Take another look at ABAP dynpros — advanced programming techniques for better user interfaces with more control and fewer surprises

by Doris Vielsack and Arndt Rosenthal

Doris Vielsack
SAP NetWeaver
ABAP QM,
SAP AG

Arndt Rosenthal
SAP NetWeaver
ABAP UI Services,
SAP AG

(Full bios appear on page 102.)

Dynpro technology has long served as the basis of reliable, user-friendly SAP GUI interfaces for SAP applications, and a wide range of SAP-supplied and custom-built applications is based on this well-known technology. SAP NetWeaver ’04 introduces the next generation of interface technology, Web Dynpro, which promises enhanced features and capabilities for both developers and users, and is now the preferred SAP user interface (UI) strategy. However, the widely used and proven classic dynpro programming technique will continue to be a presence in existing applications as well as new development as Web Dynpro matures as a technology. This is the second installment of a two-part article series intended to help both dynpro beginners and veterans build better dynpros and efficiently troubleshoot and avoid common problems.

In the first installment of this two-part article series (on page 45 of this issue of *SAP Professional Journal*), we addressed how to build better dynpros with a focus on intelligent design from the dynpro programming perspective. In this second installment, we go beyond the basics of dynpro programming to explore some advanced techniques for enhancing your dynpros and writing better ABAP programs to support them:

- We start by discussing how to combine classic dynpros and custom front-end controls to incorporate more complex GUI elements, such as trees and viewers.
- We then turn to the ABAP programmer perspective and provide guidance for working effectively with dynpros in your ABAP programs.
- Finally, we complete our tour of dynpro programming with an example that illustrates how easy it is to write clear, understandable ABAP programs that combine object orientation and dynpros in order to achieve robust dialog programs without much effort.

Let's begin the second part of our dynpro programming tour with a discussion of how front-end controls work and how you use them in combination with dynpros to enhance your UI.

Note!

As with the first installment of this two-part article series, the information in this second installment applies to all supported SAP R/3 and SAP NetWeaver releases, except where release-specific differences are explicitly noted. It addresses classic dynpros only. Web Dynpros, introduced with SAP NetWeaver '04, are a completely different technology. Look for upcoming *SAP Professional Journal* articles that cover Web Dynpro technology in detail, and also visit www.sap-press.com for additional resources.

Making dynpros and custom front-end controls work together

As you learned in the previous article, one of the big advantages of a dynpro is its simplicity. Define a field in the Layout section of the Screen Painter (transaction SE51), assign a name that corresponds to a field in your ABAP program (preferably with a reference to an ABAP Dictionary type in order to gain maximum benefit from the dynpro processor), and you are done. With this simple declaration, everything else will automatically work:

- Data transport (including conversion of data representation) between the variable in the ABAP program and the dynpro, and vice versa
- GUI services such as help and value help
- Formal validity checks for entered values

But this simplicity in itself also imposes some disadvantages. For example, the GUI elements that are used at the front end are also quite simple. The actual work is done at the back end, whether that consists of making a readable number out of all the zeros and ones used by the computer, checking if an entered name of a city really corresponds to the chosen country, or doing whatever is necessary to support users in their job. (See the sidebar on the next page for more information about the roles of the front end and back end.)

What do you do if you need more complex GUI elements with some features running at the front end? For example, suppose you want an element that visualizes a tree with features to expand or collapse the nodes. Obviously you do not want to trigger communication with the back end with each click, which would cause the front end to wait for an answer from the back end (and in the meantime annoy users by displaying the hourglass icon). Therefore, you need some additional code running at the front end to work with the given data and maintain the element state. For these types of elements, you use the custom front-end controls (known simply as *front-end controls*) that were introduced in SAP R/3 4.5. They are designed to enable you to build more complex (in other words, more intelligent or interactive) GUIs.

With front-end controls, you can now build powerful GUIs that greatly increase user productivity and satisfaction. But there are some concepts you need to be aware of, including some potential pitfalls, in order to make your application user interface really great. We'll look at these next.

Understanding how front-end controls work

As the name implies, front-end controls run at the front end. They run as part of the SAP GUI, providing the intelligent behavior and maintaining the state of the control (for example, which node of the tree is expanded, which node is marked, and so on). Within the SAP GUI for Windows environment, ActiveX controls are usually used for implementing front-end controls.

Understanding the roles of the “front end” and “back end”

According to the SAP client-server architecture, communication with a user is divided into two main parts. One part, acting as the client, is responsible for displaying (that is, rendering) the data and enabling the user to interact with the system. GUI elements are used for this purpose. These GUI elements are components of the UI, such as input fields, labels, icons, and so on. This part is hypostatized as the SAP GUI running on a user’s computer. Because this part is closest to the user, it is referred to as the *front end*.

The other part, acting as the server, is responsible for providing the data to be displayed and reacting to user interactions. This part usually runs on a server machine in some computer room, far away from the user, and is therefore referred to as the *back end*. The back-end part is realized within the SAP application server by the dynpro processor and the application program components (that is, the ABAP modules) that it calls.

User interaction is distributed between the two parts, with some elements running at the front end (that is, within the SAP GUI program) and others running at the back end. With classic dynpros, most of the dialog processing — computing external representations of data, performing validity checks of input data, and so on — is done at the back end. The SAP GUI essentially does the rendering and triggering of user interaction events.

The simple GUI elements that are used for dynpro rendering only need the value to display and a few other properties, such as whether the element is highlighted, to do their work. In contrast, front-end controls need much more information, such as the data of all the nodes, which node is initially expanded, what to do if a node is double-clicked, and so on. Therefore, the interface to a front-end control consists of data as well as methods for manipulating the state of the control.

Because most of the application program runs at the back end, you need a way to call the interface methods of the control at the front end from the back end. For this purpose, a proxy for the front-end control exists at the back end. You work with this proxy when using front-end controls — to visualize data as a tree with complex user interaction, for example. To increase the level of confusion, in the SAP world the term *front-end control* is often used when what is really meant is *front-end control proxy*.

This mechanism relies on an additional generic component called the Control Framework. This framework is responsible for the technical aspects of the control, such as communicating with optimal performance between the control proxy at the back end and the control itself at the front end. The Control Framework also manages the Automation Queue, which provides an optimized mechanism for handling control method calls.

As a developer, you are responsible for the following tasks in your application program when you want to use a front-end control to display data:

- Create an instance of the desired front-end control (the back-end proxy, of course) within the application program.
- Register the event handlers for the instance events.
- Use the instance interface to provide the data to display and set the front-end control to the correct state.

The Control Framework does everything else — creating the front end, transporting the data, and so on. Refer to the SAP online documentation if you

want to know more, as we want to keep the focus of this article on dynpro programming.[1]

Understanding how dynpros and front-end controls work together

Apart from ABAP lists (which actually use a special internal dynpro to display data), dynpros are the only way to build a UI within classic ABAP.[2] In order to achieve more ambitious UI objectives, you need to use a combination of front-end controls and dynpros. Front-end controls can only be displayed together with a dynpro, usually within a dynpro where the dynpro has a container in which the control is displayed. They can also be displayed in parallel with a dynpro, either as a docking control that is displayed on the window border or as a separate control in its own amodal window.

Because the dynpro processor starts all ABAP processing, the Control Framework also runs within the dynpro processor. The dynpro processor gives control to the Control Framework at certain points when processing a dynpro, regardless of whether any front-end controls are actually used. The Control Framework then determines whether something needs to be done because only the Control Framework is aware of the existence of any front-end controls. The dynpro processor doesn't care what the Control Framework does and simply relies on a return code from the Control Framework to determine what to do next. It can either continue dynpro processing or immediately send the dynpro together with any front-end control data to the SAP GUI, which then displays the screen containing both the dynpro and control content to the user.

[1] For more information about the Control Framework, refer to the SAP Help Portal at http://help.sap.com. In the SAP NetWeaver 2004s documentation, navigate to SAP NetWeaver Library → SAP NetWeaver by Key Capability → Application Platform by Key Capability → ABAP Technology → UI Technology → Frontend Services (BC-FES) → SAP Control Framework. Also see the article "SAP Controls Programming Essentials — What Every ABAP Developer Now Needs to Know" (*SAP Professional Journal*, November/December 2001).

[2] Newer techniques such as ABAP-based Business Server Pages (BSPs) and Web Dynpros now exist, but they are outside the scope of this article.

