Dynamic Documents: Completing the New Face of Reporting

Jonathan Maidstone

Jonathan Maidstone joined SAP in 1996 as a translator for the ABAP Workbench and ABAP language groups. He is now a technical author, writing documentation for the SAP Control Framework, Desktop Office Integration, and SAP DCOM Connector teams, as well as assisting with their training and rollout activities.

(complete bio appears on page 102)

In the past, screen programming and list programming occupied two different worlds. In developing an application, you had to weigh the relative advantages and disadvantages of each technique, pick the one you thought best, and stick to it. The only way of incorporating list output into a transaction was either to use the `LEAVE TO LIST-PROCESSING` command, or to call a separate executable program using `SUBMIT`, neither of which was a particularly convenient option.

The advent of controls programming changed this approach. The ALV Grid Control (introduced in Release 4.6A), for example, allows you to display a list in tabular form in a control container. As part of the Control Framework,[1] it is fully integrated with screen programming, allowing you to place the list and traditional graphical screen elements (pushbuttons, tabstrip controls, and so on) side by side on the same screen. This can be extremely useful from a usability point of view, since the ability to combine list output with the flexible design possibilities afforded by screens allows you to streamline your applications by reducing the number of different screens you need.

So you have the ALV Grid Control, which allows you to replace tabular lists with control-based output on the screen, and the SAP Tree Control (used, for example, for the object list in the Object Navigator of the ABAP Workbench from Release 4.6A), which you can use to do the same thing for hierarchical lists. But something is still missing — how can you display *unstructured* output in a control-based environment? Say, for example, you want to create an application in which you have a

[1] If you would like to learn more about the SAP Control Framework, I would recommend SAP's book *Controls Technology*, written by a team of experienced SAP authors, and available online from **http://shop.sap.com**.

list with a free-form header — maybe to display the name and address of a customer at the top of a report about his or her various orders. The output in this case is neither tabular nor hierarchical, so the controls I have just mentioned are not going to provide us with any solutions. However, it used to be a relatively simple task in traditional lists — you could use `WRITE` and its additions, and the `SKIP` statement to place output almost anywhere on the screen or on a printed list.

One possible solution is to use the SAP HTML Viewer Control on a screen and load a template from the SAP Web Repository (introduced as part of SAP Web Reporting in Release 3.1G). The HTML Viewer Control contains a method[2] that enables you to do just that. If you use this approach, however, you are bound to a static structure and layout of your template, so you cannot guarantee that your HTML document will match the look and feel of the rest of the system once the users have started to change their color schemes. You would also have to create a new template every time you needed such an HTML fragment. It would be much more practical if you could create an HTML Viewer instance *and its content* on the fly in the program and know that they would always have the right styles and colors to fit in with the rest of the system. From Release 4.6C onward, you can, using *dynamic documents*.

A dynamic document is an HTML document that you can construct in a program using ABAP methods. Dynamic documents are based on a set of global ABAP classes whose methods allow you to add elements to a document in much the same way that you would use `WRITE` statements to construct a list.

Figure 1 shows dynamic documents in action — in the R/3 Travel Management application. The left-hand frame of the screen is a very simple dynamic document. It incorporates only a few elements — text, icons, and underlines, to be specific.

The dynamic documents you create may contain:

- **Text**
- **Icons**
- **Underlines**
- **Links**
- **Pictures**
- **Tables**
- **Forms**, within which you can add input fields, pushbuttons, and dropdown list boxes

Dynamic documents are fully compatible with the three SAP GUI environments — Windows, Java, and HTML — and are designed to automatically take into account, and operate in accordance with, a user's settings (graphical or classic design, color scheme, and so on). This is not the case with HTML templates that you create yourself, which don't necessarily blend in with the overall SAP look and feel. You will also find dynamic documents to be a better fit with SAP screen processing than standalone HTML templates, because the contents of input fields in a dynamic document are always available in the PAI event of the relevant screen as an attribute of the corresponding proxy object in ABAP. If you used your own HTML template instead, you would have to submit the relevant form before you could access the field contents.

This article provides an introduction to dynamic documents — how they fit into screens in the SAP system, and how you can use them for both simple output of text, pictures, or links, and for more complicated documents involving tables and forms for user interaction — in short, a brief overview of how you can move away from traditional list-bound output to create more compact and attractive user interfaces.

Dynamic Documents: A New Reporting Technique

Figure 2 provides a summary of the changes that have occurred in reporting over recent releases, starting with the classic list, and concluding with a state in Release 4.6C where it is possible to transfer most conventional list functions into a unified screen environment.

[2] Method `load_html_document` of class `cl_gui_html_viewer`.

