Type | Size |
|---|---|---|---|
| cs-CZ | 12.07.2016 15:52 | File folder | |
| de-DE | 12.07.2016 15:52 | File folder | |
| en-US | 12.07.2016 15:52 | File folder | |
| es-ES | 12.07.2016 15:52 | File folder | |
| fr-FR | 12.07.2016 15:52 | File folder | |
| hu-HU | 12.07.2016 15:52 | File folder | |
| it-IT | 12.07.2016 15:52 | File folder | |
| ja-JP | 12.07.2016 15:52 | File folder | |
| ko-KR | 12.07.2016 15:52 | File folder | |
| pt-BR | 12.07.2016 15:52 | File folder | |
| ro-RO | 12.07.2016 15:52 | File folder | |
| ru-RU | 12.07.2016 15:52 | File folder | |
| sl-SI | 12.07.2016 15:52 | File folder | |
| tr-TR | 12.07.2016 15:52 | File folder | |
| zh-CN | 12.07.2016 15:52 | File folder | |
| zh-TW | 12.07.2016 15:52 | File folder | |
| ConnectivityFramework.dll | 08.07.2016 08:33 | Application extens... | 1.160 KB |
| ConnectivityFramework.pdb | 08.07.2016 08:33 | PDB File | 2.620 KB |
| DataFiles.dll | 07.07.2016 06:38 | Application extens... | 523 KB |
| DataFiles.pdb | 07.07.2016 06:38 | PDB File | 690 KB |
| EMPExampleImplementation.dll | 12.07.2016 15:52 | Application extens... | 7 KB |
| EMPExampleImplementation.pdb | 12.07.2016 15:52 | PDB File | 14 KB |
| MethodProcessingFramework.dll | 08.07.2016 08:34 | Application extens... | 16 KB |

Figure 32 Structure within folder bin/Release

Copy the EMPExampleImplementation.dll into the System folder of your PCo installation.

