VNSG Magazine
Workflow tips en tricks
Editie maart 2007

ABAPOO en SAP Workflow

avelon

avelon

ABAP OO en SAP Workflows

Het debuggen van SAP Workflows is soms net wat lastiger dan het debuggen van “reguliere” ABAP’s. De uitgevoerde code wordt met behulp van de workflow engine uitgevoerd waardoor het in bepaalde gevallen lastiger is om detailtests uit te voeren of problemen op te lossen. Dit geldt met name bij workflows die gestart worden door triggering events en taken die in de achtergrond worden uitgevoerd.

Uiteraard zijn er voldoende mogelijkheden om dit probleem op te vangen. Deze mogelijkheden zijn soms enigszins verstopt in de verschillende menupaden en zelfs binnen de test-transacties.

De debug- en tracemogelijkheden binnen SAP Workflow worden toegelicht in de lange beschrijving. Deze beschrijving is terug te vinden via http://www.vnsg.nl/magazine/ -> Tips & tricks.

Maart 2007, Sander van der Wijngaart, Avelon workflow tips & tricks

© Copyright 2007 Avelon BV

Om het gebruik van ABAP OO binnen SAP Workflow te bestuderen zijn er een paar uitstekende blogs van Jocelyn Dart (SAP Australia) via SDN beschikbaar gesteld. Deze blogs zijn te benaderen via
https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/u/4075

Om een compleet beeld te schetsen zijn hieronder de op dit moment beschikbare tutorials opgenomen:

Hoofdstuk 1: Waarom Abap OO gebruiken binnen SAP workflow?
Hoofdstuk 2: Kennismaking van het gebruik van ABAP OO binnen SAP workflow
Hoofdstuk 3: Gebruik van ABAP OO methoden in workflow taken
Hoofdstuk 4: Gebruik van ABAP OO events binnen SAP workflow
Hoofdstuk 5: Gebruik van ABAP OO attributen in taken en workflows
Hoofdstuk 6: Gebruik van functionele methoden in taken en workflows

© Copyright 2007 Avelon BV

1. Why use ABAP OO with Workflow?

Jocelyn Dart
Business Card
Company: **SAP Australia**
Posted on Jun. 19, 2006 11:23 PM in ABAP, Business Process Management

This is the start of a series on how to use ABAP OO with workflow.

1.1.1.1 How do I know this is relevant to me?

ABAP OO for workflow is available on any SAP NetWeaver platform release 6.20 or above (ABAP stack of course). That includes R/3 4.7, ECC, and the latest versions of SCM, SRM, CRM, xRPM, etc. P.S. There are a couple of minor restrictions in 6.20 that are resolved in 6.40 - but we'll deal with them later in the series.

1.1.1.2 What's the problem with BOR (Business Object Repository)?

A little SAP/Workflow history here... SAP Business Workflow came into being in R/3 3.0C. Now for those who can remember back that far, there certainly wasn't any such thing as ABAP OO back then (and it wouldn't be until R/3 4.6C that ABAP OO was available in sufficient depth to support workflow handling). So the Business Object Repository was created as an intermediate, very approximate, but at least object-oriented, solution for SAP Business Workflow. Because it was never intended to be the final solution, there was a limit to how far BOR was going to go. Consequently BOR suffers from:

A very old ABAP editor and relatively feature-poor support tools

Awkward and non-obvious macro coding necessitating workflow developers to have special skills above and beyond normal ABAPers or workflow creators

Limitations for instantiation, inheritance, generic handling, and calculation of attributes

1.1.1.3 How does ABAP OO solve these problems?

Well for starters, ABAP OO is now a fully-fledged object-oriented language with comprehensive feature-rich support tools. ABAP OO also provides:

Functional methods for calculation of attributes

Proper instantiation and garbage collection

Complex inheritance options

High reusability for true generic handling of objects

1.1.1.4 What other reasons are there for using ABAP OO with workflow?

I've saved the best for last because ABAP OO for workflow means: ***Any ABAP developer who can code ABAP OO can (after reading this series) code for workflow.*** So now project managers and workflow creators can call on any ABAP programmer to assist with custom development for workflow - you won't need a specially-trained workflow code-cutter any more.

1.1.1.5 Is there anything I still need to use BOR for?

There are a few areas where you might still want to use BOR objects - purely for convenience reasons.

The SELFITEM object for sending mails. This is still useful as everything, including the "Send Mail" step type, is pre-built for you. So it's probably not worth creating an ABAP OO equivalent.

© Copyright 2007 Avelon BV

The DECISION object - for user decisions. There's a lot of special functionality around this object for simple question/answer decisions - especially if you are using the UWL.

Generic Object Services - a lot of transactions still link to BOR objects - so it can be worth referencing the BOR object to keep the Workflow Overview feature going.

BOR events from standard transactions - a lot of transactions that have inbuilt workflow coding that raises BOR events - so it's still handy to reference BOR objects in your ABAP OO classes for these scenarios too.

And of course there's a fair bit of business content in provided BOR objects you might want to reuse.

1.1.1.6 Coming Attractions...

Over the coming series we'll be discussing

1. Getting started... using the IF_WORKFLOW interface
2. Using ABAP OO methods in workflow tasks
3. Raising ABAP OO events for workflow
4. Using ABAP OO attributes in workflows and tasks
5. Using functional methods in workflows and tasks
6. Referencing BOR objects in ABAP OO classes

So there'll soon be plenty of help to get you going. In the meantime, make sure you get involved in the Business Process Management forum.

Jocelyn Dart co-authored "Practical Workflow for SAP", and is currently supporting Workflow, Guided Procedures, MDM, xRPM, and xPD for Australia/New Zealand customers.

© Copyright 2007 Avelon BV

2 Getting started with ABAP OO for Workflow

[Jocelyn Dart]
[Business Card]
Company: **SAP Australia**
Posted on Jun. 28, 2006 06:01 AM in [ABAP], [Business Process Management]

The basic building block of all ABAP OO for Workflow classes is the IF_WORKFLOW interface. Unless you use the IF_WORKFLOW interface in your classes, the workflow system will not recognise your class as suitable for creating workflow tasks, events, accessing attributes, etc. The class could be a new or existing class.

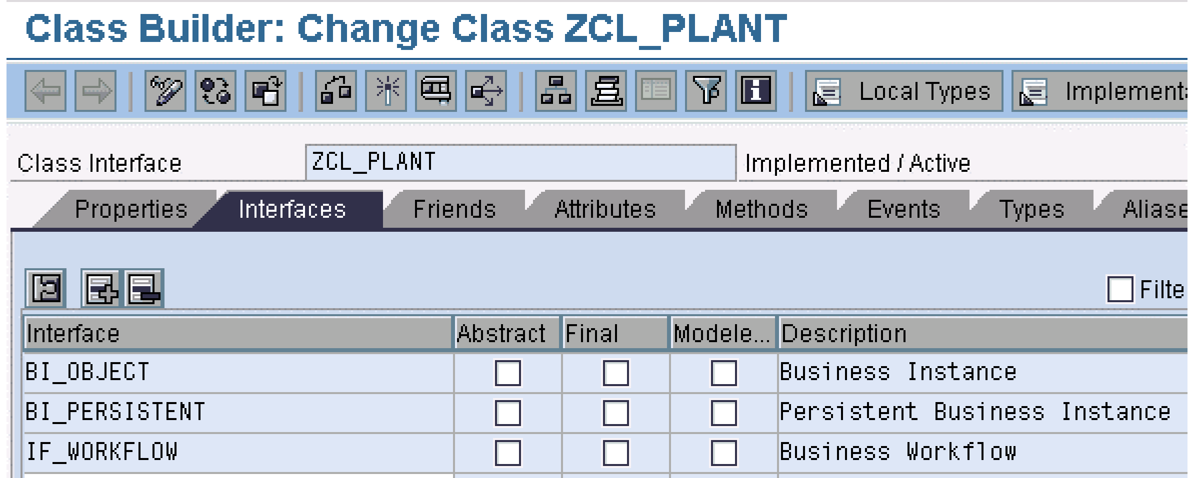
Note: This applies even to utility classes that implement static methods such as calculating deadlines based on factory calendars, de-duplicating a list of agents, or an empty background method for forcing a switch to the workflow system user.

1.1.1.7 What should I know before reading further?

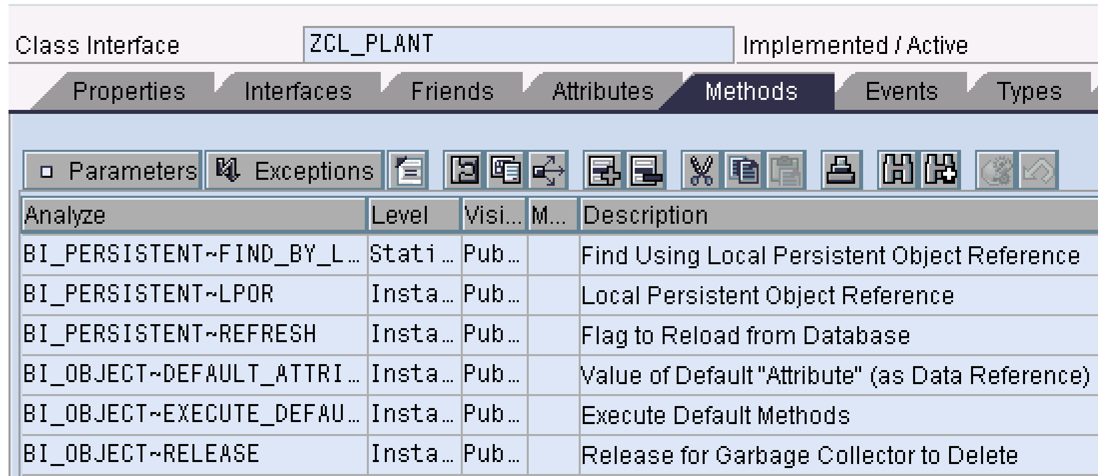
If you've never created an ABAP OO Class before you won't learn how to do that here. So you should have a look at the ABAP Development blogs and tutorials first, or read an ABAP Objects book.

1.1.1.8 Adding the IF_WORKFLOW interface to an ABAP Class

Attaching the IF_WORKFLOW interface to an ABAP Class is simple. In the Class Builder (transaction SE24), go to the Interfaces tab and add the IF_WORKFLOW interface. As soon as the interface is added, two sub-interfaces appear: BI_OBJECT and BI_PERSISTENT.



Move across to the Methods tab and you will see some methods of these interfaces have been automatically inherited to the ABAP Class.



1.1.1.9 What's the absolute minimum I need to implement from the IF_WORKFLOW interface for a utility class?

Open each of the methods inherited from the IF_WORKFLOW interface, and activate the empty source code, then activate the ABAP Class itself. No, really - that's it!

1.1.1.10 What's the absolute minimum I need to implement from the IF_WORKFLOW interface for a class representing a business entity?

Open each of the methods inherited from the IF_WORKFLOW interface, and activate the empty source code, then activate the ABAP Class itself. That's the easy bit, the next bit is only fractionally harder, but requires a little explanation.