Figure 1 A Dynamic Document in Use in the R/3 Travel Management Application

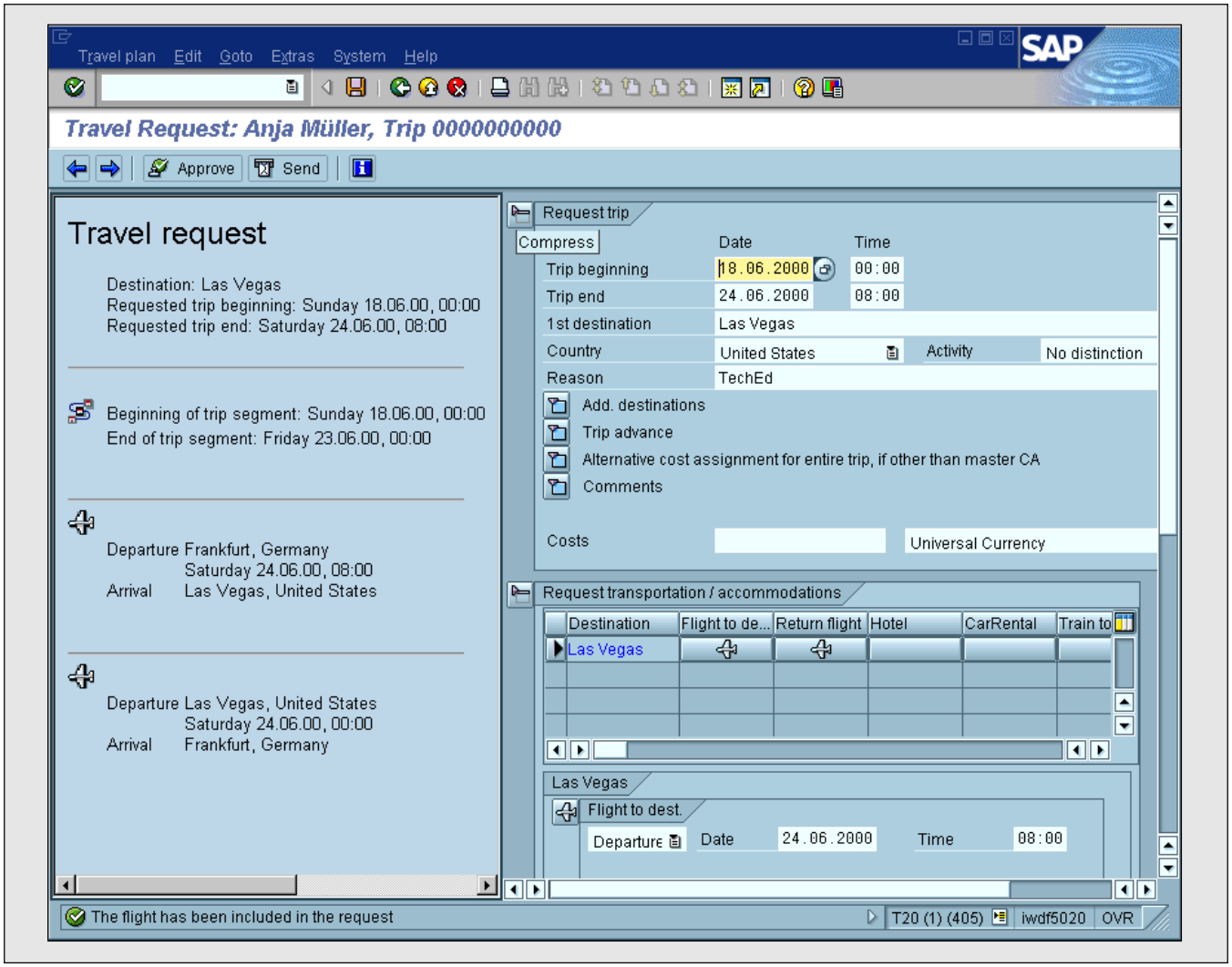
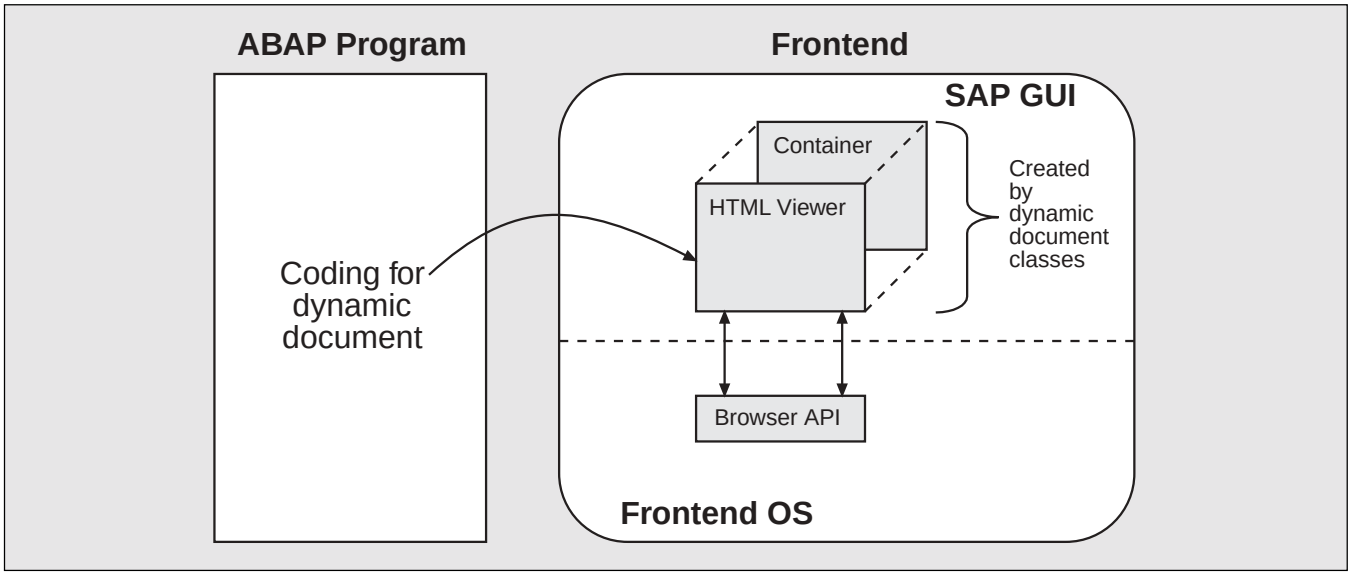


Figure 2 Summary of Changes in Reporting Techniques Since Release 4.6A

Release	Technique	Advantages	Disadvantages
< 4.6	"Classic" list	Flexible output possibilities.	"Standalone" solution — could not be incorporated into screen programming.
4.6A	SAP Control Framework, including: • ALV Grid • SAP Tree Control	Practical solution for displaying tables and hierarchies in screens. More modern look and feel.	Still no solution for free-form input.
4.6C	Dynamic documents	Brings the flexible output possibilities of the list to the screen environment. Styles automatically applied reflecting the SAP standard look and feel.	Not all UI elements supported in HTML are available.