As you can see, dynpros and front-end controls are actually very loosely coupled. (See the "Control handoffs between the dynpro processor and the Control Framework" sidebar on the next page for more details about this interaction.)

Note!

Knowing the details of how the Control Framework works is not absolutely necessary. However, being familiar with the names of its form routines is sometimes useful because you may see them in the ABAP call stack (for example, in ABAP short dumps). If you need to analyze short dumps while troubleshooting a dynpro problem, knowing that all form routines that start with `%_CTL` belong to the Control Framework is helpful.

Some critical programming points to remember

To recap, using a front-end control in your user interface always requires using a dynpro as well. The event handlers of the front-end control, in addition to the normal dynpro flow logic, run under the control of the dynpro processor. Consequently, programming with front-end controls introduces some additional considerations for managing all of these elements, especially if you are responsible for developing both the dynpro flow logic and the front-end control event handlers.

The main point to keep in mind is the coexistence of two separate machines, each of which maintains its own state:

- The dynpro processor (which is actually a state machine) maintains the current state of dynpro processing. You manipulate this state via ABAP statements such as `MESSAGE` or `LEAVE SCREEN` in the implementation of the modules that are called in the dynpro flow logic. The current state of the dynpro processor includes information such as:

Control handoffs between the dynpro processor and the Control Framework

Understanding what happens behind the scenes is beneficial if you are actively working with dynpros and front-end controls. The dynpro processor automatically hands control to the Control Framework and vice versa in a well-ordered sequence:

- After a dynpro is loaded, the dynpro processor calls the Control Framework (regardless of whether any front-end controls are used) to perform its internal initialization if necessary. If no front-end controls are used, the Control Framework does nothing. For readers who are interested in what the Control Framework does at this point, the implementation is in the form `%_CTL_INIT` of the program SAPMSSYD.
- The dynpro processor next calls the Control Framework at the end of PBO after all of the dynpro field transports are finished. For interested readers, the form `%_CTL_OUTPUT` of the program SAPMSSYD does the work here.
- Each time data is sent to the SAP GUI, the form `%_CTL_OUTPUT_FLUSH` of the program SAPMSSYD program is invoked. This action gives the Control Framework a chance to release the Automation Queue (which is built up by the back-end proxies of the used front-end controls) to the corresponding front-end controls.
- If the SAP GUI sends data to the application server (for example, any user input and a function code describing the event that triggered the back-end communication), the dynpro processor automatically gets control. The dynpro processor forwards the data to the Control Framework (specifically, to the form `%_CTL_INPUT` of the program SAPMSSYD) to decide whether it should handle the event based on the event description. In general, function codes that start with `%_G` indicate events that are to be handled by the Control Framework.

Continues on next page

  - Which fields are transported
  - Which statement in the flow logic will be processed next
  - Whether error processing, help processing, or exit command processing are active
- The Control Framework maintains the current state of its own processing and does not know anything about the state of the dynpro processor. You manipulate this state by calling the appropriate methods of the back-end proxies of the front-end controls. The current state of the Control Framework includes information such as:
  - Instances of front-end controls (each of which has its own state)
  - The state of the Automation Queue
  - Registered event handlers

Because the Control Framework runs within the dynpro processor, it is technically possible to manipulate the state of the dynpro processor from within an event handler of a front-end control (for example, using the `MESSAGE` statement). But this action is generally dangerous, resulting in strange system behavior such as skipped modules in the dynpro flow logic or even ABAP short dumps. To avoid these consequences, be careful when using ABAP statements such as `MESSAGE` and `LEAVE SCREEN` that change the dynpro processor state.

The `MESSAGE` statement influences dynpro

Continued from previous page

- For dynpro events (that is, events where the function code does not start with `%_G`), control is automatically returned to the dynpro processor, which then continues with normal dynpro processing (data transport, PAI, and so on).
- For front-end control events (that is, events where the function code starts with `%_G`), the Control Framework must first identify whether it is a system or application event (see the table below). Because the event type is defined in the description for each front-end control, the Control Framework asks the back-end proxy of the front-end control that triggered the event to make this determination. System events are usually completely handled by the Control Framework in the corresponding method of the back-end proxy of a front-end control. Application events are handled by methods that you as the application developer must provide, usually as ABAP object event handlers of the back-end proxy. Some system events may also require supporting application code, such as a method to retrieve the appropriate data in the value help control. In this case, application code runs in the context of a system event of a front-end control.

Event type	Description
System	The Control Framework automatically calls the event handlers (that is, the corresponding methods of the back-end proxy). The work is done by the form `%_CTL_INPUT` of the program SAPMSSYD. After the event handlers are finished, the Control Framework usually sends the data (that is, the data to display and the state of the front-end control) directly to the SAP GUI. This action triggers a new rendering of the corresponding front-end control, without any dynpro processing occurring. You can force dynpro processing to occur after the event handlers are finished by calling the method `CL_GUI_CFW=>SET_NEW_OK_CODE` in the event handler. You might need this ability in cases where user input in the front-end control requires a change to the data that is displayed in the dynpro, such as choosing a different node of a tree control. In this case, as with dynpro events, normal dynpro processing starts immediately after the event handlers return, with the OK_CODE field set to the value that is provided with the method call.
Application	After the Control Framework has pre-processed the event data, control goes back to the dynpro processor, which starts dynpro processing as usual. In this case, call the method `CL_GUI_CFW=>dispatch` in one of the PAI modules of the dynpro in order to call the Control Framework event handlers. This technique enables you to control when and in which application context the event handlers run. It is generally a good idea to always call this method because within a PAI module you cannot usually detect whether any front-end control events are pending. However, if for some reason the Control Framework event handlers were not invoked during PAI processing, the Control Framework automatically calls them after PAI is finished via the form `%_CTL_END` of the program SAPMSSYD. This action avoids leaving any pending front-end control events unhandled, which could lead to inconsistent application data.

processing by sending a message or initiating an error dialog, depending on the specified `TYPE` parameter.

Follow these guidelines when using this statement in your application code:

- In the context of a system event of a front-end control, never use messages of type E (error) or W (warning) to avoid disrupting dynpro processing.
- Even within a module of the dynpro flow logic, only use a message of type E or W when you are positive that no front-end control event could change the state of the dynpro data or issue a message of type E or W itself. Front-end controls are not involved in the processing that occurs when the dynpro processor locks fields against input during error or warning processing. Therefore, front-end controls remain enabled when an error or warning is displayed. After receiving any user input, the dynpro processor resumes processing at the correct point in the flow logic (refer to the section "Understanding where program execution continues after message handling" on page 70 in the first installment of this article series). Therefore, front-end control events may not be processed correctly, which usually leads to unexpected application behavior. The best way to avoid these types of problems is to disable all front-end controls by calling the appropriate method of the back-end proxy of each control before you issue a message of type E or W in a flow logic module.
- Messages of type I (information) or S (status) are not critical because they do not impact the current state of the dynpro processing.

The ABAP statements `LEAVE SCREEN` and `LEAVE TO SCREEN n` immediately stop the processing of the current dynpro. Never use them in a system event handler for a front-end control. These statements manipulate both the dynpro and the ABAP program stack. Depending on the current stack, these statements may often be executed successfully. However, if the event handler is called inside the dynpro message processing, these statements could lead to an inconsistency between the two stacks and cause a short dump.

On a final cautionary note, remember that the system event handlers for front-end controls run before the normal dynpro processing starts. More specifically, they run before the dynpro data is transported to the ABAP fields, which happens before or as part of PAI processing. (Refer back to the section "Controlling field transports between a dynpro and the ABAP program" on page 74 in the first installment of this article series.) Be careful when using dynpro data in these event handlers because the data in the ABAP fields could differ from what the user just entered on the dynpro.

Working with dynpros in your ABAP programs

So far we have focused on the dynpro side of the programming task. Next we turn our attention to the ABAP perspective to help you make smart choices in your ABAP code and avoid other types of potential pitfalls with your dynpros.