Workflow uses the IF_WORKFLOW interface to generically and efficiently handle all references to ABAP OO for Workflow classes. There is a special part of this interface called the **Local Persistent Object Reference**, which is simply a way of converting from a generic reference to any object instance used by the workflow system to the specific instance of your ABAP Class and vice versa.

The instance is made up of three fields:

CATID - A two character code indicating the type of object. This is always "CL" for ABAP Classes. I'll explain about other values in a future blog on referencing BOR objects.

TYPEID - This holds the technical name of the ABAP Class.

INSTID - This is a 32 character field holding the unique instance id.

So what do you put in the **INSTID** field? Unless your ABAP Class is a utility class, usually the ABAP Class would represent a business entity of some kind. In the screenshots in this blog I've shown an example of a class that represents a Plant. To uniquely identify a plant I would use the 4 character Plant number, so for this class the plant number is what I would place in the INSTID field.

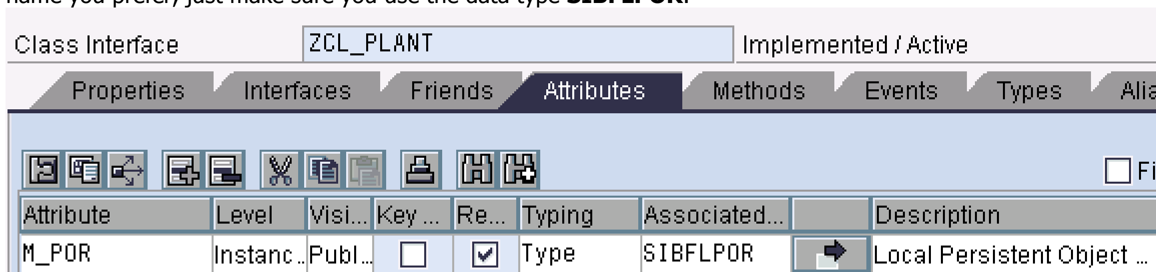
© Copyright 2007 Avelon BV

Similarly if I had a class representing a Sales Order, I would use the Order number to fill the INSTID field. If I had a class representing a Customer I would use the Customer Number. If I had a class representing a financial document I would concatenate the company code, document number, and fiscal year and use that as the INSTID value.

Note: It's a good idea (although not essential) to mark the attribute you use to fill the INSTID value as the "key attribute" in the Class Attributes section. This helps other developers to quickly see which attribute uniquely identifies the instance without having to drill down into method code.

What happens if the class represents something that has a unique key longer than 32 characters? In this case you would use a Globally Unique Id (GUID) to represent your unique key, and then relate the GUID back to the real key. A GUID is unique throughout your system - for this reason a lot of the latest SAP solutions use GUIDs rather than meaningful ids as technical ids. To create the GUID, you simply call function module GUID_CREATE. To relate the GUID back to the real key, you could either cross-reference the GUID in a separate custom table, or use an append structure to extend an existing table to also hold a GUID reference.

It's convenient to hold the Local Persistent Object Reference value in a public, instance attribute of your class. Workflow won't use the attribute directly, but holding it as an attribute allows you to efficiently fill it once in the Constructor method, and can also be handy for debug and testing purposes. You can call the attribute whatever name you prefer, just make sure you use the data type **SIBFLPOR**.



To be able to reference a unique instance of a ABAP Class in workflow, you need to use the Local Persistent Object Reference in the following two methods inherited from the IF_WORKFLOW interface.

1.1.1.10.1 BI_PERSISTENT~FIND_BY_LPOR

This is a Static method that is used to convert from the Local Persistent Object Reference used by workflow to a unique instance of the ABAP Class. It's used any time workflow needs to implicitly instantiate the ABAP Class, e.g. when raising an event.

```
data: lv_plant type werks.

   move lpor-instid(4) to lv_plant.

   create object result TYPE ZCL_PLANT
      exporting plant = lv_plant.
```

1.1.1.10.2 BI_PERSISTENT~LPOR

This is an Instance method that converts from the current instance of the ABAP Class to the Local Persistent Object Reference used by workflow. It's used any time the workflow wants to pass a reference to the ABAP Class, e.g. when handling an event.

You can either fill in the Local Persistent Object Reference attribute here or preferably (as it is then filled once only per instance) in the Constructor method of the class.

© Copyright 2007 Avelon BV

```
* These 3 lines would normally be in the Constructor method
  me->m_por-CATID = 'CL'.
  me->m_por-TYPEID = 'ZCL_PLANT'.
  me->m_por-INSTID = me->plant.
* If the 3 lines above are in the Constructor method,
* the line below is all you need in the BI_PERSISTENT~LPOR  method
  result = me->m_por.
```

1.1.1.11 What else can I use the IF_WORKFLOW interface for?

The other methods inherited from the IF_WORKFLOW interface are optional, but are useful in certain scenarios. It's up to you to decide whether you want to use them or not.

1.1.1.11.1 BI_PERSISTENT~REFRESH

This method is used when workflow asks to refresh the object, i.e. reload database and other stored values. You can use it to clear out and re-read existing attributes that were filled in your Constructor and Class Constructor methods, but most of the time it's ok to leave this method empty.

1.1.1.11.2 BI_OBJECT~DEFAULT_ATTRIBUTE_VALUE

This method is used in workflow inboxes, such as the Universal Worklist, and workflow administration tools to indicate which unique instance of the class is being referenced. By default, whatever value is in the INSTID field of the Local Persistent Object Reference will be shown. If you want to show a different attribute, just add the relevant code.

For example, if I have a class representing an instance of a Plant, then the key might be Plant Id, but I want to show the attribute "Description" instead, so I'll add the line:

```
 GET REFERENCE OF me->description INTO result.
```

1.1.1.11.3 BI_OBJECT~EXECUTE_DEFAULT_METHOD

This method is used in workflow inboxes, such as the Universal Worklist, and workflow administration tools, to control what happens when you drill down on the instance of the class being referenced. By default the drill down does nothing. If you want the drill down to call a specific method, typically a display method, then just add the relevant code.

For example, if I have a class representing an instance of a Plant, then I might want the default method to be the display method so I'll add the lines:

```
  TRY.
      CALL METHOD me->display.
    CATCH cx_root.
  ENDTRY.
```

1.1.1.11.4 BI_OBJECT~RELEASE

This method is used by the garbage collector when deleting an instance. You could use it to, for instance, write to a custom log, but most of the time it's ok to leave this method empty.

© Copyright 2007 Avelon BV

1.1.1.12 Can I call the IF_WORKFLOW methods directly from a workflow task?

Even though the IF_WORKFLOW methods are public methods, they are intended to only be used internally by the workflow engine, and so no, you can't call them directly from workflow (you'll get an error if you even try). So that's what the next blog will be about - how to use an ABAP Class in a workflow task.
Jocelyn Dart co-authored "Practical Workflow for SAP", and is currently supporting Workflow, Guided Procedures, MDM, xRPM, and xPD for Australia/New Zealand customers.

3 Using ABAP OO methods in Workflow Tasks

Jocelyn Dart
Business Card
Company: **SAP Australia**
Posted on Jul. 25, 2006 05:15 AM in ABAP, Business Process Management

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.

1.1.1.13 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.

1.1.1.14 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'
```

© Copyright 2007 Avelon BV

```
        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.

*Note: Make sure you use the **Create Constructor** button to add the constructor method to your class.*

As our ABAP Class represents a business entity, rather than a technical entity, an instance is meaningless without having this key attribute provided. So add an import parameter PLANT (data type WERKS) to the CONSTRUCTOR class so that the plant id can be provided during instantiation.

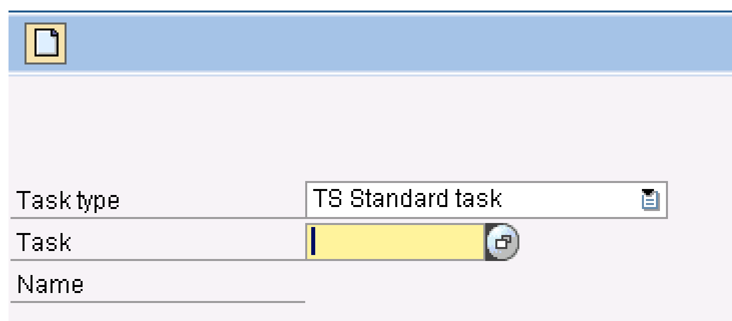
Here is some very simple code for the CONSTRUCTOR method.

```
METHOD constructor .
  me->plant = plant.
ENDMETHOD.
```

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

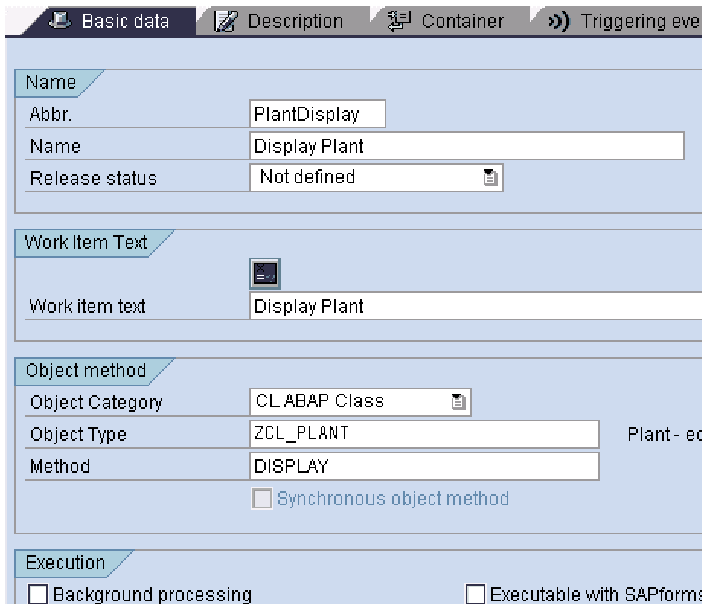
Now that we can instantiate the ABAP Class and have a method, DISPLAY, that we can call from workflow, we are ready to include the method in a single-step task.

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.



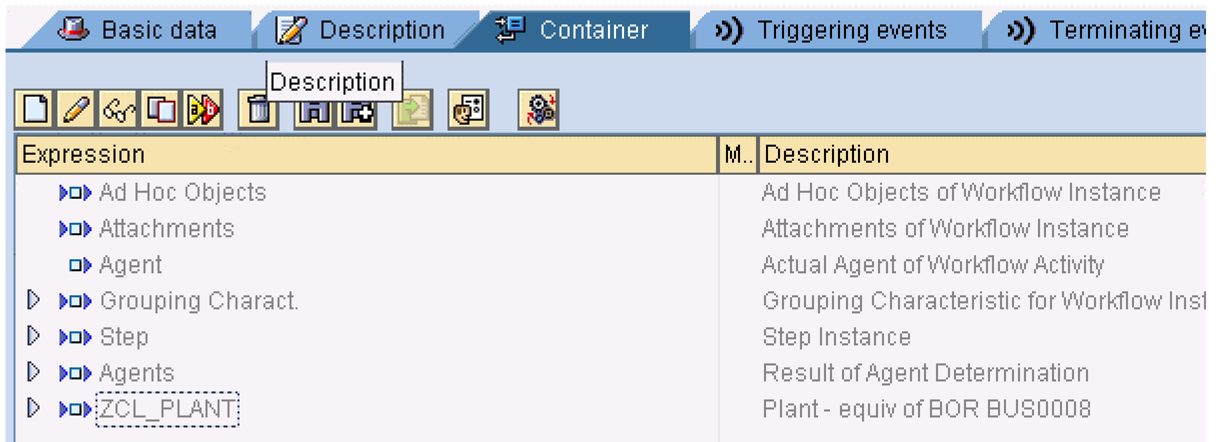
On the second screen the real work begins. On the Basic Data tab, give your task an abbreviation, name, and work item text - these are all free text fields so enter whatever you like. Then choose Object Category "ABAP Class", enter your ABAP Class as the Object Type, and your DISPLAY method as Method.