Figure 3 How Dynamic Documents Are Incorporated into Screens



Dynamic documents function as an extra layer between you, the programmer, and the SAP HTML Viewer Control. Instead of creating an HTML Viewer instance and then supplying it with the content you want to display, you start with the content. The dynamic document classes then convert this content into an HTML document, and create instances of an SAP Container Control and the SAP HTML Viewer to display it on the screen. The HTML Viewer uses the API of the underlying Web browser on whatever frontend your end user has installed (see **Figure 3**).

Creating a Simple Dynamic Document

A dynamic document, its constituent areas, and the elements the document may contain (text, icons, pictures, links, underlines, tables, forms) are represented by the set of global classes illustrated in **Figure 4**.[3]

[3] Only the most significant are illustrated. There is a further set of classes, derived from `cl_dd_form_element`, which represent pushbuttons, input fields, and dropdown list boxes. We will discuss these later.

There are six steps to creating a simple dynamic document:

1. Create the screen area.
2. Declare the document reference.
3. Create the document instance.
4. Fill the document.
5. Merge the document.
6. Display the document.

Contained in the following descriptions of these steps is the coding necessary to produce the sample dynamic document shown in **Figure 5**.

Step 1: Creating the Screen Area

The first thing to do when you create a dynamic document is to decide where you want to display it. There are two possibilities: you can either create a custom container area using the Screen Painter, and tell the dynamic document instance to use that, or you can instantiate any SAP Container Control, and pass its reference variable to the dynamic document. The first method is the simpler of the two, and is perfectly