## 6.5 Import New DLL/Method

Start the management console and import your DLL within a server of your choice

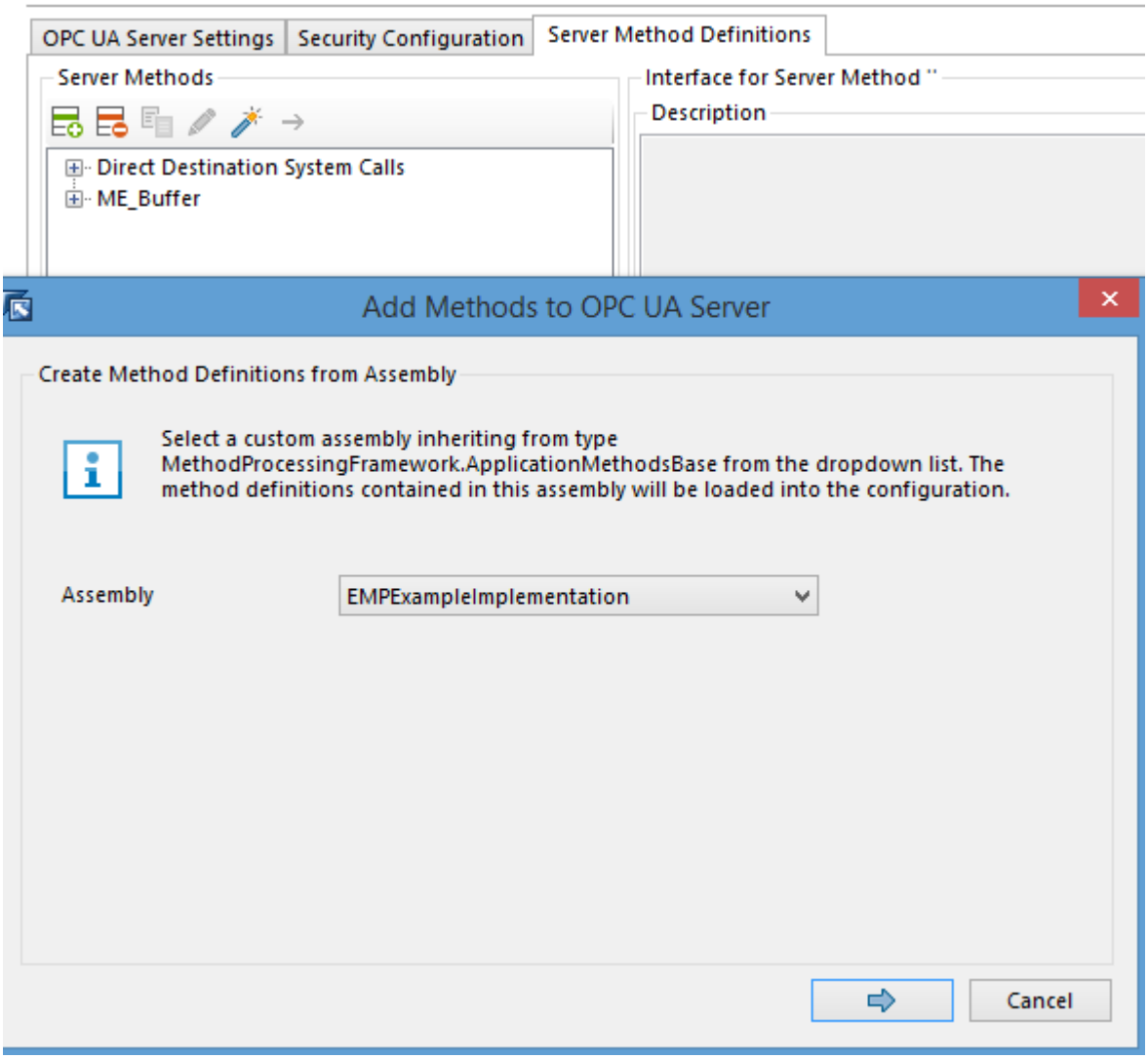



Figure 33 Import your DLL

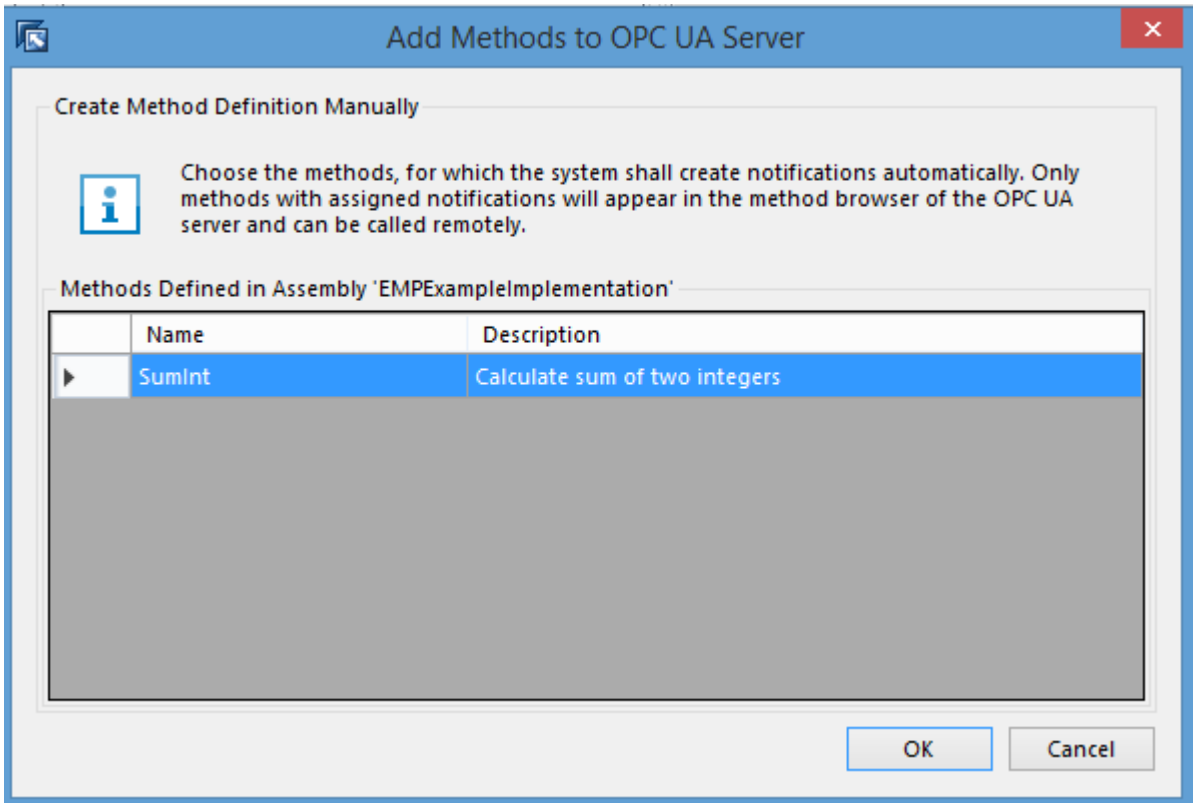



Figure 34 Your method

You should now see a server node and a notification for your method:

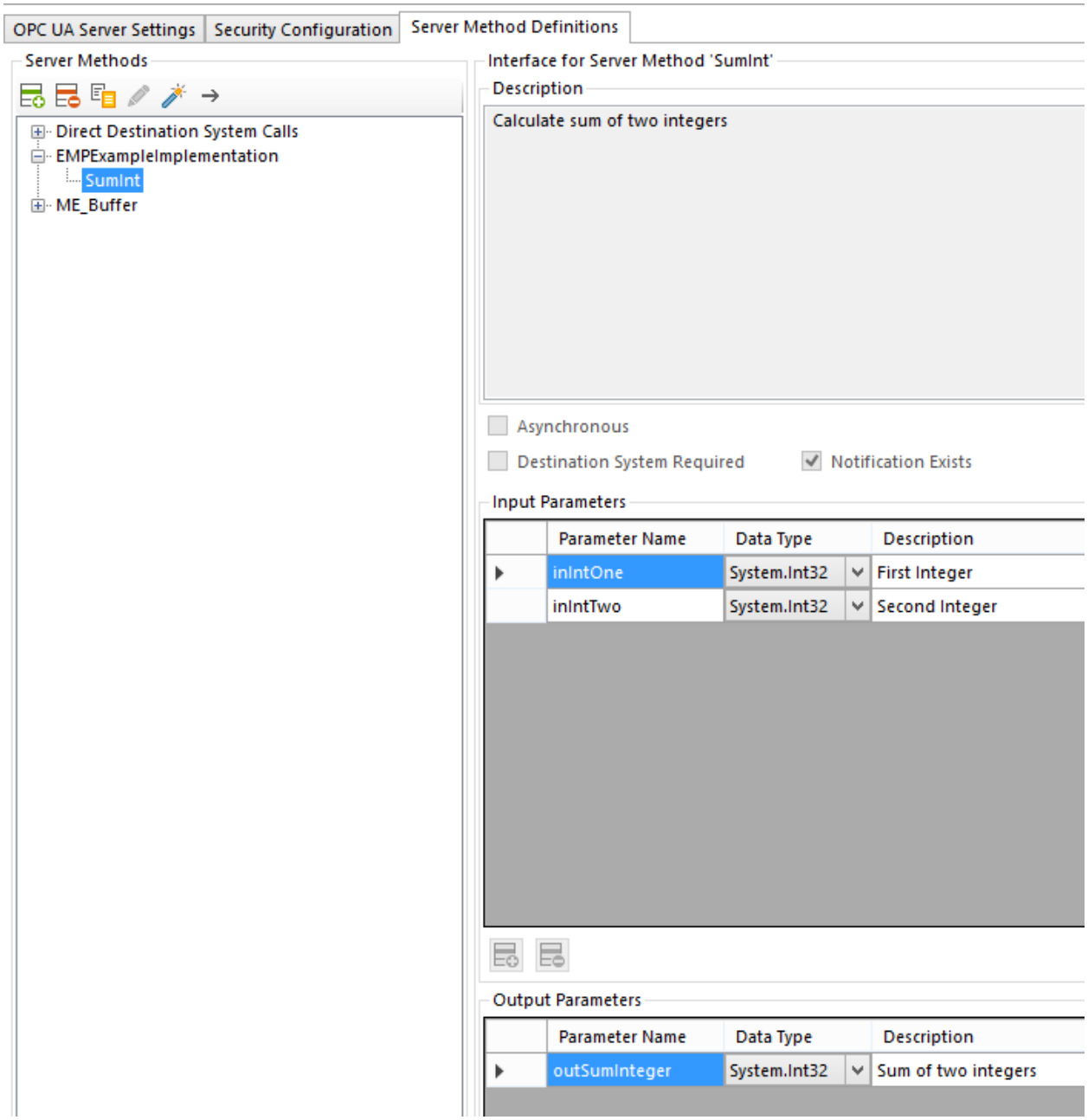



Figure 35 New Server Node and method

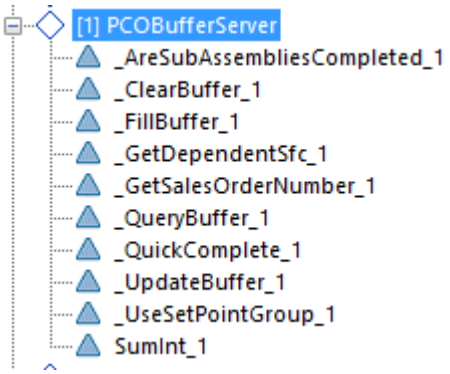



Figure 36 New notification "SumInt_1"

Now you are ready to connect your server from any client (example OPC UA expert) and call the new method:

## 6.6 Test Method

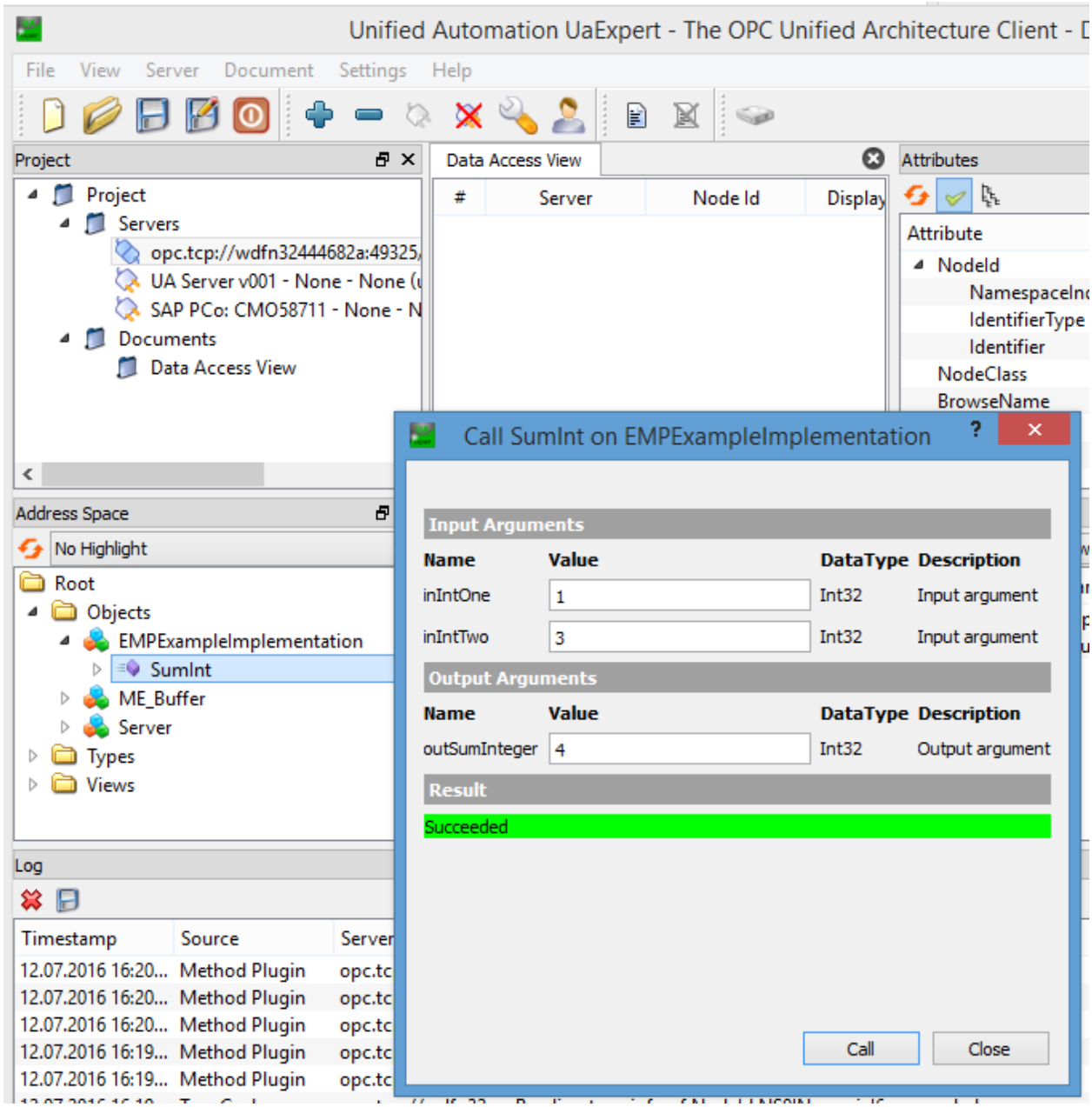



Figure 37 Call of the new EMP method

Congratulations. You have successfully created your first EMP DLL.

# 7 EMP Example Implementation (Method with EMP Variables)

In general there is not much difference in the approach of creating and implementing a DLL that uses EMP variables so the following chapters will focus on the differences within the implementation.

As already said the sense of EMP variables is to use them to call additional services from your implementation while using the PCo connectivity framework for the actual service call.

This service can be any PCo destination of your choice. Refer to the PCo documentation on how to maintain a PCo destination.

## 7.1 Implementation Class

### 7.1.1.1 Meta Data Definition

Within the metadata definition you have to foresee a section for EMP variables:

```
//EMP Input Variables (= input variables mapped against the web service input parameters)
quickCompleteMetaData.InputEmpVariables.Add(ME_BufferConstants.inSfcRef, new Tuple<string, Type>(ME_BufferResources.sfcRef, Type.GetType("System.String")));
quickCompleteMetaData.InputEmpVariables.Add(ME_BufferConstants.inOperationRef, new Tuple<string, Type>(ME_BufferResources.operationRef, Type.GetType("System.String")));
quickCompleteMetaData.InputEmpVariables.Add(ME_BufferConstants.inResourceRef, new Tuple<string, Type>(ME_BufferResources.resourceRef, Type.GetType("System.String")));

//EMP Output Variables (= output variables mapped against the web service response)
quickCompleteMetaData.OutputEmpVariables.Add(ME_BufferConstants.outStepId, new Tuple<string, Type>(ME_BufferResources.nextOperation, Type.GetType("System.String")));
```