© Copyright 2007 Avelon BV



*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.

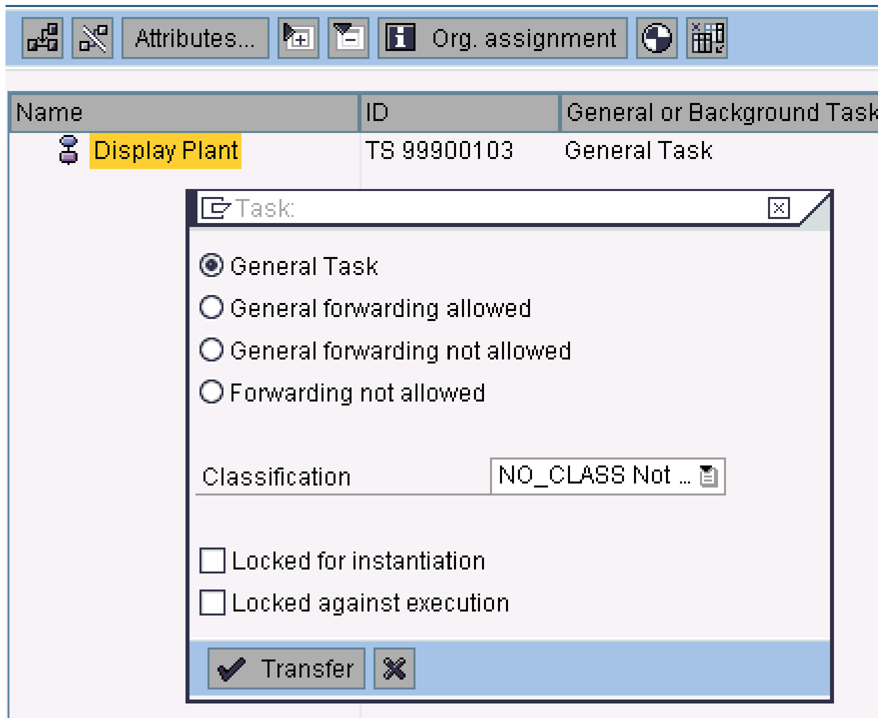


Save your task, and then there's two things we still need to do before you can test it.

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.

© Copyright 2007 Avelon BV

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.

Test Workflow

Refresh Organizational Environment

Workflow	TS99900103
Type	Standard task
Name	Display Plant
Validity	01.01.1900 To 3

Input Data | Ad Hoc Agents | DeadlineData

Test Data | Load | Save

Expression	M..	Values
Grouping Charact.		< No Instance >
Agents		< No Instance >
_Wi_Object_ID		ZCL_PLANT:1000

ObjectType	Object Type	InstanceID
CL	ZCL_PLANT	1000

Now simply press the Execute button in the top left hand corner of the screen and you should see the plant details

Display View "Plants": Details

Plant	1000
Name 1	Walldorf Plant
Name 2	

Detailed information

Language Key	DE	German
House number/street	Walker Street	
PO Box		
Postal Code	22299	
City	Hamburg	
Country Key	DE	Germany
Region	02	Hamburg

displayed. That's it!

© Copyright 2007 Avelon BV

1.1.1.15 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.

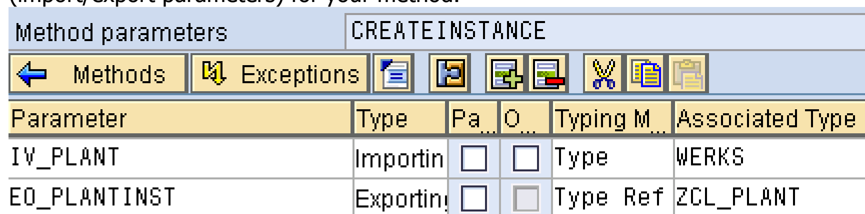
I'll talk about events in the next blog, but there are occasions when you do want to explicitly create an instance in a workflow step. For example it can be more convenient to instantiate an object based on a flat value attribute of another object, rather than risk potentially recursive code by making 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.



Method parameters CREATEINSTANCE

Methods Exceptions

Parameter	Type	Pa...	O...	Typing M...	Associated Type
IV_PLANT	Importin	☐	☐	Type	WERKS
EO_PLANTINST	Exportin	☐	☐	Type Ref	ZCL_PLANT

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.

© Copyright 2007 Avelon BV

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.

Work item information

ID	000000543644
Type	Background Step
Title	Instantiate Plant
Status	Completed
Priority	
Agent	

Deadline D

Current

Requeste

Latest

Description

Display Container Instance

Expression	Values
Ad Hoc Objects	< Not Set >
Attachments	< Not Set >
Agent	USWF-BATCH
Grouping Charact.	< No Instance >
Step	WORKINGWI:00000543€
Agents	< No Instance >
IV_PLANT	1000
EO_PLANTINST	ZCL_PLANT:1000
Local Persistent Object Reference	ZCL_PLANT:1000

1.1.1.16 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.

1.1.1.17 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.

Jocelyn Dart co-authored "Practical Workflow for SAP", and is currently supporting Workflow, Guided Procedures, MDM, xRPM, and xPD for Australia/New Zealand customers.

© Copyright 2007 Avelon BV

4 Raising ABAP OO events for workflow

Jocelyn Dart
Business Card
Company: **SAP Australia**
Posted on Jul. 27, 2006 11:32 PM in ABAP, Business Process Management

Subscribe
Print
Permalink

1.1.2

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

Easy ... did you read the first 3 blogs and create the examples mentioned in blogs 2 and 3? If not, here are the links...

1. Why use ABAP OO with Workflow?
2. Getting started with ABAP OO for Workflow ... using the IF_WORKFLOW interface
3. Using ABAP OO with Workflow Tasks

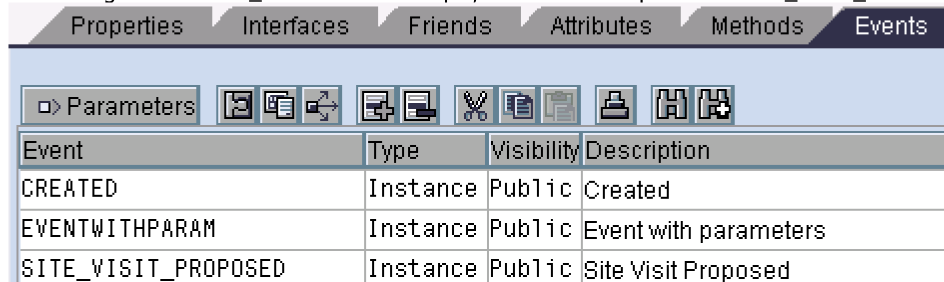
If you want to try the exercise in this blog in your own system, you will need an ABAP Class with the **IF_WORKFLOW** interface, and a system/client (SAPNetWeaver 6.20 minimum) with the workflow environment activated. You also need to have created the "Display Plant" workflow task mentioned in blog number 3.

Note: 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.

1.1.2.2 Defining an ABAP OO Event for Workflow

Defining an ABAP OO Event for Workflow is the same as defining any other event for ABAP OO. To start just go to transaction SE24, enter your class in edit mode, and go to the Events tab. Give your event a technical name and make sure it is a **Public** event.

Continuing with the ZCL_PLANT class example, here's an example event SITE_VISIT_PROPOSED.


Properties | Interfaces | Friends | Attributes | Methods | Events

Parameters

Event	Type	Visibility	Description
CREATED	Instance	Public	Created
EVENTWITHPARAM	Instance	Public	Event with parameters
SITE_VISIT_PROPOSED	Instance	Public	Site Visit Proposed

You can also define event parameters if you wish by selecting the **Parameters** button. By definition all event parameters are **Exporting** parameters, i.e. they are exported with the event when the event is raised.

Usually however you don't need to create any event parameters, as the standard Workflow event container (remember that's workflow's name for a data area) adds sufficient standard parameters, such as the user who raised the event, and a local persistent object reference to the ABAP OO class (of course you know all about that because you've read the previous blogs). Of course, you won't see the standard Workflow event container in your class - you'll only see it in the Workflow screens.

© Copyright 2007 Avelon BV

Here's an event parameter PROPOSED_VISIT_DATE for the event SITE_VISIT_PROPOSED. I've made the parameter **Optional** so that later we can look at how to raise an event with or without parameters.

Event parameters SITE_VISIT_PROPOSED

Events

Parameters	O...	Typing	Associated Type	Def...	Description
PROPOSED_VISIT_DATE	☑	Type	DATUM		Proposed date of visit

Tip! Don't forget to activate your class afterwards.

1.1.2.3 Using an ABAP OO Event in Workflow

For anyone who's using Business Object Events with Workflow, you use ABAP OO Events in exactly the same way. The only real difference is you specify category **ABAP Class** instead of Business Object. For those who haven't done much with events before, in Workflow an event can be used to:

Trigger a multi-step workflow

Terminate a multi-step workflow

Perform special functions on a multi-step workflow, e.g. re-evaluate agents of active work items, cancel and restart with same data

Trigger a single workflow task

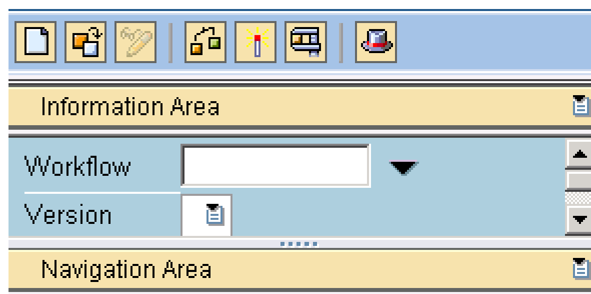
Terminate a single workflow task

*Note: The last option - terminate a single workflow task - is particularly useful for asynchronous scenarios. For example if you had a task (it could be background or dialog - i.e. involving a user) that kicked off a database update in background, and then let the raising of an event confirm that the change had been committed to the database. If you did this of course you would leave the **Synchronous object method** flag off in your background task (this flag was mentioned in blog number 3).*

Triggering a single workflow task with an event isn't a very common practice, simply because most workflows involve more than one step. So usually we want to trigger a multi-step workflow that represents a whole business process. So let's expand our example to a multi-step workflow triggered by an event.