Options for defining ABAP variables used in dynpros

In ABAP, you define a variable either as a work area with the keyword `TABLES` or an ABAP variable with the keyword `DATA`:

```
TABLES SFLIGHT.
DATA SFLIGHT type SFLIGHT.
```

The keyword that you choose has a significant impact on how you can use the variable on a dynpro. If you define an ABAP variable with the keyword `TABLES`, you can specify the corresponding field on the dynpro as being defined in the ABAP Dictionary. You do this in the Screen Painter (transaction SE51) by setting the From Dict. (From Dictionary) attribute for the field. Based on the settings defined for the corresponding element in the ABAP Dictionary, the Screen Painter automatically inserts the Parameter ID (which is used in setting a default value for the field) and the conversion exit, if one exists, for the corresponding domain. This approach is useful because it means that all dynpros will handle the field in the same manner. However, you cannot change or delete these dictionary-defined attributes for the field.

If you define an ABAP variable with the keyword `DATA`, you cannot specify the corresponding dynpro

field as From Dictionary. You must remember to enter the values for the Parameter ID and conversion exit attributes in the Screen Painter when you define the dynpro. If you forget, you may experience unexpected differences in the contents of the field on different dynpros. However, other differences are more important. Except for F1 and F4 help, the field will not have other key characteristics that are defined in the ABAP Dictionary, such as:

- Foreign key test
- Automatic text adaptation for input or output field labels on dynpros
- Value range defined in a domain

If you need one of these features for a field on a dynpro, remember to use the keyword `TABLES` to define the variable for the corresponding field in your ABAP program.

Structuring the modules that define the ABAP variables

Modules are the only pieces of ABAP code that you can call directly in the dynpro flow logic. However, modules have a small disadvantage in that they do not recognize local variables, which means that you cannot structure code within them perfectly. Suppose you write the following block of code in your ABAP program:

```
MODULE initvalues.
  data mydata.
  …
ENDMODULE.
```

Based on the definition of the ABAP language, this code represents a global declaration of the `mydata` variable. As this implementation is error-prone and not very intuitive, a better approach when programming with dynpros is to call a method or form that contains both the variables and the dynpro code from within the module. The `initvalues` method serves this purpose in the following example:

```
MODULE initvalues.
  lcl_init=>initvalues( ).
ENDMODULE.
```

Even though you need to write a few more lines of code, you can structure your code as usual. We recommend that you follow this approach when programming with dynpros, although for space reasons we chose not to do so in the examples in this article.

Initializing and working with tabstrip controls

Like table controls (which we discussed in the first installment of this article series), all tabstrip controls that are used on a dynpro must have a corresponding representation in an ABAP program. The statement for creating the object for a tabstrip control is much simpler:

```
CONTROLS <contrl> TYPE TABSTRIP.
```

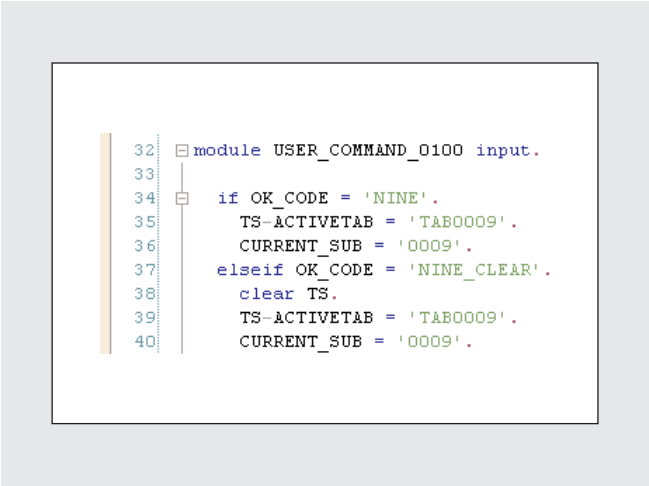
where `<contrl>` is the name of the tabstrip control.

The control `<contrl>` has only two components, `ACTIVETAB` and `_SCROLLPOSITION`, which are well-described in the ABAP keyword documentation. The `ACTIVETAB` component defines which tab page appears on top of the tabstrip control. The `_SCROLLPOSITION` component defines the tab that appears at the leftmost position of the tabstrip control. The SAP GUI automatically sets the `_SCROLLPOSITION` component, which you should normally leave unchanged. However, if you change the `ACTIVETAB` component in the ABAP program or set a tab to invisible, remember to also clear the `_SCROLLPOSITION` component. Otherwise, you may experience the strange effect of being able to see a tabstrip page but not its corresponding tab.

To illustrate this principle, we defined an example tabstrip control with 10 tabs (see **Figure 1**). Only five tabs are currently visible because the GUI window does not have enough room to display more. The purpose of the two buttons — To Tab9 and To Tab9 with clear — is to change the visible tab on the tabstrip control to Tab9.

Clicking on the To Tab9 button sets the OK code to `NINE` and sends it to the back end. Clicking on the To Tab9 with clear button sends an OK code of `NINE_CLEAR` to the back end. **Figure 2** shows the

Figure 1 Tabstrip control with 10 tabs



```
32  module USER_COMMAND_0100 input.
33
34    if OK_CODE = 'NINE'.
35      TS-ACTIVETAB = 'TAB0009'.
36      CURRENT_SUB = '0009'.
37    elseif OK_CODE = 'NINE_CLEAR'.
38      clear TS.
39      TS-ACTIVETAB = 'TAB0009'.
40      CURRENT_SUB = '0009'.
```

Figure 2 ABAP code to handle the OK codes for the buttons

Figure 3 Tabstrip control without clearing the _SCROLLPOSITION component

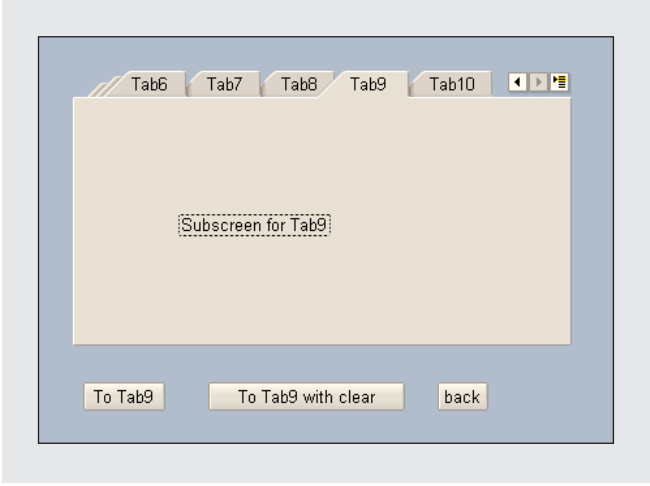


Figure 4 Tabstrip control after clearing the _SCROLLPOSITION component

ABAP code that is executed to handle these OK codes. The difference between the code for these two buttons is that the To Tab9 with clear button executes the `clear TS` statement, which clears all components of the tabstrip control TS, including the `_SCROLLPOSITION` component. The To Tab9 button leaves this component untouched.

Figure 3 shows the results of clicking on the To Tab9 button. Notice that the leftmost position of the tabstrip control is set to Tab1 instead of Tab9. The Tab1 value is still stored in the `_SCROLLPOSITION` component and therefore is where the SAP GUI automatically positions the leftmost tab.

Figure 4 shows the results of clicking on the To Tab9 with clear button. In this case, the `_SCROLLPOSITION` component is empty because we cleared its value. The SAP GUI does not receive any instructions regarding the leftmost tab location and therefore automatically positions the tabs appropriately and as expected on the tabstrip control.

Location is important — understanding dynpros and program groups

When an ABAP program is addressed for the first time by calling one of its objects (a report, class, subroutine, or dynpro), the program is loaded into memory in a logical unit that is referred to as a *program group*. When you call a program with a mechanism other than an external `PERFORM` statement (for example, with any of the statements `SUBMIT REPORT`, `CALL TRANSACTION`, `CALL DIALOG`, or `CALL FUNCTION`), a new program group is created for the called program. When you call a program via an external `PERFORM` statement, the program that contains the called subroutine is loaded into the same program group as the caller (assuming that this is the first call to the module pool that contains the form). This design was intended to facilitate sharing data between table work areas.[3]

Due to this concept of a program group, dynpros may not appear to belong to the correct caller when:

- A subprogram of a different program is called from an ABAP program via an external `PERFORM` statement
- A subscreen of a different program is embedded in the flow logic of a dynpro with the `CALL SUBSCREEN` statement, which implies a call to the ABAP program of the called dynpro

Suppose a program `<prog>` is addressed for the first time in the dynpro flow logic. What do you think will happen with the following subscreen call?

```
CALL SUBSCREEN sub INCLUDING <prog>
<dynnr> .
```

where `sub` is the subscreen area, `<prog>` is the program to which the dynpro belongs, and `<dynnr>` is the number of the dynpro.