Figure 38 Add EMP Metadata

## 7.1.2 Service Call

Within the actual implementation you have to call the additional service while using the EMP variables. Here is an example:

```
//call further destination if required
if (callDestination == true)
{
    //prepare the input variables for the ME service call
    Dictionary<string, object> destinationInput = new Dictionary<string, object>();
    destinationInput.Add(this.empMetaData.Methods[QuickCompleteMethodId].InputEmpVariables.ElementAt(0).Key, "SFCBO:" + inSite + "," + inSFC);
    destinationInput.Add(this.empMetaData.Methods[QuickCompleteMethodId].InputEmpVariables.ElementAt(1).Key, "OperationBO:" + inSite + "," + inOperation + ",#");
    destinationInput.Add(this.empMetaData.Methods[QuickCompleteMethodId].InputEmpVariables.ElementAt(2).Key, "ResourceBO:" + inSite + "," + inResource);

    //call ME service
    Dictionary<string, object> destinationResult = currentDestinationCallback.callDestination(destinationInput, true);
}
```

Figure 39 EMP Service Call

As you can see you perform the call while using the EMP variables. In this example they had to be modified in order to be able to call the service.

## 7.1.3 Notification Maintenance

Refer to chapter Notification Maintenance to see how notification mappings can be done when EMP variables are involved.

The rest of the steps are equal to the steps described in chapter EMP Example Implementation (Method with EMP Variables).

# 8 ME_Buffer Project reference implementation

## 8.1 Scenario

With the ME_Buffer project it is possible to buffer routing information from SAP Manufacturing Execution (ME) on PCo level in order to support high speed production and avoid time consuming communication with ME.

Therefore the routing information is requested from ME and stored within PCo when the production starts. From that point in time the ME_Buffer within PCo is the main source for dealing with routing information. For completing routing steps within ME the ME_Buffer implementation uses the quick complete web service of ME. Therefore the quick complete implementation within ME_Buffer makes use of EMP variables to call the web service.

## 8.2 Implementation

You don't have to do further implementation when you want to use the ME_Buffer project to buffer ME routings on PCo level. All necessary methods are included.

### 8.2.1 Basic Assumptions

- The PCo buffering concept mainly supports the interaction with SAP ME. This does not exclude scenarios where other ME systems deliver buffer data. Below you will find the main xsd structure which is the basis of the buffer implementation. When a third party ME system respects this structure all buffer methods described in the following can be used.
- In the following you will read about updating the PCo buffer. The basic assumption is that in an update case the complete XML file will reach the buffer implementation. Supported update cases are:
  - Adding of new steps
  - Deletion of steps
  - Adding of parameters
  - Deletion of parameters
  - Change of parameter values

  This also means that even if a complete part of the ME routing is not known to the PCo buffer at the point in time where it was filled it could be added at a later point in time when PCo requests a buffer update.

- Within the following section only methods that can be imported and therefore can be used within a PCo server are explained in detail. The implementation of the ME_Buffer project relies on the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://www.example.org/test/"
targetNamespace="http://www.example.org/test/">
 <xs:element name="Rowset">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Row">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="outSfcDetails">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="rowList">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="sfcList">
              <xs:complexType>
               <xs:sequence>
                <xs:element name="sfc" maxOccurs="unbounded" minOccurs="0">
                 <xs:complexType>
                  <xs:sequence>
                   <xs:element name="sfcNumber" type="string"/>
                   <xs:element name="materialNumber" type="string"/>
                   <xs:element name="erpSalesOrderNumber" type="string"/>
                   <xs:element name="step" maxOccurs="unbounded" minOccurs="0">
                    <xs:complexType>
                     <xs:sequence>
                      <xs:element name="name" type="string"/>
                      <xs:element name="parameter">
                       <xs:complexType>
                        <xs:sequence>
                         <xs:element name="key" type="string"/>
                         <xs:element name="value" type="string"/>
                        </xs:sequence>
                       </xs:complexType>
                      </xs:element>
                     </xs:sequence>
                     <xs:attribute name="id" type="string"/>
                     <xs:attribute name="operation" type="string"/>
                     <xs:attribute name="resource" type="string"/>
                     <xs:attribute name="complete" type="string"/>
                    </xs:complexType>
                   </xs:element>
                   <xs:element ref="sfc" minOccurs="0" maxOccurs="unbounded"/>
                  </xs:sequence>
                 </xs:complexType>
                </xs:element>
               </xs:sequence>
              </xs:complexType>
             </xs:element>
```

```
            </xs:sequence>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

As you can see an XML file consists of multiple sfc tags which have the attributes "Workcenter" and "complete" to indicate whether the sfc has been completed. An sfc element consits of the element "sfcNumber", "materialNumber", erpSalesOrderNumber", "step" and "sfc" which means that an sfc element can have other sfc elements as children. The "step" element contains parameter information if applicable.

