Using ABAP OO methods in Workflow Tasks

In the previous blogs of this series, I talked about why we want to use ABAP OO with workflow and how to make an ABAP Class workflow-ready. This blog deals with using a workflow-ready ABAP Class directly in the simplest type of workflow - a single-step task.

How do I know if I'm ready for this?

If you haven't read the first two blogs yet, do that now.
1. Why use ABAP OO with Workflow?
2. Getting started with ABAP OO for Workflow ... using the IF_WORKFLOW interface

If you want to try the exercise in this blog in your own system, you will need an ABAP Class with the IF_WORKFLOW interface, and a system/client (SAPNetWeaver 6.20 minimum) with the workflow environment activated. If you aren't sure if your workflow environment is activated, check transaction SWU3 in your system to quickly view and activate the workflow environment in your system. Make sure both the definition and runtime environments are activated, so you can create a task and then test it.

As mentioned in the last blog, this blog won't teach you ABAP OO. It won't teach you workflow either, but it should give you enough to get going even if you've never touched workflow before.

Note: I've based the examples on an ABAP Class representing a Plant as this is a well known business entity in most R/3 and ECC systems. However if you are not in a suitable system or just don't want to use a Plant example for whatever reason, you can use your own business entity. All you will need for this exercise is an entity with a key (preferably less than 32 characters in length) and some code to call a display user interface (transaction, function, whatever) for your entity.

How do I use an ABAP OO method in a Task?

The simplest type of workflow is a single-step task, i.e. a piece of work that does one thing only. In ABAP OO terms, a single-step task calls one method only to perform this piece of work. In future blogs we'll deal with multi-step workflows, but as multi-step workflows essentially organise single-step tasks, single-step tasks are a good place to start when building workflows.

In the last blog I mentioned that you cannot call any method inherited from the IF_WORKFLOW interface directly. So to start we need an additional method in the ABAP Class example we have been creating.

The simplest type of method for this purpose is a Public, Instance method with no parameters. A classic example of this is a DISPLAY method. Now of course the code in the method will vary depending on what you are displaying. Here is the code needed to display a Plant in a typical R/3 or ECC system. method DISPLAY .

```
data: ls_vt001w type v_t001w.
CLEAR ls_vT001W.
ls_VT001W-MANDT = SY-MANDT.
ls_VT001W-WERKS = me->PLANT.
CALL FUNCTION 'VIEW_MAINTENANCE_SINGLE_ENTRY'
EXPORTING
ACTION = 'SHOW'
VIEW_NAME = 'V_T001W'
CHANGING
ENTRY = ls_vT001W.
endmethod.
```

Have you noticed that I have used an attribute of the class called PLANT? This is the key (Public, Instance) attribute of my class (data type is WERKS). Make sure you add this attribute to the ABAP Class before you activate your new method.

How do you set up the value of Plant? Well of course to create an instance of an ABAP Class in the first place we need a CONSTRUCTOR method, and it's in this method that we fill the attribute plant.

```
 Tip! Don't forget to activate and test your class in transaction SE24 before continuing.
```

Now that we can instantiate the ABAP Class and have a method, DISPLAY, that we can call from workflow, we are ready to include the method in a single-step task.

To create a single-step task, go to transaction PFTC_INS. Choose the task type "TS" (Standard Task, i.e. a single-step task) and press the Create button.
On the second screen the real work begins. On the Basic Data tab, give your task an abbreviation, name, and work item text - these are all free text fields so enter whatever you like. Then choose Object Category "ABAP Class", enter your ABAP Class as the Object Type, and your DISPLAY method as Method.

Note: If you are in a 6.20 system and can't see the Object Category choice "ABAP Class", then you first need to execute report SWF_CATID to enable "ABAP Classes" as an available object category. It's already done for you in releases 6.40 and higher.

Move across to the Container tab. This is the workflow data storage area - think of it as the equivalent of a data declaration area in a program. You'll notice that workflow has automatically added a container element (equivalent of a data variable in a program) to hold an instance of your class. It's even marked it as available for import/export from the task to other workflow objects such as multi-step tasks and events. You can drill down on the container element if you want to see more.