Start by calling transaction SWDD, the Workflow Builder. Press the **Create New Workflow** button - the top-left hand button immediately under the text "Workflow Builder".

Workflow Builder - Create 'Unna



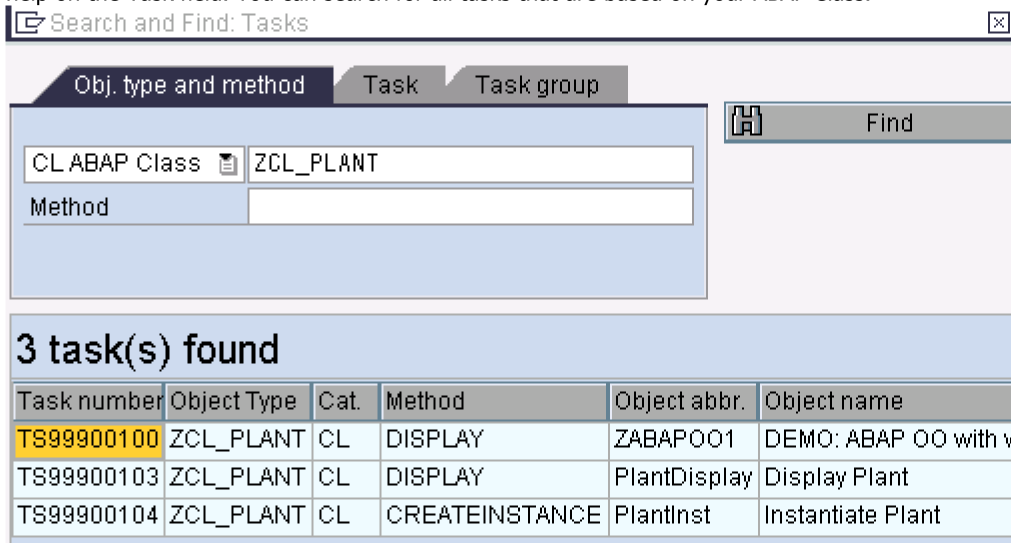
Then before you go any further, save your new empty workflow. You'll be asked to give it an abbreviation and a name - these are free text and are mainly used when searching for your workflow. Just like our task or an ABAP

© Copyright 2007 Avelon BV

Class, you'll need to add it to a change request or save it as a Local Object.

This going to be a very simple workflow for now, as we are just going to include our "Display Plant" task in it. So to do that, either single-click on the **Undefined** step in either the Navigation area or double-click on the Undefined step icon in the center Graphical Model editing area. Select the step type **Activity** and continue on to the next screen.

The very first field in the **Control** tab asks for a task id. Search for your "Display Plant" task by using the search help on the Task field. You can search for all tasks that are based on your ABAP Class.



Then just double-click on the appropriate row to bring the task id back into the activity type step Control tab.

Now press the <ENTER> key. Workflow will then propose that a reference to your ABAP Class should be added to the Workflow container and automatically creating a binding (mapping) from the reference in the Workflow container to the task container. All you have to do is press the green tick to confirm.

© Copyright 2007 Avelon BV

Define Container Elements and Binding

Selected task may require additional container elements in
the workflow container and additional binding definitions.
Check following proposal and confirm with 'Continue'.

Container: Workflow (new elements)

ZCL_PLANT	ZCL_PLANT	ZCL_PLANT

Tasks <=> Workflow

_WI_OBJECT_ID	←	&ZCL_PLANT&

All we need to do now is nominate who should perform this step of our business process. To keep it simple for our example, just make the workflow initiator, i.e. the person who started the process, the **Agent** of the step. In the Agents section, drop down the Expression field to choose the Workflow initiator.

Agents

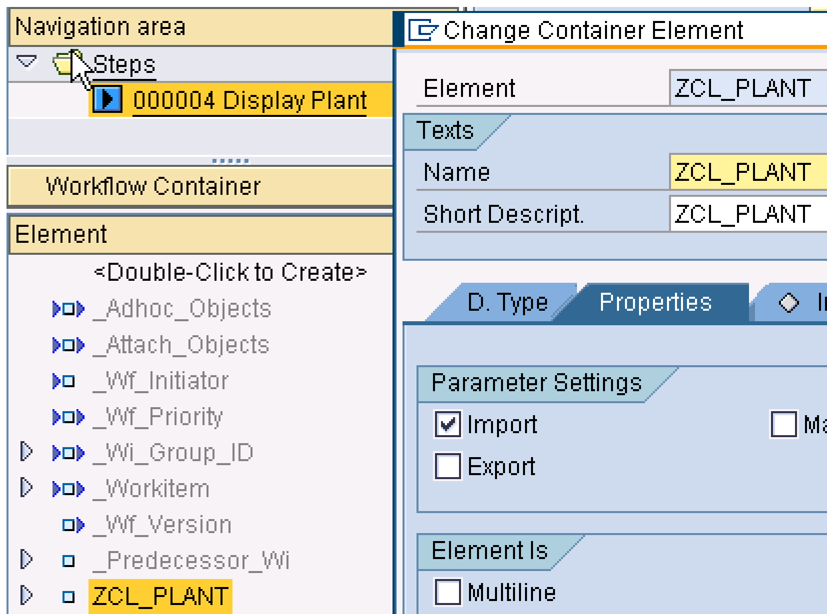
Expression &_WF_INITIATOR&

Note: Usually we want to calculate who's going to do the work using an agent determination rule but this is just an example after all.

Press the green check icon **Transfer and to graphic** to return to the graphical flowchart view. You can save your workflow again at this point if you wish.

We are going to start our workflow with an event and pass in our ZCL_PLANT reference from the event, so we need to allow the new ZCL_PLANT container element to be imported. On the left hand side, under the Navigation area, use the dropdown to swap the bottom left hand area to the **Workflow Container** display. Double-click on the new ZCL_PLANT container element, and on the **Properties** tab check the **Import** flag.

© Copyright 2007 Avelon BV



So having completed our mini how-to-create-a-workflow exercise we now want to add the SITE_VISIT_PROPOSED event as the triggering event of the workflow. At the top of the screen press the **Basic Data** hat icon. On the **Version-Independent** tab, choose the sub-tab **Start Events**. Add a row and enter category **CL**, then Object Type = your ABAP Class, and then Event of the object = SITE_VISIT_PROPOSED.

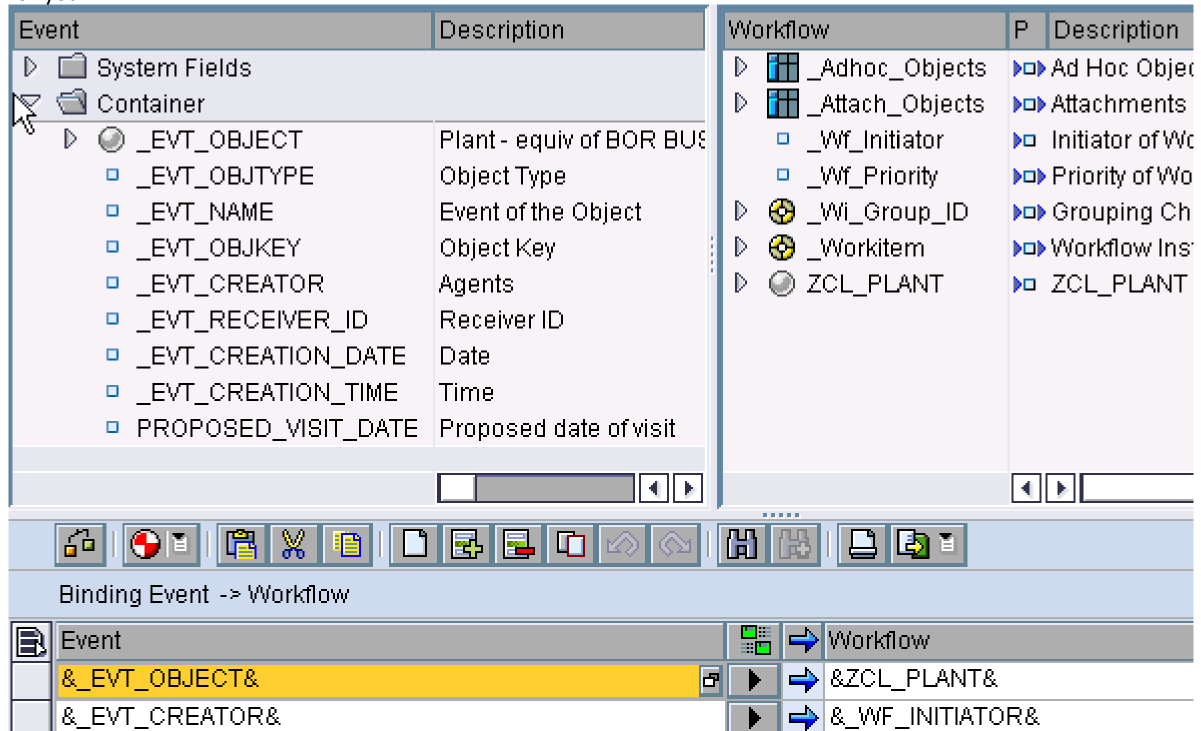
A...	B...	C...	Cate...	Object Type	Event of the object
			CL	ZCL_PLANT	SITE_VISIT_PROPOSED

Tip! It's a good idea to use the dropdown help for all of these fields to avoid typing errors.

You can see the event row starts with an A, B, and C columns. B stands for **Binding**. We need to create the binding (mapping) between the event container and the workflow container to pass across the instance of our ABAP Class and the person who raised the event. Just click on the Binding icon and the binding will be generated

© Copyright 2007 Avelon BV

for you.



Remember the A, B, and C columns? A stands for **Activate**. Click on the activate icon until the event activation icon turns green. Now use the back arrow icon to return to the graphical flowchart display and use the Activate (lit match) icon at the top of the screen to activate the workflow. Check the status of your workflow is **Active, Saved** in the **Information Area** in the top left hand window pane.

So now you have a workflow that's linked to a triggering event of your ABAP Class! If you want to test it, use transaction SWUE to generate the event, or just wait until you have completed the next section on raising events. You should see a new "Display Plant" work item added to the Workflow section of your SAP inbox (transaction SBWP).

1.1.2.4 Raising an ABAP OO Event for Workflow

Now of course we would not usually raise the event manually, normally an application - a transaction, a BAPI, a function module, a report, a WebDynpro, etc. - would actually trigger the event using code.

Raising Business Object events usually involved a traditional function module call - such as SAP_WAPI_CREATE_EVENT. Raising ABAP OO events, not unsurprisingly, uses an object-oriented approach - instead of a function module call, a method call is used to raise the event.