Figure 4 Class Hierarchy of the Main Dynamic Document Classes

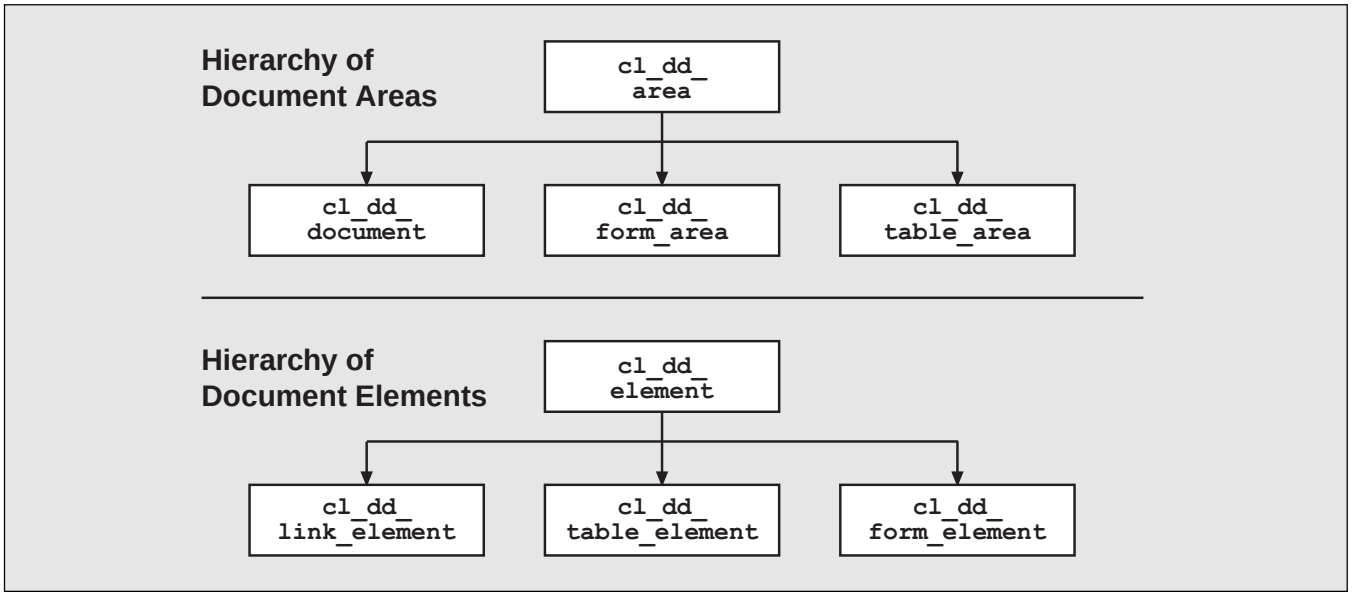


Figure 5 Display from a Simple Example Program

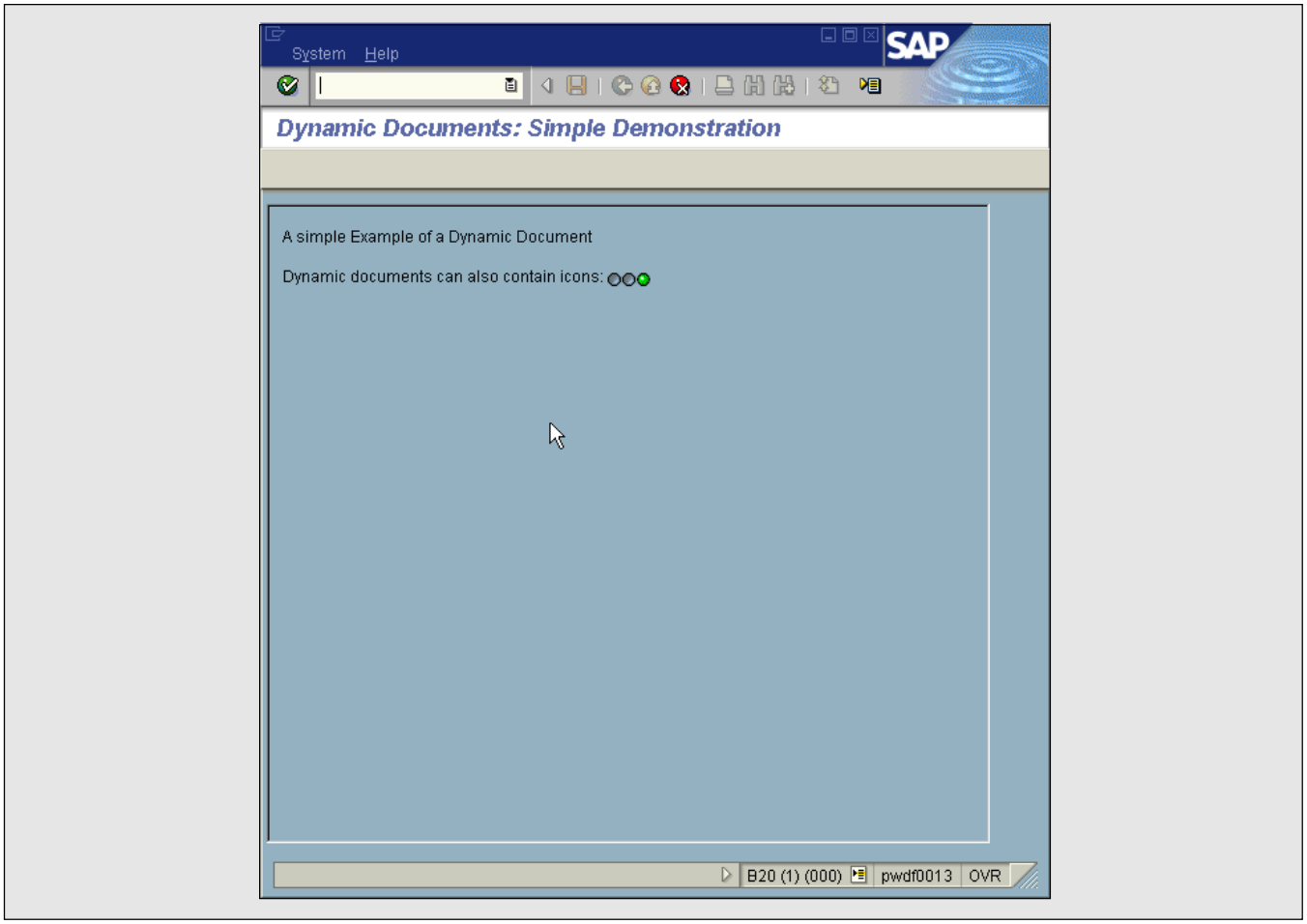
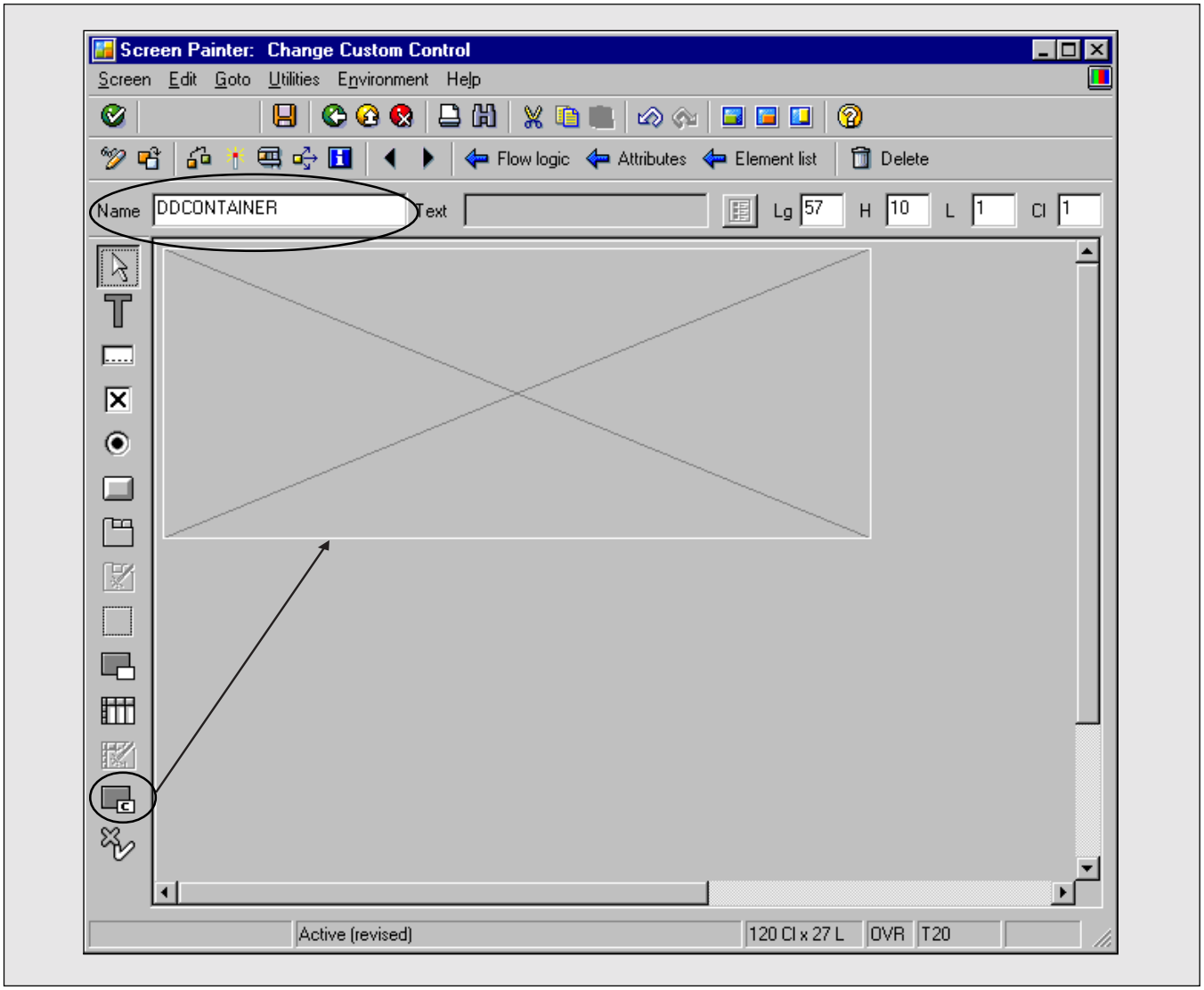