There are two possibilities:

- The program `<prog>` is loaded into the same program group as the caller if the called dynpro is part of a module pool.
- If the program `<prog>` is part of a function group, the program is loaded into its own program group.

Unlike the `call subscreen` statement, the `call screen` statement does not have a parameter to specify the program to which the dynpro belongs. Therefore, you cannot use this statement to call the dynpros of two different programs in the same program group. The `call screen` statement always uses the name of the first program of a program group, which is why only the dynpros of the first program become visible. This difference is the reason why defining a dynpro in a module pool vs. in a function group is so important.

Let's look at an example. Suppose you have two module pools, M1 and M2. The program starts in the M1 module and then calls the M2 module via an external `PERFORM`. Both modules are loaded into the same program group because a call via an external `PERFORM` statement to a module pool does not create a new program group. However, the dynpros of the M2 module are not available because only the dynpros of the first program of a program group (in this case, the M1 module) are visible. When you call a dynpro from the ABAP code of the M2 module, you will either receive a dynpro of M1 if one exists with the same number or a runtime error of `DYNPRO NOT FOUND`.

Neither of these results are really what you want.

Key terms to know

A *module pool* is a specific type of ABAP program that contains dynpros and their associated ABAP modules. As a local object to the program, a module pool can also contain classes or subroutines. You typically use a transaction code to start this type of program. In contrast, a *function group* contains function modules and dynpros with their dialog modules. You typically call a function module to load a function group.

[3] For more information, see SAP Note 2841.

Optimal use of dynpros in ABAP programs

If you want to use dynpros in an application, be sure to follow the guidelines presented in the article series "An insider's guide to writing robust, understandable, maintainable, state-of-the-art ABAP programs" (*SAP Professional Journal*, January/February 2006).* You can achieve a perfect separation of presentation logic and application logic (that is, a "separation of concerns") by following these simple rules:

- Only define dynpros in function groups.
- Do not include any operational statements in dialog modules.
- Define types in the ABAP Dictionary for data that you want to display on a dynpro. You can take advantage of the dynpro support of the data dictionary and also be able to use the type when communicating with the application program.
- For everything else, use class pools and interface pools. Do not use functions groups.

* Part 2 of this article series appears on page 3 of this issue of *SAP Professional Journal*, and Part 3 will be published in an upcoming issue.

Here is the solution: If you have a dynpro that you want to call from a program other than the program in which it is defined, *always* define the dynpro in a function group and *never* in a module pool. Within a function group, the `call screen` statement always calls a dynpro of the function group.

Working with dynpros and ABAP object instances

Dynpros were clearly not designed for object-oriented programming, as object orientation did not exist when they were introduced. They cannot exist in class pools or as attributes of an ABAP object. However, you do not have to avoid dynpros when writing object-oriented programs, as we demonstrate in this final section. As an example, at SAP we used an object-oriented design combined with dynpro technology to implement the New ABAP Debugger.[4] This approach was beneficial because we could write intelligible ABAP Objects code[5] and use proven dynpro technology.

Object orientation means (among many other things) that you can have several concurrent instances of an object. Sometimes you also want to display several instances of an object. But you cannot simply create an instance of a dynpro for an object. Instead, you can only create static dynpros that exist in the context of ABAP module pools and function groups.

You generally do not know how many instances of an object are needed in a program because objects are created dynamically. On the other hand, dynpros are visible objects. If you want to see the same type of information many times on a screen (which roughly corresponds to one ABAP object), you use one line of a table control to display the information for one object and you need only one static dynpro. That is not very difficult to program.

But in some situations you need to have one

[4] The New ABAP Debugger is available starting with SAP Web Application Server 6.40 in SAP NetWeaver '04. For more information, see the article "Introducing the next generation of ABAP debugging — the New ABAP Debugger" in the January/February 2006 issue of *SAP Professional Journal*.

[5] For more on ABAP Objects, see the article "Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Programmer Should Give It a Second Look" in the September/October 2004 issue of *SAP Professional Journal*.

```
NOODLES_DEMO_SHAPE with value range PENNE, FARFALLE, SPAGHETTI
NOODLES_DEMO_COLOR with value range YELLOW, GREEN, RED
NOODLES_DEMO_SIZE with value range SMALL, MEDIUM, LARGE
```

Figure 5 Defining the noodle attributes in the ABAP Dictionary

```
SHAPE type NOODLES_DEMO_SHAPE
COLOR type NOODLES_DEMO_COLOR
SIZE  type NOODLES_DEMO_SIZE
```

Figure 6 Public attributes of the CL_NOODLES_DEMO class

dynpro per object. We constructed a simple example to illustrate the possibilities.

Creating the basic elements for the example

In this example, we wanted to display objects as fields on a dynpro, with each object corresponding to a dynpro. This technique is a simplified version of what the SAP development team did to implement the New ABAP Debugger. We chose a "noodle" as the object. A noodle is defined by attributes for shape, color, and size. For the attributes, we defined three types as domains in the ABAP Dictionary, as shown in **Figure 5**.

To enable the creation of noodle instances, we defined a `CL_NOODLES_DEMO` class with the public attributes shown in **Figure 6**.

Next we wrote an ABAP report named ZDV_NOODLES (**Figure 7**) to create some instances of the `CL_NOODLES_DEMO` class. Each noodles instance has its own set of attributes for shape, color, and size that will be filled in by the `CL_NOODLES_DEMO` class constructor, which we create later in the example.

How can we display the four noodles instances in a dynpro? In order to make a realization possible using dynpro technology, we decided that the `CL_NOODLES_DEMO` class must know that it can have only four instances. (Setting an upper limit on the maximum number of instances is a restriction that in real life is usually not a problem.) In addition, each noodles instance must know which subscreen dynpro to use for display purposes. This information is stored in the `PROGRAM` and `DYNNR` attributes of the noodles instances and is accessed from the flow logic of the dynpro that includes the subscreen dynpros for the noodles instances. This technique, which is necessary because we cannot define subscreen dynpros dynamically, is made possible by restricting the maximum number of instances to four. With these decisions made, we defined the dynpro 100 with four subscreen areas for the four instances and the flow logic shown in **Figure 8** on page 96.

The `PROGRAM` and `DYNNR` attributes of the instanced noodles are accessed via the `call subscreen` statements of the flow logic. Let's look at how these attributes were assigned. We used the Class Builder (transaction SE24) to implement an ABAP class named `CL_NOODLES_DEMO`.[6] Like all classes, `CL_NOODLES_DEMO` has a class constructor, which is a piece of ABAP code that is executed the first time the class is accessed. Here is the appropriate place to define an instance handler. **Figure 9** on page 96

[6] You can also use the Object Navigator (transaction SE80) to implement a class.

Report ZDV_NOODLES Active

```
7   *&---------------------------------------------------------------------*
8
9   report  ZDV_NOODLES.
10
11  data OK_CODE type FCODE.
12  data SHAPE type NOODLES_DEMO_SHAPE.
13  data COLOR type NOODLES_DEMO_COLOR.
14  data SIZE  type NOODLES_DEMO_SIZE.
15
16  data NOODLE_INSTANCE1 type ref to CL_NOODLES_DEMO.
17  data NOODLE_INSTANCE2 type ref to CL_NOODLES_DEMO.
18  data NOODLE_INSTANCE3 type ref to CL_NOODLES_DEMO.
19  data NOODLE_INSTANCE4 type ref to CL_NOODLES_DEMO.
20
21  SHAPE = 'SPAGHETTI'.
22  COLOR = 'GREEN'.
23  SIZE =  'MEDIUM'.
24
25  create object NOODLE_INSTANCE1
26    exporting
27      P_SHAPE = SHAPE
28      P_COLOR = COLOR
29      P_SIZE = SIZE.
30
31  SHAPE = 'FARFALLE'.
32  COLOR = 'YELLOW'.
33  SIZE =  'MEDIUM'.
34
35  create object NOODLE_INSTANCE2
36    exporting
37      P_SHAPE = SHAPE
38      P_COLOR = COLOR
39      P_SIZE = SIZE.
40
41  SHAPE = 'PENNE'.
42  COLOR = 'RED'.
43  SIZE =  'LARGE'.
44
45  create object NOODLE_INSTANCE3
46    exporting
47      P_SHAPE = SHAPE
48      P_COLOR = COLOR
49      P_SIZE = SIZE.
50
51  SHAPE = 'PENNE'.
52  COLOR = 'YELLOW'.
53  SIZE =  'MEDIUM'.
54
55  create object NOODLE_INSTANCE4
56    exporting
57      P_SHAPE = SHAPE
58      P_COLOR = COLOR
59      P_SIZE = SIZE.
60
61  call screen 100.
```

ABAP Ln 60 Col 1 Ch 1 NUM

Figure 7 ZDV_NOODLES report that creates noodles instances

shows the `CLASS_CONSTRUCTOR` method for the `CL_NOODLES_DEMO` class and **Figure 10** (on page 97) shows the class attributes.