The provided SAP ABAP Class that handles the raising of workflow events is **CL_SWF_EVT_EVENT**. This class contains two Static, Public methods for raising ABAP OO events for Workflow:

RAISE - for raising an event immediately

RAISE_IN_UPDATE_TASK - for raising an event in the update task of a Logical Unit of Work

Here's some example code to raise our SITE_VISIT_PROPOSED event using the RAISE method - without any parameters.

```
DATA:
  ls_zcl_plant_key TYPE werks,

  lv_objtype        TYPE sibftypeid,
  lv_event          TYPE sibfevent,
  lv_objkey         TYPE sibfinstid.

lv_objtype = 'ZCL_PLANT'.
lv_event   = 'SITE_VISIT_PROPOSED'.

* Set up the LPOR instance id
ls_zcl_plant_key-werks = '1000'.
MOVE ls_zcl_plant_key TO lv_objkey.

* Raise the event
TRY.
    CALL METHOD cl_swf_evt_event=>raise
      EXPORTING
        im_objcateg         = cl_swf_evt_event=>mc_objcateg_cl
        im_objtype          = lv_objtype
        im_event            = lv_event
        im_objkey           = lv_objkey
*       IM_EVENT_CONTAINER =
        .
  CATCH cx_swf_evt_invalid_objtype .
  CATCH cx_swf_evt_invalid_event .
ENDTRY.

COMMIT WORK.
```

*Tip! Just as for Business Object events, the **COMMIT WORK** statement is crucial when raising a Workflow event - without this the event will not be raised.*

Check the event was raised correctly by turning on the event log (using transaction SWELS), and executing the event log. If your event was raised correctly you should have a new line in the event log that looks something like this:

Object Type	Event	Current Date	Time	Receiv...	Information	Handler/A
ZCL_PLANT	SITE_VISIT_PROPOSED	28.07.2006	04:13:44	WS999...		SWW_WI_

The green light icon shows that a workflow was started as a result of this event.

The above code is fine for most events that just use the standard Workflow event container and have no specific parameters. Of course if you want to pass specific parameters you need a little extra code to:

1. Instantiate an empty event container
2. Add your event parameter name/value pairs to the event container
3. Raise the event passing the prepared event container

Note: You don't need to worry about filling any of the standard event container parameters such as the event creator - Workflow will do that as part of raising the event.

```
DATA:
```

© Copyright 2007 Avelon BV

```
  ls_zcl_plant_key    TYPE werks,

  lv_objtype          TYPE sibftypeid,
  lv_event            TYPE sibfevent,
  lv_objkey           TYPE sibfinstid,

  lr_event_parameters TYPE REF TO if_swf_ifs_parameter_container,
  lv_param_name       TYPE swfdname,

  lv_visit_date       TYPE datum.

lv_objtype = 'ZCL_PLANT'.
lv_event   = 'SITE_VISIT_PROPOSED'.

* Set up the LPOR instance id
ls_zcl_plant_key-werks = '1000'.
MOVE ls_zcl_plant_key TO lv_objkey.

* Instantiate an empty event container
CALL METHOD cl_swf_evt_event=>get_event_container
  EXPORTING
    im_objcateg  = cl_swf_evt_event=>mc_objcateg_cl
    im_objtype   = lv_objtype
    im_event     = lv_event
  RECEIVING
    re_reference = lr_event_parameters.

* Set up the name/value pair to be added to the container
lv_param_name = 'PROPOSED_VISIT_DATE'.
lv_visit_date = sy-datum + 7.

* Add the name/value pair to the event conainer
TRY.
    CALL METHOD lr_event_parameters->set
      EXPORTING
        name       = lv_param_name
        value      = lv_visit_date
*       UNIT       =
*     IMPORTING
*       RETURNCODE =
    .
  CATCH cx_swf_cnt_cont_access_denied .
  CATCH cx_swf_cnt_elem_access_denied .
  CATCH cx_swf_cnt_elem_not_found .
  CATCH cx_swf_cnt_elem_type_conflict .
  CATCH cx_swf_cnt_unit_type_conflict .
  CATCH cx_swf_cnt_elem_def_invalid .
  CATCH cx_swf_cnt_container .
ENDTRY.

* Raise the event passing the prepared event container
TRY.
    CALL METHOD cl_swf_evt_event=>raise
      EXPORTING
```

© Copyright 2007 Avelon BV

```
        im_objcateg        = cl_swf_evt_event=>mc_objcateg_cl
        im_objtype         = lv_objtype
        im_event           = lv_event
        im_objkey          = lv_objkey
        im_event_container = lr_event_parameters.
  CATCH cx_swf_evt_invalid_objtype .
  CATCH cx_swf_evt_invalid_event .
ENDTRY.

COMMIT WORK.
```

If you want to check that your proposed visit date was passed to the workflow, you'll need to create a new workflow container element to hold the proposed visit date, adjust the event to workflow binding, and re-activate your workflow before testing it.

1.1.2.5 What about Configuration options for Workflow Events?

Those of you who are familiar with Business Object events in R/3 and ECC are also aware that often no code at all is needed in these systems to raise an event for Workflow - you just need to complete the configuration in one of the many event configuration tables. These tables are provided by many different modules within the application, such as HR, SD message control, status management, and change documents. As of ECC6, most of these configuration tables unfortunately do not yet support ABAP OO events via configuration, with one notable exception - change documents.

In transaction SWEC, change document event configuration, you can specify the object category **ABAP Class** when specifying an event. The only difficulty you may have is matching the key of the change document to the key of your ABAP Class. You can overcome this conversion issue by coding your own conversion routine as a function module.

This function module can be called any valid name (within the customer namespace) but must have the same interface as the template function module **SWE_CD_TEMPLATE_OBJKEY_FB_2**. All you have to do in this function module is use the data in the change document header/items to build the local persistent object reference for your workflow-ready ABAP Class.

You then add the function module to the **Function module** field in the relevant row of transaction SWED - which defines the change document object relationship to an object type.

Hopefully some of the other configuration options will catch up soon and allow for ABAP OO event configuration also.

1.1.2.6 Where to now?

The next blog will look at fleshing out our workflow by using ABAP OO attributes to add meaningful text, descriptions, and additional parameters to each part of our workflow.
Jocelyn Dart co-authored "Practical Workflow for SAP", and is currently supporting Workflow, Guided Procedures, MDM, xRPM, and xPD for Australia/New Zealand customers.

1.1.2.7 5 Using ABAP OO attributes in wfs and tasks

Jocelyn Dart
Business Card
Company: SAP Australia
Posted on Aug. 21, 2006 07:14 PM in ABAP, Business Process Management

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

This is number 5 in a series of blogs on ABAP OO for workflow, so you it would be a really good idea to make sure you have worked through the first 4 blogs. The examples we'll use here continue on from those we used in the earlier blogs.

Here's the list of the earlier blogs:

1. Why use ABAP OO with Workflow?
2. Getting started with ABAP OO for Workflow ... using the IF_WORKFLOW interface
3. Using ABAP OO with Workflow Tasks
4. Raising ABAP OO events for Workflow

1.1.2.9 A little background history for those who know BOR

VERY HELPFUL TIP! If you are an ABAP OO programmer and never did any code in the Business Object Repository, skip this section... you don't need to know about it, and you probably don't want to know either.

Once upon a time, in the land of BOR (Business Object Repository), there were 3 types of attributes:

1. Database attributes
2. Virtual attributes
3. Status attributes

Database attributes were always the easiest of attributes - just say what table and field you want to read and the BOR would generate some code to read all the data from a record of the table. You just had to fill in the blanks. Then you could add other database attributes referring to the same table and they would all use the same pre-generated code. Easy for workflow devlopers, but not always the most efficient way to read code, as you may only be using one or two fields but you still read the entire table row.

Virtual attributes were the most useful type of attribute as you could derive anything here including complex values, structures, tables, and instances of other objects.

Status attributes were a neat trick for business objects that used SAP status management, but most programmers didn't even know they existed. By using a special interface, and filling in the code behind a few predefined virtual attributes, you could then generate a yes/no flag to indicate if a particular status of that object was active.

© Copyright 2007 Avelon BV

In ABAP OO we no longer make the distinctions between these different types of attributes. So what do you lose? No more pre-generated code. And what do you gain? Flexibility and control over how your attributes are coded so that you can make the code as performance-efficient as possible.

Tip! Wanting to find whether a status attribute is set or not? Just call function module STATUS_OBJECT_READ with the status object number. All the statuses set for that object will be returned.

1.1.2.10 How do I define ABAP OO attributes for workflow?

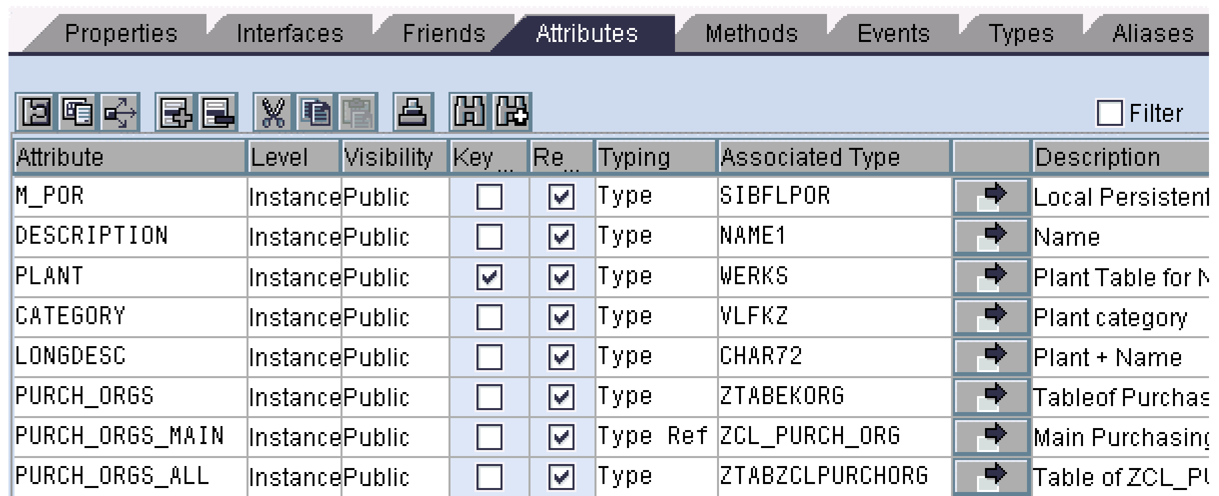
Simple - just define them on the attributes tab of the ABAP OO Class. You need to specify:

Attribute id

Instance or Static or Constant

Data type

Don't forget to make the attributes **Public** so they can be used by workflow. You can also specify a description and an initial value if you wish. In the example below a number of attributes have been added to the ZCL_PLANT class we have been using in the previous blogs to give you an idea.



Attribute	Level	Visibility	Key...	Re...	Typing	Associated Type		Description
M_POR	Instance	Public	☐	☑	Type	SIBFLPOR		Local Persistent
DESCRIPTION	Instance	Public	☐	☑	Type	NAME1		Name
PLANT	Instance	Public	☑	☑	Type	WERKS		Plant Table for N
CATEGORY	Instance	Public	☐	☑	Type	VLFKZ		Plant category
LONGDESC	Instance	Public	☐	☑	Type	CHAR72		Plant + Name
PURCH_ORGS	Instance	Public	☐	☑	Type	ZTABEKORG		Tableof Purchas
PURCH_ORGS_MAIN	Instance	Public	☐	☑	Type Ref	ZCL_PURCH_ORG		Main Purchasing
PURCH_ORGS_ALL	Instance	Public	☐	☑	Type	ZTABZCLPURCHORG		Table of ZCL_PU

Hint! If you don't want outside programs to be able to change your attribute it's also a good idea to mark it as **Read Only**. I like to do this for safety's sake.