Save your task, and then there's two things we still need to do before you can test it.

1. You need to clarify how the system will know when this task is complete. For a display task this is easy... running the method is all we have to do, nothing has to be changed on a database or synchronised with another system. So all that's needed is to check the flag Synchronous object method below the Method field on the Basic Data tab, and save the task again.
2. You need to specify who is allowed to execute this task. For simplicity's sake, you can let anyone execute the task by making the task a General Task. Follow the menu path Additional Data - Agent Assignment - Maintain. Place your cursor on the task name, and press the Attributes... button. Select the General Task radio button, press enter and Save your task again.

Finally make a note of your task number and go to transaction SWUS to test your task. Type in your task id ("TS" followed by the task number) and press Enter. You'll see the import container elements listed below. You'll need to fill in which plant we want to display. Select the container element _WI_Object_Id and the local persistent object reference to your class is displayed below. Type a valid plant id into the field INSTID, and press Enter.

Now simply press the Execute button in the top left hand corner of the screen and you should see the plant details displayed. That's it!

**How do I explicitly instantiate an instance in a Task?**

Last time I mentioned that you can't call the special method CONSTRUCTOR directly from a task. Usually that's not a problem for workflow as we would use an event to implicitly create the instance and pass that in as we started the workflow.

I'll talk about events in the next blog, but there are occasions when you do want to explicitly create an instance in a workflow step. For example it can be more convenient to instantiate an object based on a flat value attribute of another object, rather than risk potentially recursive code by making the attribute an object instance itself. Or you might simply want to take one of the standard workflow tracked fields, such as the user id that executed a previous workflow step, and get more details, such as the user's name.

This is also a good chance to look at:

- Using method parameters with workflow
- Using background methods with workflow

Start by creating a static, public method in your class - call it CREATEINSTANCE. The purpose of this method is to import a plant id and export an instance of the plant class. Make sure you create a suitable signature (import/export parameters) for your method.

Inside the class all you need to do is to call the create object command to create the instance. Don't worry about error handling at this stage ... we'll cover that in a later blog. METHOD createinstance.

TRY.
CREATE OBJECT eo_plantinst EXPORTING
plant = iv_plant.
CATCH cx_bo_error.
ENDTRY.
ENDMETHOD.

Note: Don't forget to syntax check, activate, and test your method works before using it in workflow.

Create a task with your CREATEINSTANCE method just as you did with the DISPLAY method. You'll be asked if you want to "Transfer missing elements from the object method?" - Answer "yes" and you'll notice that the import and export parameters of your method are automatically added to the task container.

This sort of task is a technical step that is called in background by the workflow system, rather than being executed by a user, so instead of making the task a General Task, mark task as a Background processing in the Execution section of the task (below the method on the Basic Data tab). Don't forget to check the flag Synchronous object method and save again as we did for the DISPLAY task.

Test your new task using SWUS. Fill the import parameter IV_PLANT with a plant id, e.g. 1000, then press the Execute function. Because the task is executed in background we need to look at the workflow log to see if it worked. Use the Workflow log button display the work item. Then use menu path Extras > Container to see the end result. You should see the container element EO_PLANTINST has an valid reference to the plant class.

**What about functional methods?**
Functional methods (i.e. a method with a RETURNING parameter) can be used as above, or you can also use them in workflow bindings (think "parameter passing" between different parts of a multi-step workflow). However at this stage, you only have a single-step task to work with so I'm going to leave how to use functional methods in a binding until later in this blog series by which time I'll have expanded our workflow example a little further.

How do I call the task from an application?

Although it's possible to call a task directly, e.g. using standard workflow API functions such as SAP_WAPI_START_WORKFLOW, that's not usually the best way to tackle this very common scenario. In general, we call a task by raising an event which then use to trigger the task. So the next blog will be on using ABAP OO events as workflow events.