- Within a buffer agent ONE buffer (memory area) is the single source of truth for all buffer (routing) operations in PCo. All routing information is condensed within the above structure in one place.
- The PCo buffer operations do more than just reflecting existing ME services. There are methods within the ME_Buffer project that don't even exist in ME but can be useful to support a production process when using buffering.
- It depends on the PCo configuration how to make use of the buffer. It can be that operating on the buffer and the classical communication with ME could be used within one and the same production process. Use the buffer whenever it makes sense to speed up the production process.
- There are no deviations or conditional operations within the PCo buffer. Conditional routing are stored and managed by ME. It is assumed that the PCo buffer always provides the case that is due at a certain point in time. It works straight from one operation to the other one without conditions. If for any reasons conditional branches have to be introduced on operational level (in PCo) it is possible to manage them by configuring operational sequences in PCo for that. But in general PCo will not double ME routing logic. Especially the question how to come to the next operation when it depends on business conditions is not handled in PCo.

## 8.2.2 ME_Buffer methods

### 8.2.2.1 _AreSubAssembliesCompleted

Input parameters: inSfcNumber, inWorkCenterId

Output parameters: outSfcNumber, outIsReleasable

Starting from and sfcNumber and its workcenter (input) the method checks if all siblings of the particular sfc are completed. If yes the main sfc can be processed. This method is used if you want to avoid the start of the main sfc unless the depending sfcs haven't been completed.

### 8.2.2.2 _ClearBuffer

Input paramter: inSfcNumber

If a main sfc has been completed it can be deleted from the buffer. This can be done for a particular main sfc or for all sfcs if you don't specify an sfcNumber as input parameter.

### 8.2.2.3 _FillBuffer

Input parameters: inBufferXML, inWorkCenterId

Output parameter: outSfcIds

Fills the PCo buffer with an XML which is coming from the method's interface. You have to specify the workcenter for which you want to know the sfcs that have been created.

Example: Your main sfc has the workcenter "M" and it has two child sfcs with workcenter "A" and "B". You call "_FillBuffer" and pass the XML string and "B" as the workcenter. The method will return all sfcs that belong to workcenter B after the buffer has been filled.

### 8.2.2.4 _GetDependentSfc

Input parameters: inSfcNumber, inWorkCenterSource, inWorkCenterTarget

Output parameters: outSfcNumber

Starting from an sfc within the buffer structure this method returns a target sfc that has a relation to the start sfc..

Example: Let's assume the following structure

```
<sfc Workcenter="HMI_XTSM" complete="false">
  <sfcNumber>HMI_CHIP_REDS_BLL-10000001</sfcNumber>
....
        <sfc Workcenter="HMI_XTSB" complete="false">
          <sfcNumber>1000200001</sfcNumber>
....
        <sfc Workcenter="HMI_XTSA" complete="false">
          <sfcNumber>1000300001</sfcNumber>
....
```

If you call the method with inSfcNumber = 1000300001; inWorkCenterSource = HMI_XTSA, inWorkCenterTarget = HMI_XTSB you will get back 1000200001 as target sfcNumber.

### 8.2.2.5 _GetSalesOrderNumber

Input parameters: inSfcNumber, inWorkCenterId

Output parameter: salesOrderNumber

Returns the sales order number that is assigned to the main sfc. You can query the method with any valid sfc/workcenter combination within an sfc tree.

Example: Let's assume the following structure:

```
<sfc Workcenter="HMI_XTSM" complete="false">
  <sfcNumber>HMI_CHIP_REDS_BLL-10000001</sfcNumber>
  <erpSalesOrderNumber>227</erpSalesOrderNumber>
....
        <sfc Workcenter="HMI_XTSB" complete="false">
          <sfcNumber>1000200001</sfcNumber>
....
        <sfc Workcenter="HMI_XTSA" complete="false">
          <sfcNumber>1000300001</sfcNumber>
....
```

If you call the method with inSfcNumber = 1000200001, inWorkCenterId = HMI_XTSB you will get back the sales order number 227.

The same result would be returned if the call the method with inSfcNumber = 1000300001, inWorkCenterId = HMI_XTSA or with inSfcNumber = HMI_CHIP_REDS_BLL-10000001, inWorkCenterId = HMI_XTSM.

### 8.2.2.6 _QueryBuffer

Input parameter: inSfcNumber

Output parameter: outBufferXML

A low sophisticated method that displays the current content of the PCo buffer. This method is mainly used for overview reasons from UA Expert or test dialog of PCo management console.

If you specify an sfc number as input you will get back the section of the buffer for this particular sfc. Otherwise the whole buffer content is returned. The content is returned in a special text format for better readability.