In the ABAP Dictionary (transaction SE11), we also defined a table type named `NOODLE_INSTANCES` with a structure named `NOODLE_INSTANCE` that defines

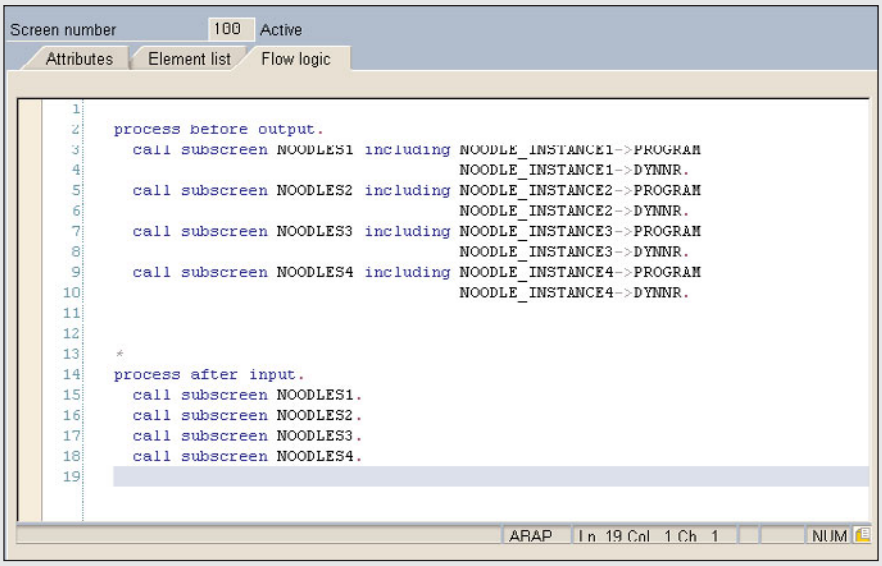


Figure 8 Flow logic of the ZDV_NOODLES 100 dynpro

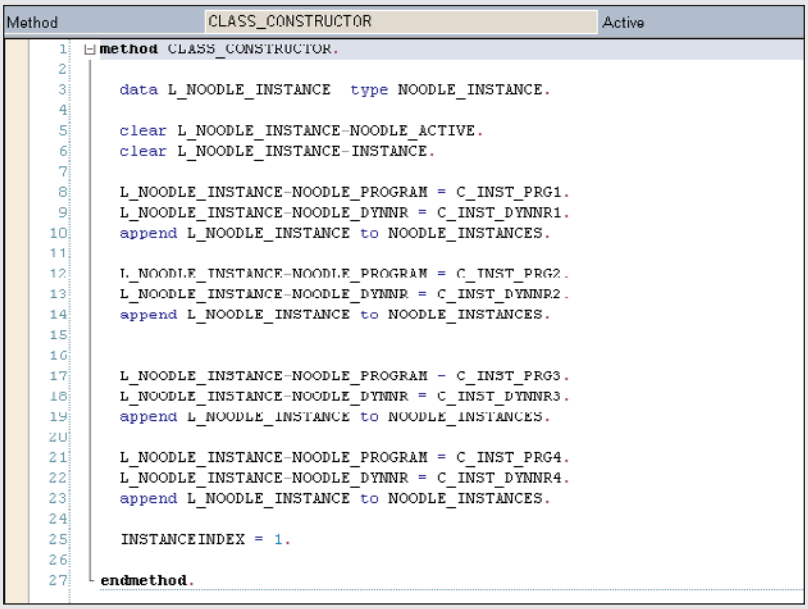
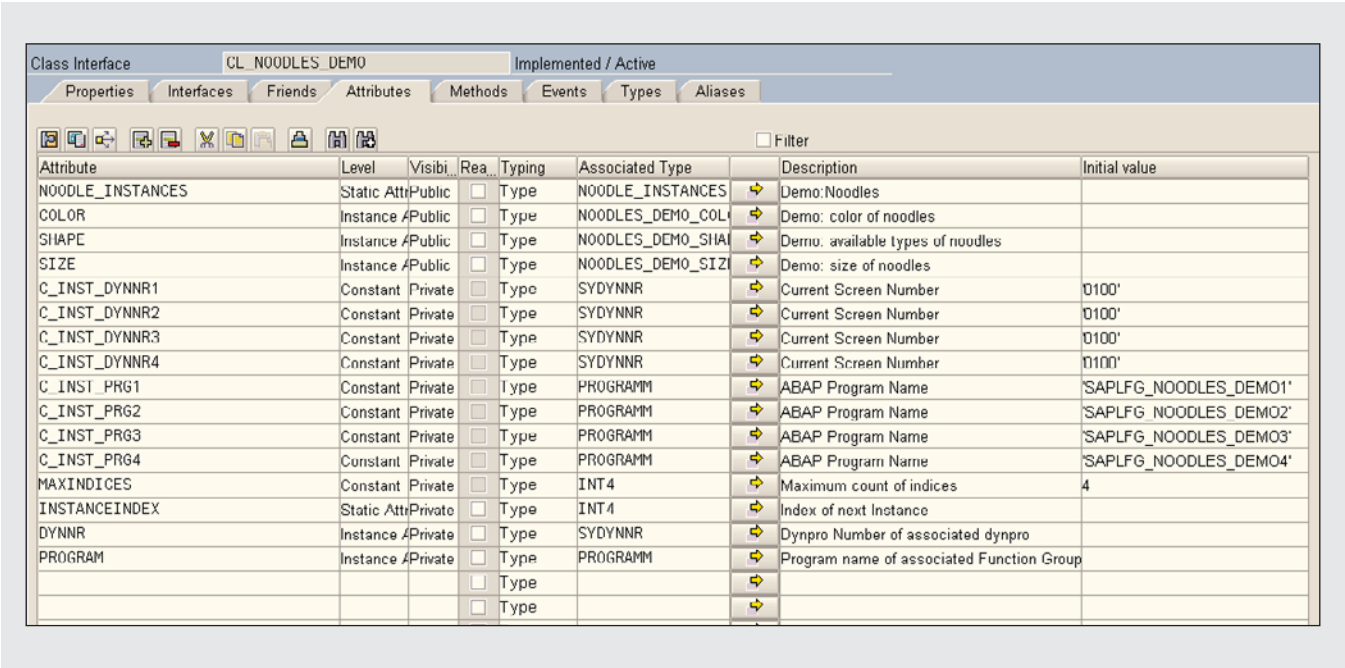


Figure 9 Class constructor method of the CL_NOODLES_DEMO class

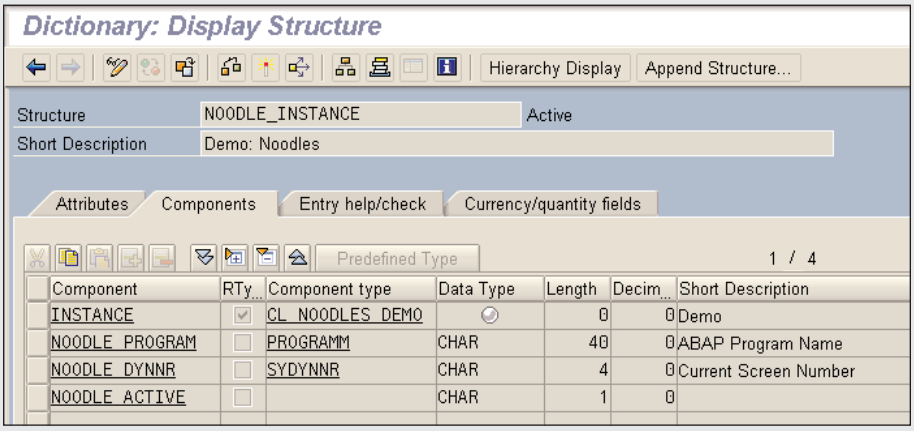
the line type (see **Figure 11**). We use this line type later in this example as the type for the `NOODLE_INSTANCES` static attribute of the `CL_NOODLES_DEMO` class. Each line of this table has four components:

- `INSTANCE` — reference to the object
- `NOODLE_PROGRAM` — function group name
- `NOODLE_DYNNR` — dynpro number
- `NOODLE_ACTIVE` — flag indicating the line is in use



Attribute	Level	Visibi...	Rea...	Typing	Associated Type		Description	Initial value
NOODLE_INSTANCES	Static Attr	Public		Type	NOODLE_INSTANCES		Demo:Noodles	
COLOR	Instance A	Public		Type	NOODLES_DEMO_COL		Demo: color of noodles	
SHAPE	Instance A	Public		Type	NOODLES_DEMO_SHA		Demo: available types of noodles	
SIZE	Instance A	Public		Type	NOODLES_DEMO_SIZ		Demo: size of noodles	
C_INST_DYNNR1	Constant	Private		Type	SYDYNNR		Current Screen Number	'0100'
C_INST_DYNNR2	Constant	Private		Type	SYDYNNR		Current Screen Number	'0100'
C_INST_DYNNR3	Constant	Private		Type	SYDYNNR		Current Screen Number	'0100'
C_INST_DYNNR4	Constant	Private		Type	SYDYNNR		Current Screen Number	'0100'
C_INST_PRG1	Constant	Private		Type	PROGRAMM		ABAP Program Name	'SAPLFG_NOODLES_DEMO1'
C_INST_PRG2	Constant	Private		Type	PROGRAMM		ABAP Program Name	'SAPLFG_NOODLES_DEMO2'
C_INST_PRG3	Constant	Private		Type	PROGRAMM		ABAP Program Name	'SAPLFG_NOODLES_DEMO3'
C_INST_PRG4	Constant	Private		Type	PROGRAMM		ABAP Program Name	'SAPLFG_NOODLES_DEMO4'
MAXINDICES	Constant	Private		Type	INT4		Maximum count of indices	4
INSTANCEINDEX	Static Attr	Private		Type	INT4		Index of next Instance	
DYNNR	Instance A	Private		Type	SYDYNNR		Dynpro Number of associated dynpro	
PROGRAM	Instance A	Private		Type	PROGRAMM		Program name of associated Function Group	
				Type				
				Type				

Figure 10 Attributes of the CL_NOODLES_DEMO class



Component	RTy...	Component type	Data Type	Length	Decim...	Short Description
INSTANCE	✓	CL_NOODLES_DEMO		0	0	Demo
NOODLE_PROGRAM		PROGRAMM	CHAR	40	0	ABAP Program Name
NOODLE_DYNNR		SYDYNNR	CHAR	4	0	Current Screen Number
NOODLE_ACTIVE			CHAR	1	0	

Figure 11 ABAP Dictionary structure to be used by the NOODLE_INSTANCES internal table

As you see in the `CLASS_CONSTRUCTOR` method (Figure 9), we initialized the `NOODLE_INSTANCES` table with four lines containing values for the `NOODLE_PROGRAM` and `NOODLE_DYNNR` components. The `NOODLE_ACTIVE` and `INSTANCE` components remain empty. For simplicity, we defined the values for the program name (C_INST_PRG1 to C_INST_PRG4) and the numbers of the subscreen dynpros (C_INST_DYNNR1 to C_INST_DYNNR4) as simple constant attributes of the class (see the first two columns in Figure 10). The assigned program name SAPLFG_NOODLES_DEMOx (where x = 1 to 4) means that the dynpros belong to the function group FG_NOODLES_DEMOx (see the last column in Figure 10). You could create a separate class to handle the function groups that contain the dynpros (in other

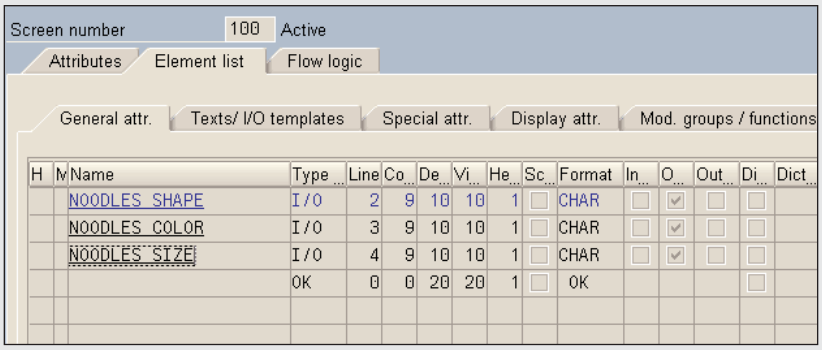


Figure 12 Subscreen dynpro 100 for the FG_NOODLES_DEMO function groups

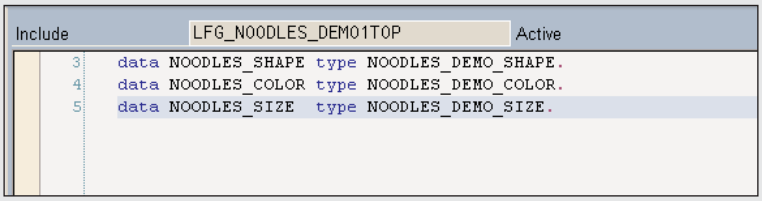


Figure 13 Defining ABAP variables in a function group for the corresponding dynpro fields

words, by implementing a subscreen handler). But our simple example works just as well with this basic definition.

Based on the program name C_INST_PRGx (where x = 1 to 4), you see that we adopted the principle of using a separate ABAP program of type function group for each noodle instance — FG_NOODLES_DEMOx (where x = 1 to 4). Each function group contains a subscreen dynpro 100 with fields defined for color, shape, and size, as shown in the Screen Painter in **Figure 12**.

To make the fields match the types of the noodle attributes, we used the ABAP Dictionary (transaction SE11) to define all of the fields as ABAP Dictionary types (`NOODLES_DEMO_COLOR`, `NOODLES_DEMO_SHAPE`, and `NOODLES_DEMO_SIZE`). We also defined similar ABAP variables in the function group and as instance attributes (see the color, shape, and size lines in Figure 10 and **Figure 13**). The ABAP variables that are defined in the function group correspond to the fields that are defined on the dynpro; they both have the same type.

Creating the object instances

With all of the necessary elements prepared, we can now write the `CONSTRUCTOR` method (see **Figure 14**) of the `CL_NOODLES_DEMO` class. This method is executed when an object instance is created and therefore is the correct place to fill in the values of the noodle attributes.

The `CONSTRUCTOR` method has parameters for the noodle attributes `P_SHAPE`, `P_COLOR`, and `P_SIZE`, as well as an exception for the case that no more noodles instances can be created (Figure 14, line 16). We defined a static class attribute `INSTANCEINDEX` to know how many instances exist at the moment, which tells us which line of the `NOODLES_INSTANCES` table to use for creating the next instance. `MAXINDICES` is a constant attribute of the `CL_NOODLES_DEMO` class (refer back to Figure 10) with a value of 4, which is the maximum possible number of instances. If creating an instance is allowed, we assign the parameter values (that is, the noodle attributes) to the fields of the internal table (Figure 14, lines 8-10) and mark the table entry as active (line 11). Notice that we also