1.1.2.11 Can I have attributes that are structures or tables?

Of course - the data type of your attribute just needs to be a structure or table type. There's no restriction on the complexity of the structure or table - and in SAP NetWeaver 6.20 and above, being able to handle components of structures and tables in workflow is much easier - especially during binding (parameter passing).

1.1.2.12 Can I have attributes that are objects themselves?

Of course - the data type of your attribute just needs to be a TYPE REF TO the appropriate ABAP OO class. You can then refer to attributes of objects that are attributes of another object. While there's no set limit on how many levels down you can go, most workflow developers don't like to delve too deep - because the technical ids become

© Copyright 2007 Avelon BV

very long and unwieldy. As a general practice if I want to go down further levels, I usually create a workflow container element for the object reference, then use a **Container Operation** step (a simple assignment step - which behaves something like an ABAP MOVE command) to assign the attribute-which-is-an-object-reference to the container element at runtime. I can then easily refer to attributes of the container element in subsequent steps of the workflow.

1.1.2.13 Can I have attributes that are BOR objects themselves?

Of course - but this is a little more involved so it's a separate blog topic for later in this blog series.

1.1.2.14 So where do I put the attribute code?

This is very straightforward. As a rule of thumb you put the attribute code of instance attributes in the CONSTRUCTOR method, and the attribute code of static attributes in the CLASS_CONSTRUCTOR method. Like this:

```
METHOD constructor .

* Filling our key attribute from a parameter
* as we did in the previous blogs
  me->plant = iv_plant.

* Now add some code to supply values to other attributes
  SELECT SINGLE *
         FROM  T001W
         INTO me->T001W
         WHERE WERKS = iv_plant.

  category = me->T001w-vlfkz.
  description = me->T001w-name1.
  CONCATENATE me->plant me->description INTO me->longdesc.

ENDMETHOD.
```

If you don't want to use the CONSTRUCTOR or CLASS_CONSTRUCTOR method that's ok. You can always code a separate Public method to derive your attributes, but it does mean you add the overhead of calling that method (as a background task) prior to using the derived attributes in your workflow. I don't personally recommend this approach in 6.40 or above, but in 6.20 it's a way of getting around the lack of functional methods (more on these later).

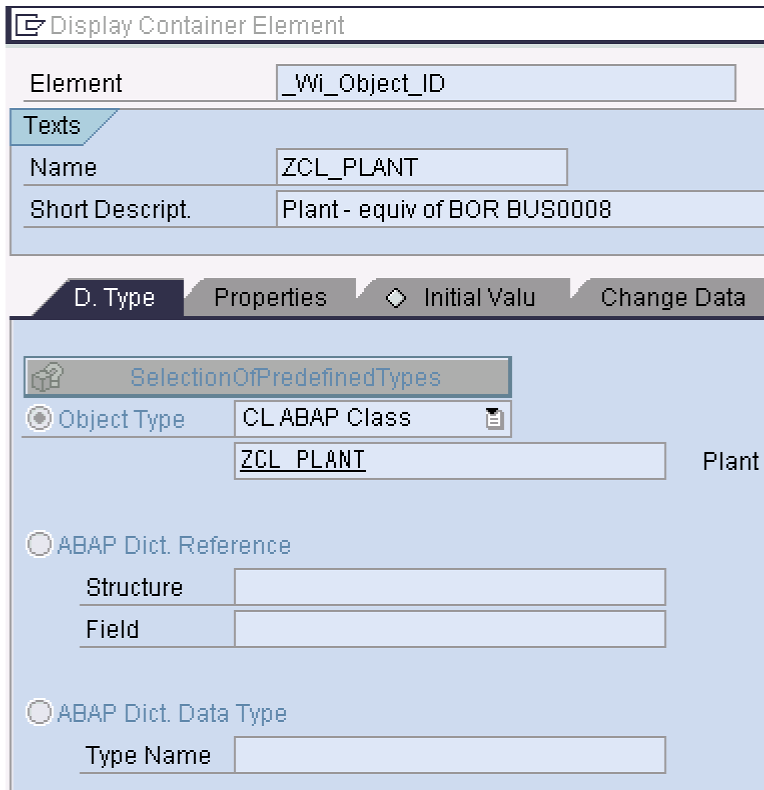
Tip: You can also use private methods as a way of modularizing code within your CONSTRUCTOR or CLASS_CONSTRUCTOR method.

1.1.2.15 Using ABAP OO attributes in tasks

For the sake of those of you who don't do a lot of workflow, before you use an attribute of an object, you must first have an object reference in the container (i.e. the data area of your task). Often the object reference is created automatically for you, e.g. an object reference with techncial id _WI_Objectid is automatically created when you assign your ABAP OO Class to the task on the Basic Data tab.

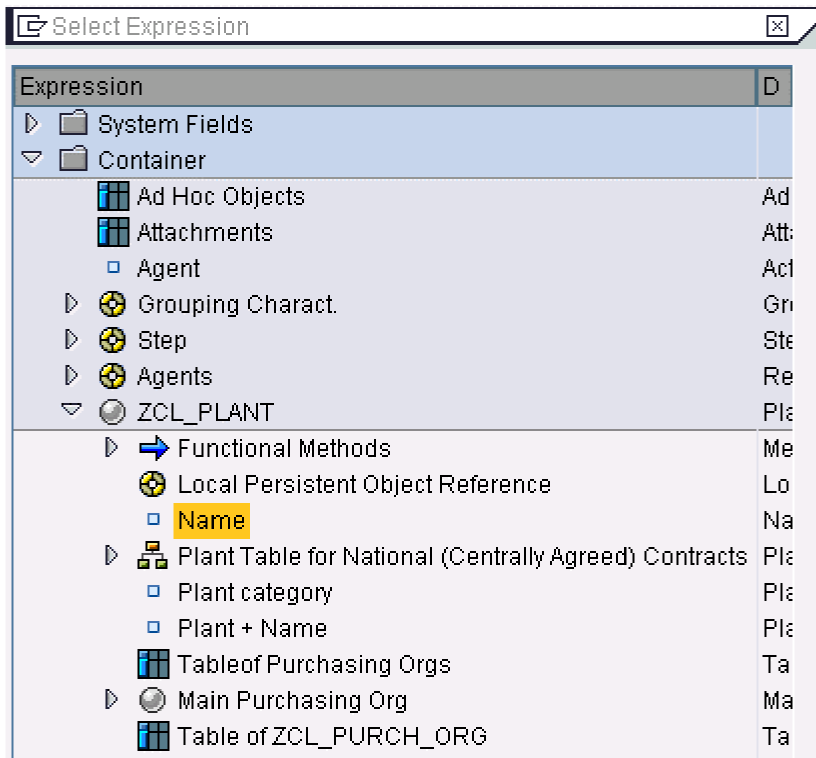
© Copyright 2007 Avelon BV

You can also create your own additional object references in the container quite easily. Go to transaction PFTC_CHG, open your "Display Plant" task from blog 3. Go to the **Container** tab. Press the **Create Element** button. Give your container element a technical name, description, data type, and make sure you mark it as **Importing** - so that you can pass some data into it later when you test it. You should see something like the example below in your "Display Plant" task from the previous blogs.



You can use an attribute in the **Work item text** on the **Basic Data** tab. Just position your cursor where you want the attribute to be inserted, press the **Insert Variables** button just about the work item text, and use the expression help to expand and click on the appropriate attribute. You can see the expression help below.

© Copyright 2007 Avelon BV



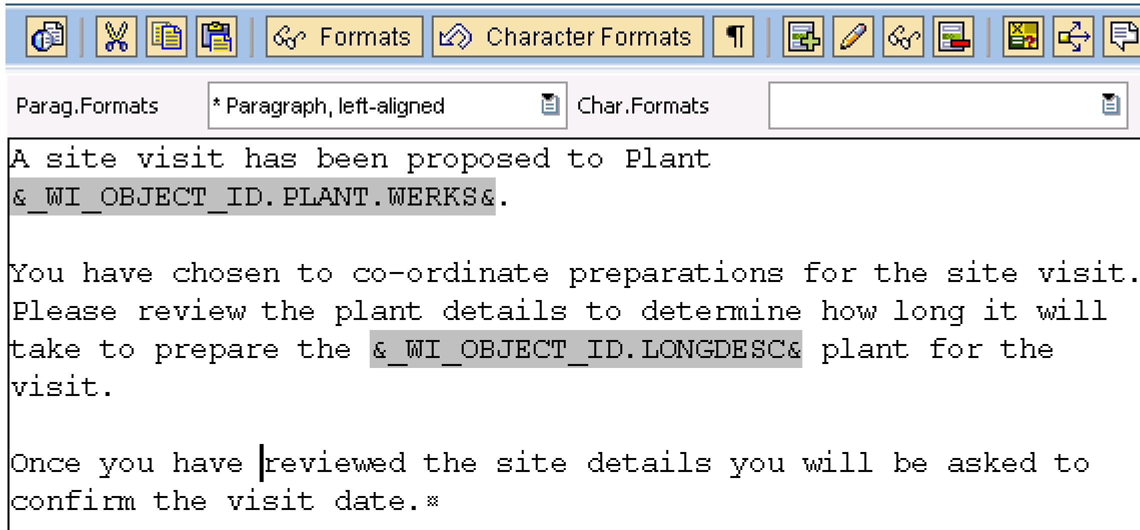
You see below why it is important to use the expression help as the attribute needs to be returned in a specific format.

Of course it only makes sense to use simple value attributes here as you only have about 50-70 characters of text that will show as the subject line of your work item in the inbox (depending on which inbox you use).

You can also use attributes in the **Long text** of your work item. Go to the **Description** tab, make sure **Task Description** is selected and press the **Change Description** button. Position your cursor where you want the attribute to go. Use the menu option ***Insert -> Expression*** to bring up the expression help. Then just expand the expressions and select the appropriate attribute. The end result should look something like this:

© Copyright 2007 Avelon BV

Task description Change



Note: If you use complex attributes here you may be asked for more details - such as whether you want each row of a table attribute to appear as a separate line of text at runtime.