### 8.2.2.7 _QuickComplete

Input parameters: inSite, inSfcNumber, inOperation, inResource, inCallDestination

Output parameter: outNextOperation

EMP input variables: inSfcRef, inOperationref, inResourceref

EMP output variables: outStepId

This is the only method of the ME_Buffer project with EMP variables. The EMP variables are needed because this method calls an ME service for quick completion of an sfc once a production has been done for the particular sfc.

Assume the following structure:

```
<sfc Workcenter="HMI_XTSA" complete="false">
  <sfcNumber>1000300001</sfcNumber>
  <materialNumber>HMI_CHIP_CONFIG</materialNumber
  <step id="0030" operation="2030" resource="HMI_A2" complete="false">
        <name>2030</name>
  </step>
</sfc>
```

To complete the step for the operation 2030 you have to call the method in the following way:

inSite: 1000, inSfcNumber: 1000300001, inOperation: 2030, inResource: HMI_A2.

With the parameter "inCallDestination" you can decide of the quick complete service in ME is called or not. If you don't set the parameter no service call to ME takes place but the internal buffer is updated.

Given the parameters from above will result in the completion of step 0030. Since this is the only step of the above sfc not only the step but also the whole sfc element will be set to complete.

### 8.2.2.8 _UpdateBuffer

Input parameters: inSfcNumber, inBufferXML

Output parameter: outNextOperation

This method assumes that a complete structure for one particular sfc is used to update the respective section within the buffer. Refer to Basic Assumptions to find out which update operations are supported.

The method returns the next operation that is due while it respects the new situation after the update.

Example:

Assume the following structure:

```
<sfc Workcenter="HMI_XTSM" complete="false">
    <sfcNumber>HMI_CHIP_REDS_BLL-10000001</sfcNumber>
    <materialNumber>HMI_CHIP_REDS_BLL</materialNumber>
    <erpSalesOrderNumber>227</erpSalesOrderNumber>
    <step id="0010" operation="1010" resource="HMI_M1" complete="true">
        <name>1010</name>
    </step>
    <step id="0020" operation="1020" resource="HMI_M2" complete="true">
        <name>1020</name>
        <parameter>
            <key>ROBOT_JOB</key>
            <value>10201</value>
        </parameter>
    </step>
    <step id="0030" operation="1030" resource="HMI_M3" complete="true">
        <name>1030</name>
    </step>
    <step id="0040" operation="1040" resource="HMI_M4" complete="false">
        <name>1040</name>
    </step>
```

Figure 40 Buffer section before update

You can see an sfc with some subsequent steps. According to the completeness of the steps the next due operation is operation 1040. Before executing operation 1040 the update method is called and changes the buffer in the following way:

```
<sfc Workcenter="HMI_XTSM" complete="false">
    <sfcNumber>HMI_CHIP_REDS_BLL-10000001</sfcNumber>
    <materialNumber>HMI_CHIP_REDS_BLL</materialNumber>
    <erpSalesOrderNumber>227</erpSalesOrderNumber>
    <step id="0010" operation="1010" resource="HMI_M1" complete="true">
        <name>1010</name>
    </step>
    <step id="0020" operation="1020" resource="HMI_M2" complete="true">
        <name>1020</name>
        <parameter>
            <key>ROBOT_JOB</key>
            <value>10201</value>
        </parameter>
    </step>
    <step id="0030" operation="1030" resource="HMI_M3" complete="true">
        <name>1030</name>
    </step>
    <step id="0045" operation="1045" resource="HMI_M4" complete="false">
        <name>1045</name>
    </step>
```

Figure 41 Buffer section after update

The _UpdateBuffer method would return operation 1045 as the next possible step.

The idea behind the usage of this method is to support asynchronous complete operations with ME.

Imagine a routing in ME that looks as follows:

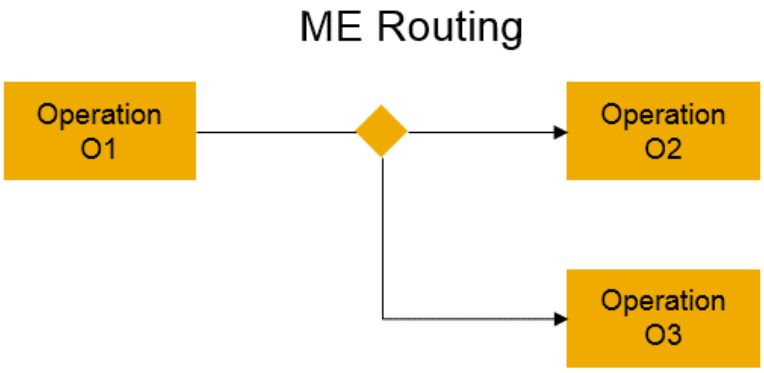



Figure 42 Possible ME routing

You see that after operation O1 ME has to take a decision either to choose O2 or O3 as the next step. The initial status of the PCo buffer would look as follows:

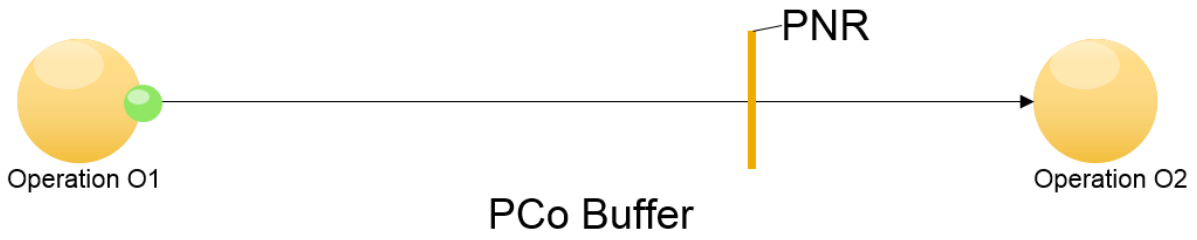



Figure 43 PCo buffer status before update

The little bullet within figure 24 should be the product that has passed operation O1 currently. In this example operation O2 is the default case. It could also be that Operation O2 is not known at this point in time and is therefore not in PCo buffer, yet.

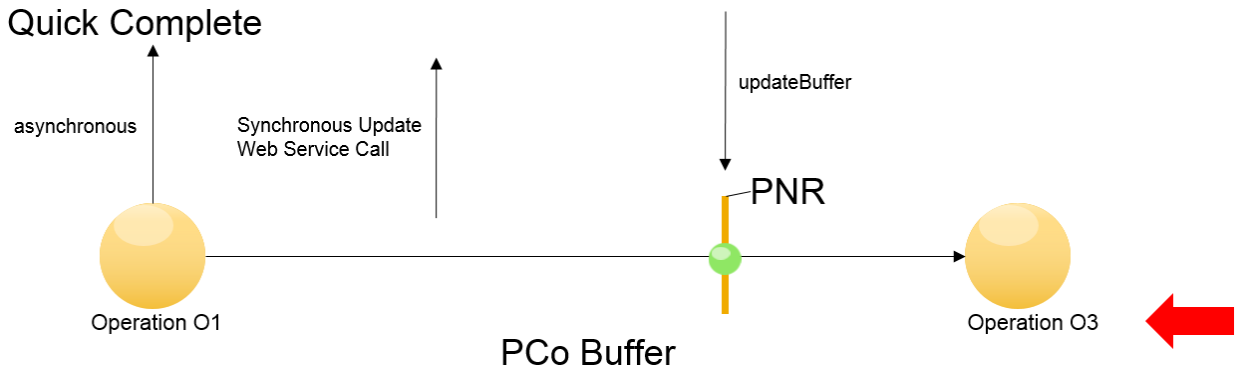



Figure 44 PCo buffer status after update

In this example it is assumed that PCo calls a QuickComplete service in ME to notify ME that the operation O1 has been completed. This call is done in an asynchronous way to save time.

Another assumption in this case is that there is a point of no return (PNR) within the process where the product stops of not update has taken place up to this point in time.

In Figure 25 you can see that on the way between O1 and PNR a synchronous service call has taken place. This service call is the update request actually. It is foreseen within the PCo configuration because when using the buffer you decide from the shop floor when to communicate with ME. In this picture the web service response takes longer. Before the response comes back the product has reached PNR, already. After PNR it would not be possible to change the route of the product within the process anymore, so it has to stop there and wait for the response of the update call.

When the update call has reached PCo the buffer is updated. You can see in figure 25 that O3 is the new operation instead of O2. Since the product has waited at PNR it is now possible to send it to O3.

In this example a PNR is assumed but in general it depends on the configurator how to deal with update calls. For the case that there is no PNR the update service call would also be possible if it is clear that the communication towards ME takes less time than the transport from O1 to O2/O3. But, as said, you are free to configure the call whenever you want it to happen.

## 8.2.2.9 _UseSetPointGroup

Input Parameters: inSite, inSfcNumber, inResource, inOperation

Output Parameter: outParameterValue

This method returns the parameter value for a parameter of a particular sfc,resource,oparation combination.

Example:

Assume the following structure:

```
<sfc Workcenter="HMI_XTSB" complete="false">
    <sfcNumber>1000200001</sfcNumber>
    <materialNumber>HMI_COVER_PRINTED</materialNumber>
    <step id="0010" operation="3010" resource="HMI_B1" complete="false">
        <name>3010</name>
        <parameter>
            <key>SHELL_TEXT</key>
            <value>nice little text</value>
        </parameter>
    </step>
```

Figure 45 Parameters within buffer

If you call the method with inSite = 1000, inSfc = 1000200001, inResource = HMI_B1 and inOperation = 3010 it will return "nice little text" as outParameterValue.

**www.sap.com/contactsap**

**Material Number**