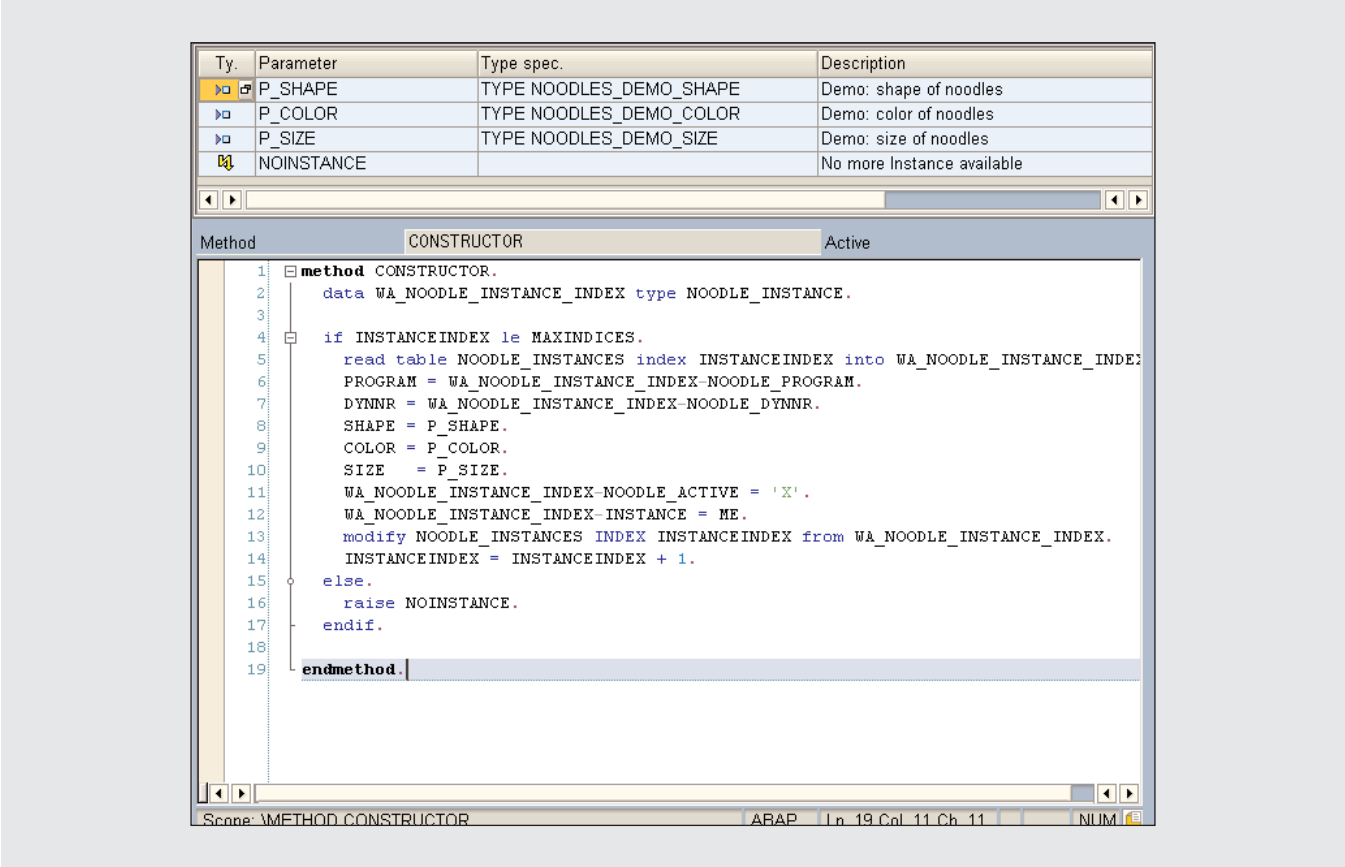


Figure 14 Constructor method of the CL_NOODLES_DEMO class

keep a reference to the instance, which we need later in order to access the noodle attributes from the function groups.

Now the ZDV_NOODLES report that we created back in Figure 7 nearly works. We defined the ABAP Dictionary structures. We created four noodles instances with the noodle attributes for shape, color, and size. We called dynpro 100 of the ZDV_NOODLES program to display the dynpro (Figure 7). The `call subscreen` statements of the ZDV_NOODLES 100 dynpro worked (Figure 8), so all of the noodles instances exist with their `PROGRAM` and `DYNNR` attributes set appropriately.

But we still have one final task. The values for the shape, color, and size attributes are stored in the `CL_NOODLES_DEMO` class, but not yet known to the FG_NOODLES_DEMOx function groups. To achieve this purpose, we need to obtain these values at PBO of the subscreen dynpro of each function group. Using either the Screen Painter or the Object Navigator, we inserted a PBO module `FILLFIELDS` in the subscreen flow logic (see **Figure 15** on the next page) for each function group to perform the task.

Figure 16 (on the next page) shows the `FILLFIELDS` module for the FG_NOODLES_DEMO1 function group. It calls a `FILLFIELDS` method, which assigns the `NOODLE_INSTANCES` table to an `ALL_NOODLES` local variable (line 24) and then reads a line of the table variable (line 25). Remember that the `NOODLE_INSTANCES` table is a static public attribute of the `CL_NOODLES_DEMO` class. For simplicity, we decided that each function group should already know its index in the table. Therefore, the FG_NOODLES_DEMO1 function group uses index 1, FG_NOODLES_DEMO2 uses index 2, and so on.