Figure 6 Creating a Custom Container in the Screen Painter



adequate as long as you want to display your document in a screen area. If, on the other hand, you want to place the document in a different area — for example, a dialog box container or a docking container control — you must use the second method.

To create a custom container area, start the Screen Painter, choose the Custom Container icon from the element toolbar, and place the container on the screen, as shown in **Figure 6**. Assign a name to the container — you will need it later when you tell your program where to display the dynamic document.

Step 2: Declaring the Document Reference

When you create a dynamic document, you work with references to one or more objects. The one object you always need for every dynamic document is a reference to the class `cl_dd_document`. I will refer to this as the *main document instance*. For more complex dynamic documents, you will need additional instances. For this first simple example, however, you only need to declare a reference to `cl_dd_document` as follows:

```
DATA mydocument TYPE REF TO
cl_dd_document.
```


Step 3: Creating the Document Instance

Once you have declared your document reference, you can instantiate it, and then use the instance to fill the document. You should perform these steps in the PBO event of the screen on which you are going to display the document. To create the instance, use the statement:

```
CREATE OBJECT mydocument.
```

Step 4: Filling the Document

You can now call the methods of your main document instance to fill the document line by line. You can do this using the methods `add_text`, `add_picture`, `add_icon`, and `add_link`. Refer to the Class Builder or the documentation[4] for precise details of these methods and their parameters.

Some examples of method calls include:

```
CALL METHOD mydocument->add_text
  EXPORTING text = 'A simple
  example of a dynamic document'.

CALL METHOD mydocument->new_line.

CALL METHOD mydocument->add_text
  EXPORTING text = 'Dynamic
  documents can also contain icons: '.

CALL METHOD mydocument->add_icon
  EXPORTING sap_icon =
  'ICON_GREEN_LIGHT'.
```

Here you can also set the formatting of the dynamic document text and specify the targets of any links.

In a `WRITE` statement, you could use the `FORMAT` addition to set the color or style of your output. To set styles and colors in a dynamic document, use the `sap_style` and `sap_color` parameters in the `add_text` method. The values that they can take are defined as static constants of the class `cl_dd_area`.

When you set a link in a dynamic document, it can have one of three targets:

- **A target within the same document:** To do this, you must set a target within the document by calling the `add_link` method at the appropriate position and setting the parameter `destination_in_doc_pos` to "`X`".
- **A URL:** To do this, enter its name in the URL parameter. The system displays the URL in the same container control. Note that if you want to return to the dynamic document, you must program your own navigation. This is only possible if you already have a handle to the HTML Viewer, that is, if you reuse an existing HTML Viewer instance (that you created yourself) when you display the dynamic document.
- **An event:** Instead of specifying a static target for a link, you can trigger an event when the user clicks it. To do this, you receive the `link` parameter of the `add_link` method into a reference variable with the type of class `cl_dd_link`. You can then register an event of a local class for this object, which will be called when the link is clicked. In the method implementation, you specify what should happen.

Step 5: Merging the Document

Once you have finished creating the document, you must merge it. When you merge the document, the system takes all of its constituent parts and combines them into the final HTML document that will be displayed on the screen. To do this, you use the `merge_document` method of the main document instance:

```
CALL METHOD
mydocument->merge_document.
```

[4] Although the classes are fully functional in Release 4.6C, it was not possible to produce full documentation until Release 4.6D. If you have a 4.6C system and would like the full documentation, e-mail me at jonathan.maidstone@sap.com, and I will send you a copy.

Step 6: Displaying the Document

The final step in creating a dynamic document is to display it. As I have already mentioned, you have the choice between displaying it in a custom container (in which case the system creates the SAP Custom Container Control and the SAP HTML Viewer Control for you), or in an existing container control (in which case the system only has to create the SAP HTML Viewer Control). Use the `display_document` method of the main document instance. To display the document in a custom container area that you have created in the Screen Painter, pass the container name to the `CONTAINER` parameter. To display it in an existing SAP Container Control, pass the reference variable to the control in the `PARENT` parameter.

In our example, the method call would be:

```
CALL METHOD
mydocument->display_document
   EXPORTING container =
   'DDCONTAINER'.
```

Although very simple, the example should have given you a feel for how dynamic documents work. The document object acts an equivalent of the traditional ABAP list buffer, storing your output until you finally display it. The entire coding is shown in **Listing 1**.

Creating More Complex, Interactive Dynamic Documents

What we have just seen is a very simple example of a dynamic document, but let us now look at how you can create more complex, interactive documents.

Adding More Complex Elements to a Document

There are two kinds of complex elements that you can place within a dynamic document — tables and forms. To create a table or form, you use the `add_table` or `add_form` method of the main document instance, each of which returns an object reference to the relevant kind of element. To fill the table or form, you work with the methods of this instance, and *not* with the main document instance (in other words, you are constructing a second subdocument).