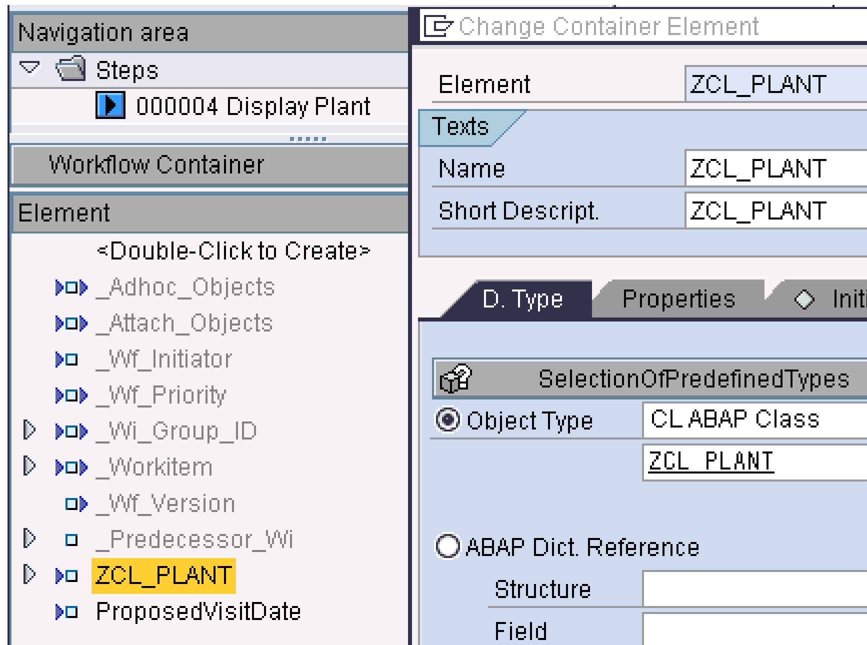
Now just test your workflow as we did in the previous blogs. In your inbox you should see that the inserted expressions have been replaced by actual values, such as the plant id and description.

1.1.2.16 Using ABAP OO attributes in workflows

Just as with tasks, before you use an attribute of an object ion a workflow, you must first have an object reference in the container (i.e. the data area of your workflow). Often the object reference is created automatically for you, e.g. when you added your "Display Plant" task to your workflow a container element was automatically created based on the ABAP OO Class used in the "Display Plant" task.

You can also create your own additional object references in the container quite easily. In the bottom left hand pane of the Workflow Builder (transaction SWDD), use the drop down to display the Workflow Container tray. Double-click on the line **Double-click to Create**. Give your container element a technical name, description, and data type. As we mentioned above, you should already have an appropriate container element in your workflow container that looks something like this:

© Copyright 2007 Avelon BV



Tip: You only need to mark it as Importing if you are going to pass some data into this element as you start the workflow - if you are just going to use it to hold data between steps of the workflow it doesn't need to be importing or exporting.

You can use ABAP OO attributes in any of the control steps in a workflow such as a Container Operation, Loop Until, Multiple Condition, User Decision, etc. Just as with tasks, any time you want to reference an attribute of your ABAP OO Class you simply use the expression help to expand and select the appropriate attribute.

To try this out, add a user decision step as the final step in the workflow we created in the previous blog. In the workflow builder flowchart, right-click on the "Workflow Completed" step and you will be presented with a list of step types. Choose the **User Decision** step type (this is a simple question/answer step). On the Decision page set up the title, outcomes and agents as shown below. Use the drop down help next to **Parameter 1**to select an appropriate attribute of your plant class.

© Copyright 2007 Avelon BV

User Decision 000008 How much preparation time is needed for plant &1 ?

Decision | Details | Control | Outcomes | Notification | Latest End | Requeste

Title: How much preparation time is needed for plant &1 ?

Parameter 1: &ZCL_PLANT.DESCRIPTION& | Parameter 2:

Parameter 3: | Parameter 4:

Agents

Expression: &_WF_INITIATOR& | Initiator of Workflow Instanc

Excluded

Decision Options

Decision Texts	Outcome Name
1 Week	Week
1 Month	Month
1 Quarter	Quarter

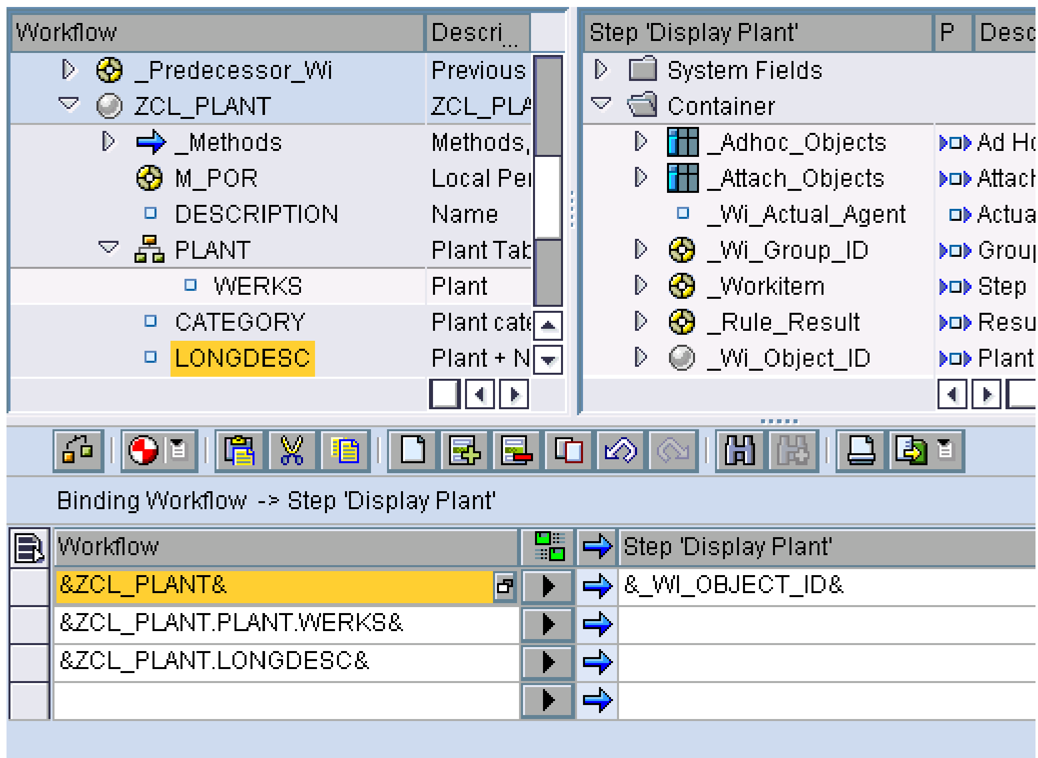
Note: The attribute will replace the &1 placeholder in the title.

Transfer the step back to the flowchart by pressing the green tick. Activate your workflow and test it again. Now after viewing your plant you should be presented with a second work item giving the question and answers you created in your decision step and with the plant attribute given a runtime value.

1.1.2.17 How do I pass attributes as parameters and handle narrowing cast?

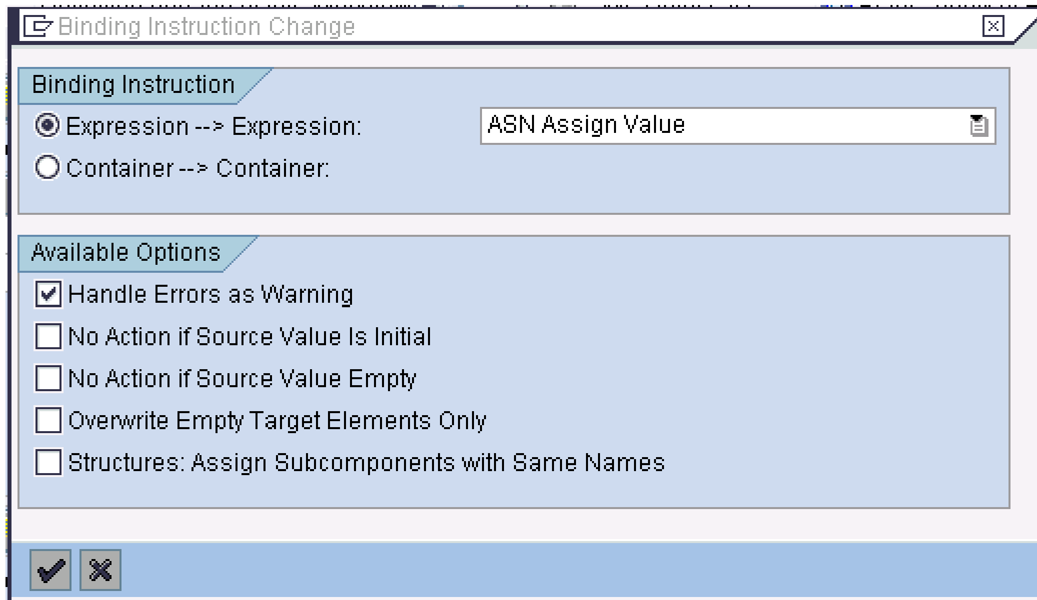
There's one other place where you can use ABAP OO attributes in workflow, **Bindings**, i.e. the workflow term for parameter passing. Open your workflow again in the workflow builder (transaction SWDD), and drill down on your "Display Plant" task, then press the "Binding (Exists)" button immediately below the Task and Step Name fields. You should see a window appear that looks something like this.

© Copyright 2007 Avelon BV



In the top panes of the window the workflow and task containers are shown. Down below are the bindings from workflow to task, and from task to workflow (the arrows show you which direction is being used). In the image above you can see I've expanded the plant class and dragged and dropped some attributes of the plant class into the binding. Of course each workflow container element on the workflow side needs to be matched to a task container element on the task side - so the above binding is not complete. I'm missing is a suitable container element (aka parameter) in the task to pass it to. Which I could easily create in the task as described earlier in this blog.

What about if your attribute is a reference to another ABAP OO Class? Can you pass it to a more generic or more specific class reference? Yes of course. Look closely at the binding window again. Between the workflow container element and the task container element it has been assigned to you will see a blue arrow (giving the binding direction) and a button with a filled in arrowhead on it. Press the button and you should see a window like this.



If you want to pass an ABAP OO Class reference to a more generic reference (widening cast) you don't have to even look at this window. But if you want to pass an ABAP OO Class reference to a more specific reference (narrowing cast), all you have to do is check the **Handle Errors as Warning** checkbox and you will pass your class without problem. Of course it's still up to you to make sure the two references are compatible.

1.1.2.18 But what about the performance implications of calculated attributes?

I know some of you are well aware that just because you might need an attribute for one particular task or scenario doesn't mean you want to code it in a CONSTRUCTOR method. Particularly if that attribute is resource intensive. So what's the best way to handle this scenario? **Functional methods** - which of course are the topic of the next blog.

Jocelyn Dart co-authored "Practical Workflow for SAP", and is currently supporting Workflow, Guided Procedures, MDM, xRPM, and xPD for Australia/New Zealand customers.

© Copyright 2007 Avelon BV

1.1.2.19 6 Using functional methods in wfs and tasks

Jocelyn Dart
Business Card
Company: SAP Australia
Posted on Dec. 19, 2006 04:58 PM in ABAP, Business Process Management, SAP Business Workflow

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

This is number 6 in a series of blogs on ABAP OO for workflow, so you it would be a really good idea to make sure you have worked through the first 5 blogs. The examples we'll use here continue on from those we used in the earlier blogs.