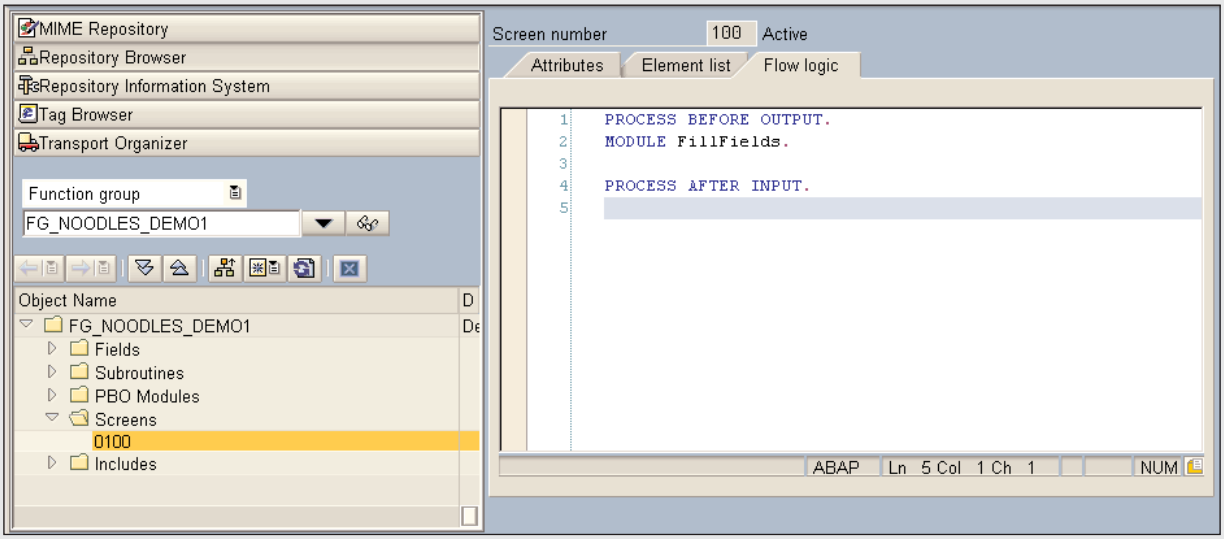


Figure 15 PBO module of the subscreen dynpro to obtain noodle attributes

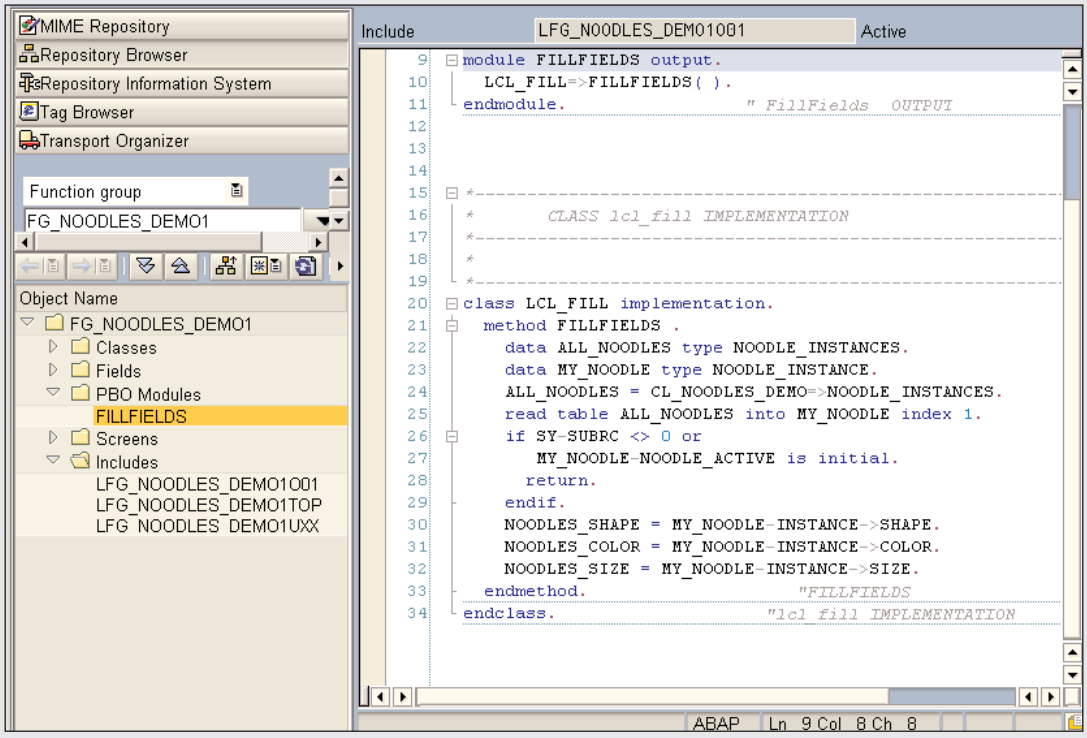


Figure 16 Getting attribute values from the ABAP object instance

Once the `FILLFIELDS` method has a line of the table, it fills the variable fields of the function group with the values obtained from the object instance of the `CL_NOODLES_DEMO` class (lines 30-32).

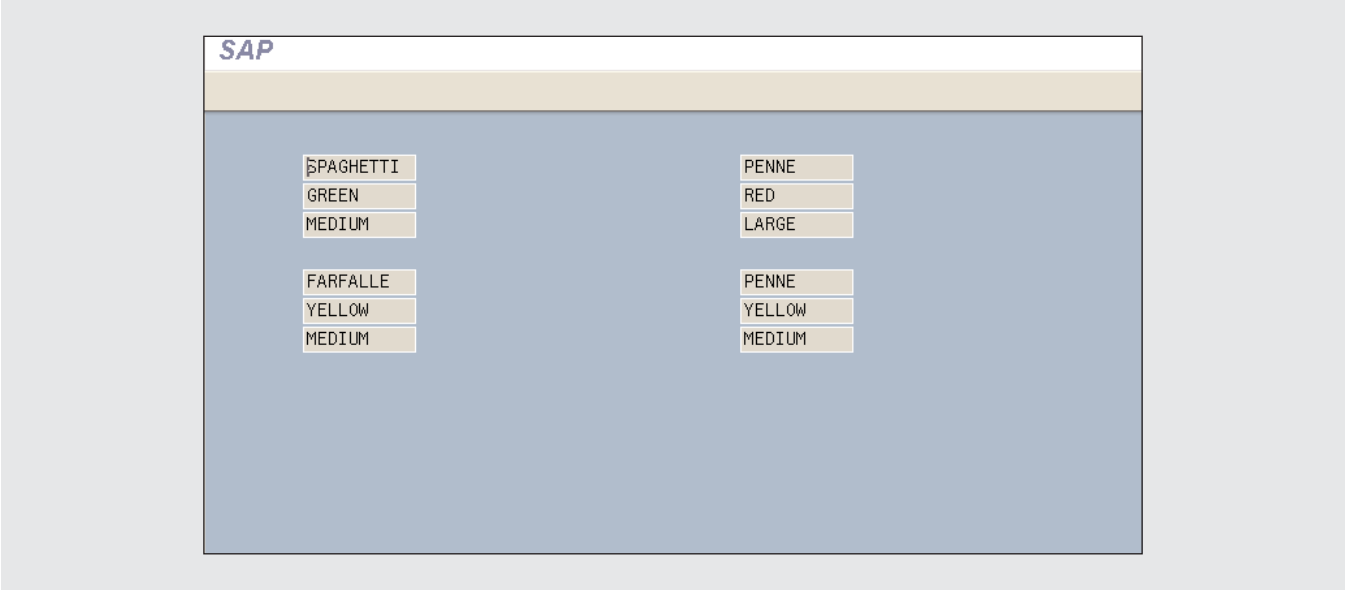


Figure 17 Output of the ZDV_NOODLES report

The results — what happened to the screen and why

Obviously this example is just a demonstration with many aspects that could be made more complex and dynamic. But we simply wanted to give you a little motivation to write programs with ABAP Objects and dynpros. Using this basic technique, you can store all data object instances of an ABAP class as usual and display them on dynpros. The results of our example are shown in **Figure 17**.

The four sections that you see on the screen show the values of the four object instances that we created in the ZDV_NOODLES report in Figure 7 (`NOODLE_INSTANCE1`, `NOODLE_INSTANCE2`, `NOODLE_INSTANCE3`, and `NOODLE_INSTANCE4`). For example, in the upper left corner, the `NOODLES1` subscreen area displays the `NOODLE_INSTANCE1` instance of the `CL_NOODLES_DEMO` class, as defined in lines 21-29 of the report. To recap, here is what we did:

- In line 25 of the report, class `CL_NOODLES_DEMO` was called for the first time, and so the class constructor method of the `CL_NOODLES_DEMO` class (Figure 9) was executed and the `NOODLE_INSTANCES` internal table was prepared.
- Also in line 25, the constructor method of the `CL_NOODLES_DEMO` class was called with the parameters `P_SHAPE = 'SPAGHETTI'`, `P_COLOR = 'GREEN'`, and `P_SIZE = 'MEDIUM'`. These values were assigned to the instance attributes (lines 8-10 of the constructor method) and a reference to the instance was entered in the `NOODLE_INSTANCES` table (line 12 of the constructor method).
- In line 61 of the report, the dynpro 100 was called. The flow logic of this dynpro called the subscreen dynpro for the `NOODLE_INSTANCE1` instance (Figure 8, lines 3-4).
- At PBO of the subscreen dynpro, the necessary values were fetched from the object instance (Figure 15) and assigned to the fields that are displayed on the dynpro.

Conclusion

We hope you have seen from this two-part article series that programming with dynpros is still a great way to easily build a robust user interface for an ABAP program if you use the SAP GUI technology. This technology has been proven over time and in thousands of applications, and as you have seen is quite full-featured:

- In the first article, we presented some design options for building better dynpros. We discussed various techniques for dynamically changing fields on a dynpro, as well as how to manage field transports, the effects of resizing and compression, and message handling.
- In the second article, we began by discussing the coupling of classic dynpros and front-end controls. Here you learned the most important principles for combining these technologies to avoid your application behaving in unintended ways.
- We then switched to the ABAP programmer point of view with some tips for avoiding hard-to-find bugs when working with dynpros.
- Finally, we concluded with an example that hopefully inspires you to combine dynpro programming and object orientation.

Armed with this information, your dynpro-based user interfaces will be much improved because you now know exactly how to define and manipulate dynpros at runtime. Further, you know how to avoid common pitfalls such as loading the wrong dynpro or transporting the wrong fields for currency fields, which will save you precious time and effort in problem identification and resolution. And you have seen how easy it is to write clear, understandable ABAP programs combined with dynpros in order to achieve robust dialog programs without much effort.

Doris Vielsack received her degree in computer science from the University of Karlsruhe, Germany, in 1989. From 1989 to 2001, she worked in the public service sector of Karlsruhe at a computer center, where she started by developing user interfaces for special output devices, and later assumed responsibility for parts of the software and hardware infrastructure of the department. Doris joined SAP in 2001, and currently works in development support as part of the SAP NetWeaver ABAP QM group. She is responsible for the development of the Dynpro Tool of the New ABAP Debugger. You can reach Doris at doris.vielsack@sap.com.

Arndt Rosenthal studied process engineering at the Institute of Technology in Merseburg, Germany, where he received his degree in 1985. From 1985 to 1990, he worked on computer-aided methods for the optimal design of experiments to determine nonlinear models of technical processes in the chemical industry. Arndt joined SAP in 1990. After working as a consultant for about a year, he became a member of the Customizing Tools development team, where he was responsible for the development of the Extended Table Maintenance Framework. In 1998, Arndt joined the dynpro processor development team, which became the SAP NetWeaver ABAP UI Services group. He is currently jointly responsible for the development and maintenance of the dynpro processor. Arndt can be reached at arndt.rosenthal@sap.com.