Once you have finished creating the main document and your table or form, you call the `merge_document` method of the main document instance. You need to do this because, internally, the contents of each part of the document (the main document, the table, and the form) are stored separately — the document contents are stored in an internal table that is an attribute of the document object, and

Listing 1: Coding for the Simple Dynamic Document Example

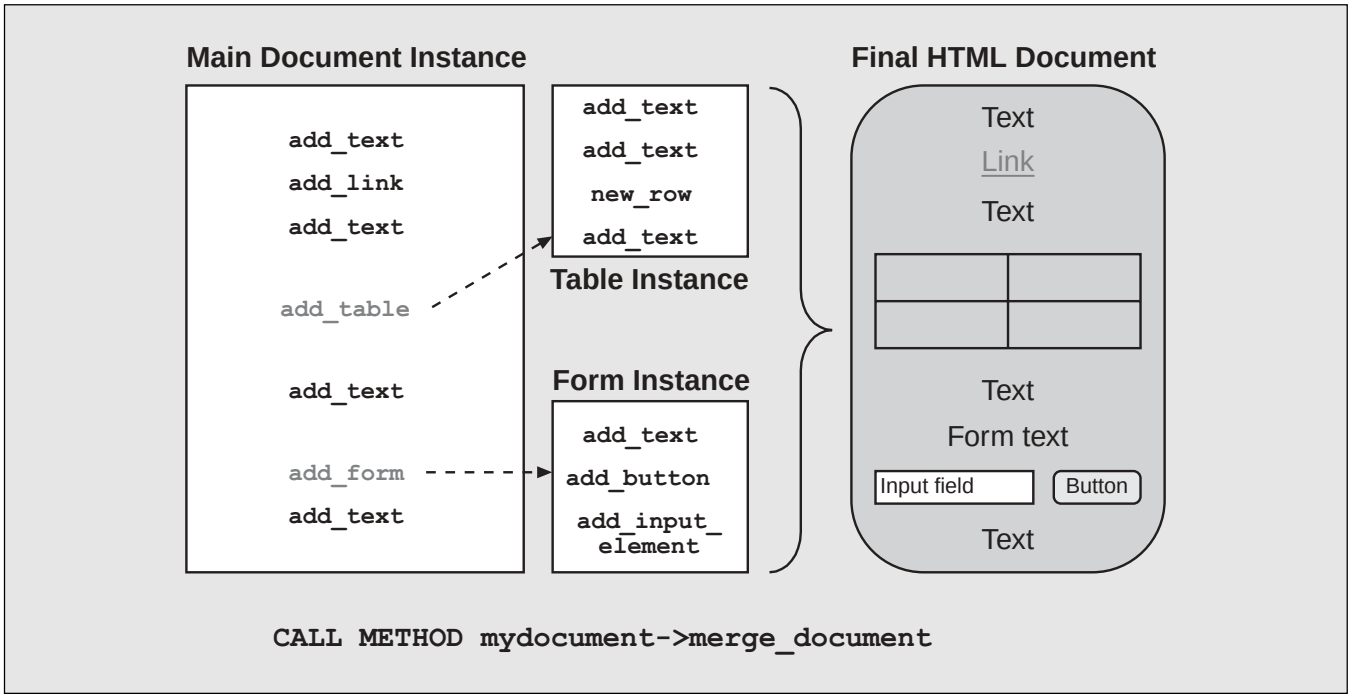
```
REPORT  jm_dynamic_documents_demo.

DATA: container TYPE REF TO cl_gui_custom_container,
      document TYPE REF TO cl_dd_document.
DATA: ok_code TYPE sy-ucomm,
      save_ok TYPE sy-ucomm.

START-OF-SELECTION.
  CALL SCREEN 100.
*&---------------------------------*
*&      Module  create_document  OUTPUT
*&---------------------------------*
*       text
*----------------------------------*
```

(continued on next page)

(continued from previous page)

```
MODULE create_document OUTPUT.

  CHECK document IS INITIAL.
  CREATE OBJECT document.
  CALL METHOD document->add_text
       EXPORTING text = 'A simple Example of a Dynamic Document'.
  CALL METHOD document->new_line.
  CALL METHOD document->new_line.
  CALL METHOD document->add_text
       EXPORTING text = 'Dynamic documents can also contain icons:'.
  CALL METHOD document->add_icon
       EXPORTING sap_icon = 'ICON_GREEN_LIGHT'.
  CALL METHOD document->merge_document.
  CALL METHOD document->display_document
       EXPORTING  container = 'CONTAINER100'.
ENDMODULE.                 " create_document  OUTPUT
*&---------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------*
*       text
*---------------------------------*
MODULE user_command_0100 INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
  CASE save_ok.
    WHEN 'CANCEL'.
      LEAVE PROGRAM.
    WHEN OTHERS.
  ENDCASE.
ENDMODULE.                 " USER_COMMAND_0100  INPUT
*&---------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&---------------------------------*
*       text
*---------------------------------*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'BASIC'.
  SET TITLEBAR '100'.
ENDMODULE.                 " STATUS_0100  OUTPUT


PROCESS BEFORE OUTPUT.

MODULE create_document.
MODULE status_0100.

PROCESS AFTER INPUT.

MODULE user_command_0100.
```

Figure 7 Structure of a More Complicated Dynamic Document

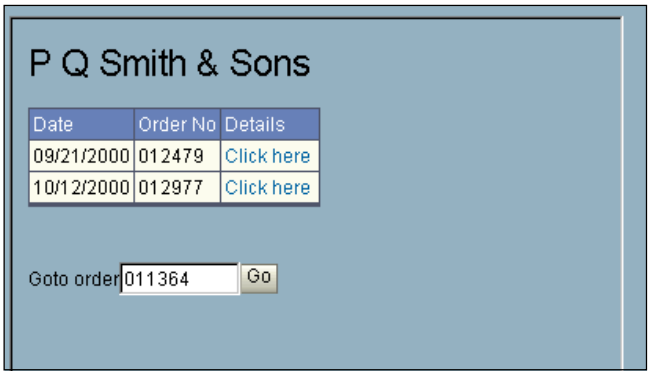


likewise, the table and form components are stored in internal tables belonging to their respective objects. All the main document knows is the names of these other instances. The `merge_document` method takes the content of all the internal tables and combines them into the final HTML document that will appear on the screen, as shown in **Figure 7**.