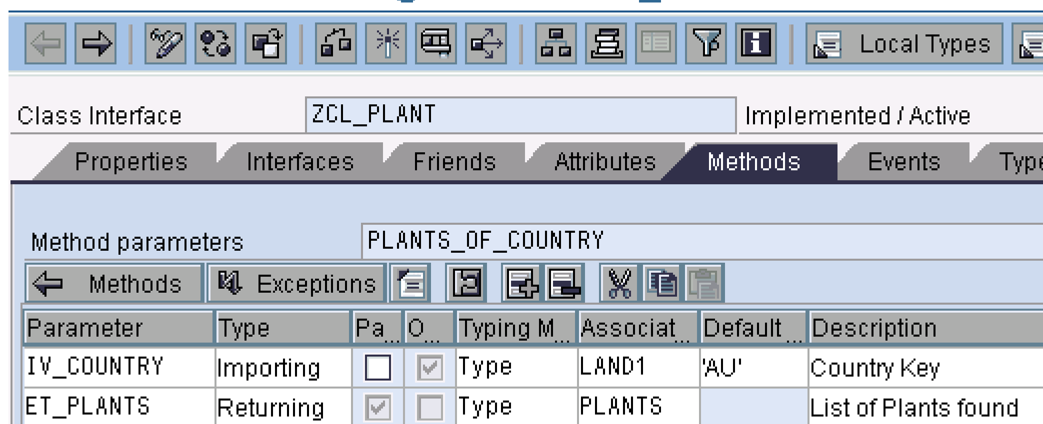
Here's the list of the earlier blogs:

1. Why use ABAP OO with Workflow?
2. Getting started with ABAP OO for Workflow ... using the IF_WORKFLOW interface
3. Using ABAP OO with Workflow Tasks
4. Raising ABAP OO events for Workflow
5. Using ABAP OO attributes in Workflows and Tasks

1.1.2.21 What are functional methods?

A functional method is a method that returns a single result, i.e. a method that has a **Returning** parameter. Typically a functional method calculates a value, e.g. you might use a functional method to return the number of entries in a table, or the formatted name of an employee or whatever. The method can have importing parameters, but does not have any other exporting parameters.

Here's a simple example using the same ABAP Class we used for the exercises in the previous blogs. This is a static method that retrieves a list of plants relevant for a nominated country. Here are the parameters:



And the method code - as you can see it's a very simple example.

```
METHOD plants_of_country.

  SELECT werks FROM  t001w
    INTO TABLE et_plants
    WHERE  land1  = iv_country.

ENDMETHOD.
```

1.1.2.22 What's the equivalent of functional methods in BOR?

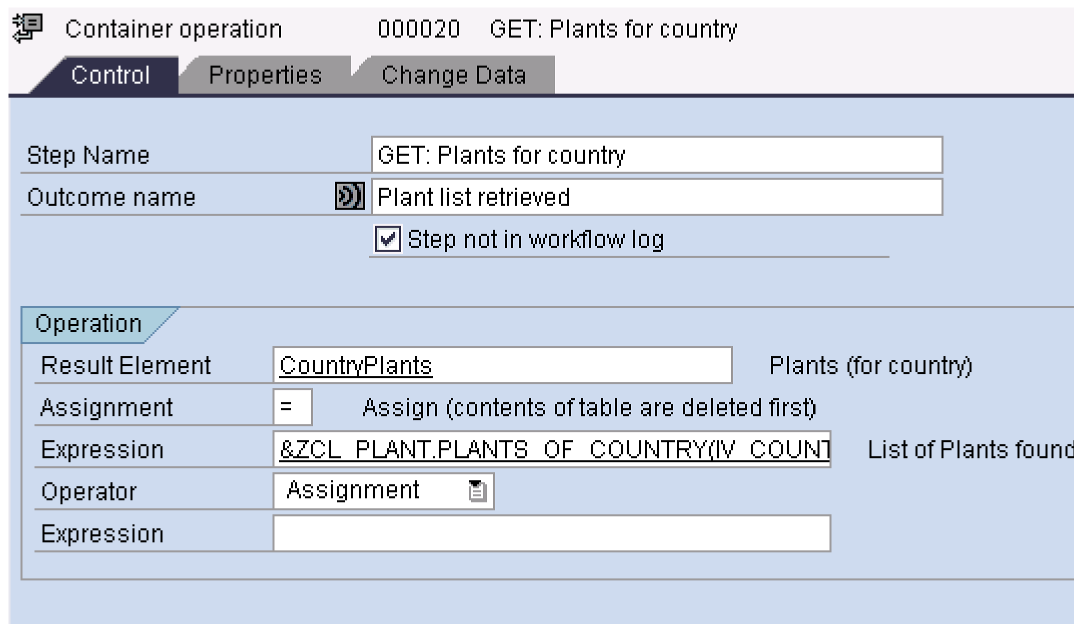
For those who are used to coding business objects (BOR) then functional methods replace **virtual attributes** (a calculation derived from the object instance). Virtual attributes, and their replacement functional methods, are particularly useful where you have a value that can be derived from the current object instance, but there is enough of a performance cost with deriving the value that you don't want to derive the value every time you construct the instance.

One advantage of functional methods over virtual attributes is that you can pass importing parameters. You can derive your calculation from the class instance (assuming you are using an instance method of course) **and** the importing parameters. However its a fairly simple mechanism so its easy to use this for simple parameters that can be hardcoded into the workflow, but you may still need to call the functional method as the main method of a normal task if you want to dynamically set the parameters.

1.1.2.23 How can I use functional methods in workflows?

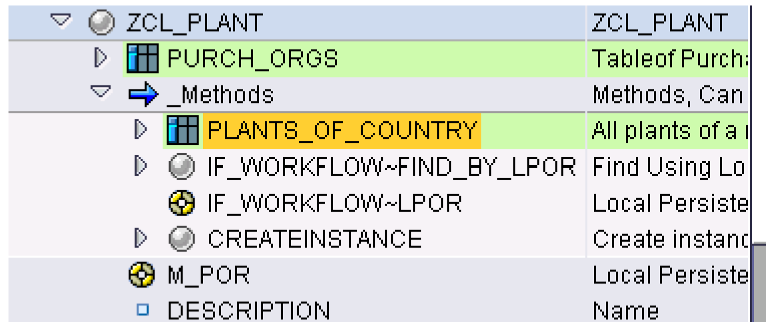
You can use a functional method in much the same way as an object attribute. For example: in a container operation, or as the source of a binding.

Here's an example where a functional method is used in container operation:



© Copyright 2007 Avelon BV

When selecting your functional method from the drop down look for the "methods" or "functional methods" section within the object. For example this is what the drop down on the Expression field of the Container Operation above looks like:



This is the format for passing importing parameters.

```
&ZCL_PLANT.PLANTS_OF_COUNTRY(IV_COUNTRY='DE')&
```

If you have multiple parameters, just separate them with an ampersand (&).

Note: You can't use functional methods with importing parameters directly in logical expressions as yet (you can select them but an error is returned) - as the condition editor does not provide a way to enter the parameter values. So in this case, use your functional method in a container operation first, then use the derived container element in your logical expression.

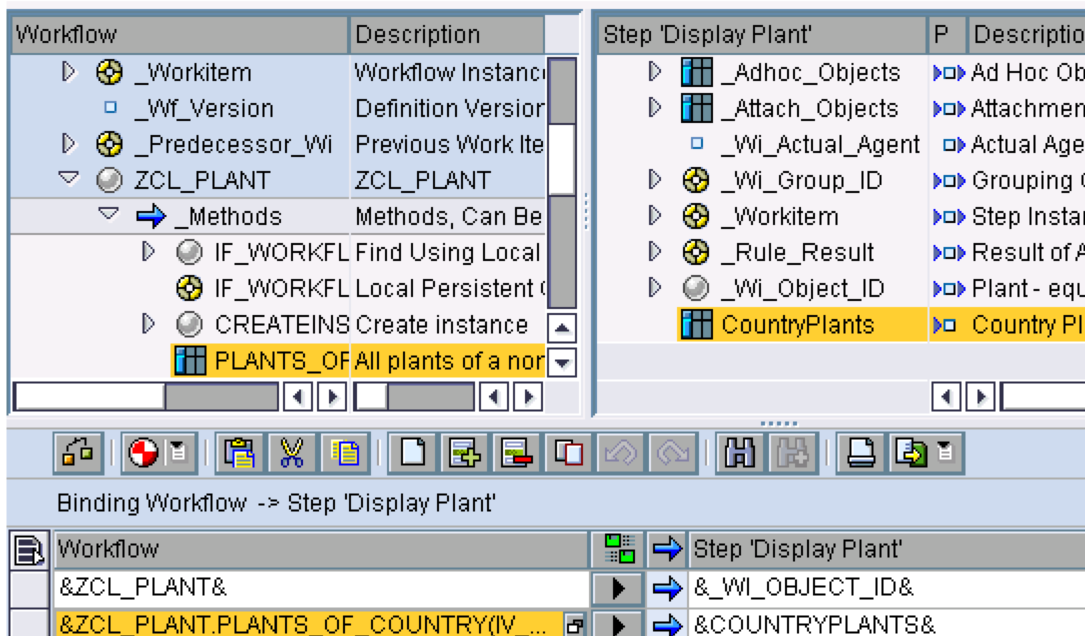
1.1.2.24 How can I use functional methods in tasks?

You can use a functional method in much the same way as an object attribute in a task, for example in the subject text of a task or in the binding to a method, with one exception:

You can't call a functional method directly from the long text of a task.

This is a SAPScript restriction not a workflow restriction - i.e. SAPScript doesn't know how to call a functional method to build the text. However this is not a major problem as you can always create an import container element to hold the result of the functional method in your task, and then in the binding source the container element from your functional method. The example below gives an idea of what such a binding would look like.

© Copyright 2007 Avelon BV



1.1.2.25 Can I use a functional method in a task like other methods?

Yes of course you can use a functional method as the main method of a task, and then it behaves exactly like any other method called from a task. This could be helpful if you need to handle exceptions for that particular workflow, or if it's simply easier to build that way as for this particular workflow you need to pass many/complex importing parameters.

1.1.2.26 How do I decide whether to use a functional method or create a task?

The decision is much the same as for virtual attributes in the BOR repository, i.e.

Do you need to return more than one value? - If so use a task (of course this task will call a normal method - i.e. one with multiple exporting parameters)

Do you need to handle exceptions? - If so use a task

Do you only need to return a single value and exceptions are unlikley or irrelevant (will be ignored, or will be checked in a subsequent step)? - If so use a functional method

1.1.2.27 Can I use functional methods for BOR objects?

You may notice that functional methods (i.e. methods with the special "result" parameter) can also be selected. However I have personally found them a little unreliable in practice (they tend to throw errors or fail to pass data). A safer approach would be to link the BOR object to an equivalent ABAP Class, and handle the call to the BOR method inside an ABAP Class method. Linking BOR objects to ABAP Classes is the topic of our next blog.

Jocelyn Dart co-authored "Practical Workflow for SAP", and is currently supporting Workflow, Guided Procedures, MDM, xRPM, and xPD for Australia/New Zealand customers.

© Copyright 2007 Avelon BV