The screen fragment represented by the pseudocode in Figure 7 could look something like **Figure 8** at runtime. (Only the dynamic document is illustrated. Around it could be any combination of classic screen elements and controls.)

In the next section, I will show how you can build a similar document and make it properly interactive.

Figure 8 Dynamic Document Containing a Table and a Form



Making Documents Genuinely Interactive

Now that you have seen how you can create a table or form in a dynamic document, the only thing that remains is to examine how you can react to user interaction with its elements. The key to this lies in using ABAP Objects events, and methods of local classes to react to them.

Whenever you add a link, pushbutton, input field, or dropdown list box to a dynamic document, the method that you use to do so returns a reference to that element (represented by an instance of the

Figure 9 Events That Can Be Triggered for Each Added Element

Method	Adds a …	Returns a reference to …	Has these events
add_button	Pushbutton	cl_dd_button_element	clicked (user clicked)
add_input_element	Input field	cl_dd_input_element	entered (user pressed “Enter”) help_f1 (user pressed F1)
add_select_element	Dropdown list box	cl_dd_select_element	selected (user selected an entry)
add_link	Link	cl_dd_link_element	clicked (user clicked the link)

relevant class). If you receive this reference in your program, you can register an event handler method for it, that is, a method that will be called when the event is fired. You write this method in a local class in your program. The events that can be triggered for each kind of element are summarized in **Figure 9**.

Example

The coding in **Listing 2** illustrates a very simple example. It produces a dynamic document containing nothing more than an input field and a pushbutton on a form. When the user clicks the pushbutton, the contents of the input field are transferred to an input/output field on the screen. There are two important lessons in the example — firstly how you can use an event to react to a button click, and secondly the fact that within the event handler method, you can find out the contents of any input field (or dropdown list box) on the form from the `value` attribute of the relevant object instance (in this case, `textfield->value`).

Listing 2: Coding for a Dynamic Document with an Input Field and Pushbutton on a Form

```
REPORT ZDYNAMICDOCUMENT.
DATA: mydocument TYPE REF TO cl_dd_document,
      myform TYPE REF TO cl_dd_form_area,
      mybutton TYPE REF TO cl_dd_button_element,
      textfield TYPE REF TO cl_dd_input_element.
DATA text(32) TYPE c.
CLASS lcl_event_handler DEFINITION.
PUBLIC SECTION.
CLASS-METHODS on_clicked FOR EVENT clicked OF cl_dd_button_element.
ENDCLASS.
CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_clicked.
text = textfield->value.
ENDMETHOD.
ENDCLASS.
```

(continued on next page)

(continued from previous page)

```
START-OF-SELECTION.
CALL SCREEN 100.
*-----------------------------------*
* PBO MODULE CREATE_DD
*-----------------------------------*
MODULE create_dd OUTPUT.
CREATE OBJECT mydocument.
CALL METHOD mydocument->add_form
    IMPORTING formarea = myform.
CALL METHOD myform->line_with_layout
     EXPORTING start = 'X'.
CALL METHOD myform->add_input_element
   IMPORTING input_element = textfield.
CALL METHOD myform->add_button
    EXPORTING label = 'Show text'
    IMPORTING button = mybutton.
CALL METHOD myform->line_with_layout
     EXPORTING end = 'X'.
SET HANDLER lcl_event_handler=>on_clicked FOR mybutton.
CALL METHOD mydocument->merge_document.
CALL METHOD mydocument->display_document
     EXPORTING container = 'DDCONTAINER'.
ENDMODULE.
```

If you try this program in your own system, you will need to create screen 0100. In the graphical layout, create a custom container called `DDCONTAINER` and an input/output called `TEXT`. In the flow logic, enter the following statement in the PBO event:

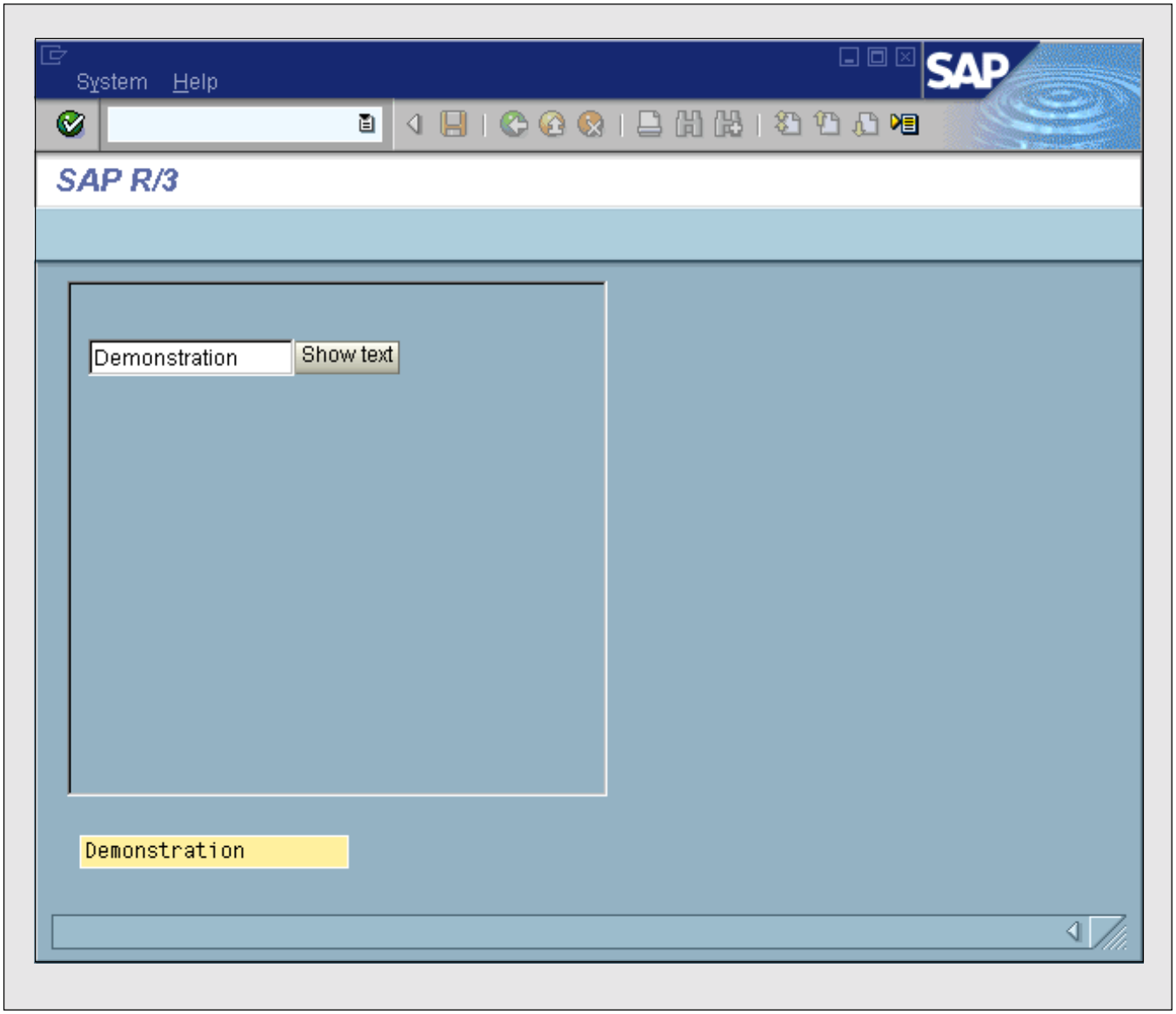
```
MODULE create_dd.
```

When you run the program, your screen should look like **Figure 10**.

When you click the **Show text** button, the text from the input field in the dynamic document is transferred to the screen field.

The program begins by declaring the various reference variables I will need to create my dynamic document. I then declare my local class — `lcl_event_handler`. Local classes in ABAP have two parts: a definition part, in which I declare the class components (methods, attributes, events) and their visibility; and an implementation part, in which I write the actual coding of my methods. The `lcl_event_handler` class has a single method called `on_clicked`, which is also registered as an event handler for the `clicked` event of our button class `cl_dd_button_element`. It is a static method, which means that it can be called even if the class has not been instantiated. In the implementation of the `on_clicked` method, I assign the value from the text field in the dynamic document to the "classic" screen text field displayed beneath the dynamic document on the screen.

Figure 10 Display from the Example Program Containing a Form



In the PBO module `create_dd`, I create the dynamic document. First, I create the main document instance, `mydocument`, then I add a form to it. The `add_form` method returns a reference to a form object, and I can now use that object to fill the form itself. The following method calls place an input field and a pushbutton on the form. Normally, these would appear on separate lines, but I can force them both to be displayed on the same line by using the `line_with_layout` method. I call this method twice — once with the parameter `start = 'X'`, and once with the parameter `end = 'X'`. All elements that I place in the document between these two calls will be displayed on the same line.

Finally, I merge the document to ensure that the contents of my form are incorporated into the final

dynamic document, and display it in the container I created in the Screen Painter. Since I have specified the name of an empty container on the Screen, the system automatically creates a Custom Container Control for me as well as an HTML Viewer Control.

When you use this approach instead of your own HTML, you have much easier access to the contents of the dynamic document, since the HTTP stream is interpreted by the system and the name/value pairs returned by it are sorted neatly into the attributes of the relevant display elements.

Helpful Hints

✔ Dynamic documents center around an instance of the SAP HTML Viewer control, which itself invokes the API of the Web browser installed on the user's frontend. Therefore, when using dynamic documents, you should bear in mind that using them for very small parts of a screen might not be worthwhile from the point of view of performance.

✔ Another performance-related issue is that of continually re-creating the same dynamic document. Consider a screen for which you set the static `next screen` attribute to itself. In other words, if the user presses "Enter," the system processes the PAI event and then processes the screen again, starting with its PBO event. Consequently, it will waste time re-creating the same dynamic document that already exists. To prevent this from happening, you can use the statement:

```
CHECK <doc reference> IS INITIAL.
```

at the start of the module in which you create the document (where <`doc reference`> is the name of your reference variable for the main document instance). This will ensure that the module is only processed if you have not already created the document object.

Conclusion

This has been a brief introduction to dynamic documents and what you can do with them. As we have seen, they produce more attractive output in the new SAP visual design than do traditional lists, and they are better integrated into ABAP programming than any standalone HTML templates that you might otherwise use.

A graduate in modern languages from the University of Bristol (UK), Jonathan Maidstone joined SAP in Walldorf in 1996 and worked as a translator for the ABAP Workbench and ABAP language groups until late 1999. He is now a technical author, writing documentation for the SAP Control Framework, Desktop Office Integration, and SAP DCOM Connector teams, as well as assisting with their training and rollout activities. Jonathan can be reached at jonathan.maidstone@sap.com.

