Igor Barbarević

Design Patterns in Object-Oriented ABAP™
4 The Adapter Pattern .......................................................... 53
  4.1 Case-Study Problem: Application with a Command Tree .......... 53
  4.2 The Adapter in the Demo Application ................................. 55
  4.3 Interface ZIF_DPD_TREE_OBJECT ....................................... 55
    4.3.1 Types ........................................................................ 56
    4.3.2 Attributes ................................................................. 56
    4.3.3 Constants ................................................................... 57
    4.3.4 get_sub_objects Method .............................................. 58
    4.3.5 get_attribute Method ............................................... 58
    4.3.6 get_commands Method .............................................. 59
    4.3.7 user_command Method ............................................... 61
  4.4 Usage of the ZIF_DPD_TREE_OBJECT Interface .................... 61
    4.4.1 Active Objects Storage in the Application Model ............ 61
    4.4.2 Expanding Tree Nodes .............................................. 62
    4.4.3 Reacting to User’s Node Selection .............................. 62
    4.4.4 Creating Command Buttons ..................................... 63
    4.4.5 Responding to Command Buttons .............................. 63
  4.5 Summary ............................................................................ 64

5 The Factory Pattern ........................................................... 65
  5.1 Real-Life Example ............................................................. 67
  5.2 Case-Study Problem: Docking Containers Orderly Arranged ...... 67
  5.3 Class ZCL_DPD_DOCK_CONTAINER_FACTORY ....................... 69
    5.3.1 Attributes ................................................................ 70
    5.3.2 get_container Method .............................................. 71
    5.3.3 destroy_container Method ........................................ 71
    5.3.4 hide_all and unhide_all Methods .............................. 72
    5.3.5 refresh_total_size Method ........................................ 73
    5.3.6 resize_containers Method ........................................ 74
    5.3.7 Usage of the Container Factory Class ......................... 76
  5.4 Summary ............................................................................ 77

6 The Model-View-Controller (MVC) Pattern .......................... 79
  6.1 Two MVCs in the Demo Application .................................... 81
  6.2 Case-Study Problem I: Single Application in Different UI
    Technologies ........................................................................ 83
6.3 Case-Study Problem II: SAP GUI — Displaying Table Data in Different Controls ........................................... 84

6.4 The Application Model: Class ZCL_DPD_APP_MOD ............... 86
  6.4.1 Attributes ................................................................. 87
  6.4.2 constructor Method .................................................... 88
  6.4.3 get_app_mod Method ................................................... 89
  6.4.4 get_tree_object Method ............................................ 89
  6.4.5 get_tree_objects Method ........................................... 90
  6.4.6 add_tree_objects Method ........................................ 90

6.5 Class ZCL_DPD_MOD_DBT .................................................. 91
  6.5.1 Interfaces ................................................................. 91
  6.5.2 Attributes ............................................................... 91
  6.5.3 constructor Method ................................................... 93
  6.5.4 zif_dpd_tree_object~user_command Method ............. 94
  6.5.5 refresh Method ......................................................... 94
  6.5.6 save Method ............................................................ 95
  6.5.7 zif_dpd_tree_object~get_sub_objects Method ........ 95
  6.5.8 zif_dpd_tree_object~get_attribute Method ............. 96
  6.5.9 zif_dpd_tree_object~get_commands Method ............. 96

6.6 Class lcl_mod_dbt_carriers .............................................. 97
  6.6.1 Definition ............................................................... 97
  6.6.2 constructor Method ................................................... 98
  6.6.3 zif_dpd_tree_object~get_sub_objects method .......... 99

6.7 Other lcl_mod_* classes .................................................. 100

6.8 Controller 1: The Program ZDPD_APP_CON_GUI .................. 100

6.9 Class lcl_main_program .................................................. 102
  6.9.1 Definition ............................................................... 102
  6.9.2 Attributes ............................................................... 103
  6.9.3 run_main_program Method ........................................ 103
  6.9.4 constructor Method .................................................. 104
  6.9.5 on_double_click Method .......................................... 106
  6.9.6 on_expand_no_children Method ............................... 109
  6.9.7 on_menu_request Method ........................................ 110
  6.9.8 on_menu_select Method ......................................... 110

6.10 Class lcl_con_alv .......................................................... 111
  6.10.1 Definition ............................................................... 111
  6.10.2 alv Attribute ........................................................ 112
  6.10.3 container Attribute ............................................... 112
Contents

9.2.4 refresh_data Method .............................................................. 223
9.2.5 get_fields Method .............................................................. 223
9.3 Class ZCL_DPD_ITAB_DEC_AIRPORTS ...................................... 224
  9.3.1 Attribute ............................................................................ 225
  9.3.2 refresh_data Method .......................................................... 225
  9.3.3 get_fields Method .............................................................. 228
9.4 Class lcl_itab_dec_fl_sched_comment ..................................... 231
  9.4.1 Definition .......................................................................... 232
  9.4.2 Attributes .......................................................................... 232
  9.4.3 get_fields Method .............................................................. 232
9.5 Other Decorators in the Application ........................................... 234
9.6 Implementation of the Decorator Pattern in the Application .......... 235
9.7 Summary .................................................................................. 236

10 Summary ...................................................................................... 237

  10.1 Examples of Extension ........................................................... 237
    10.1.1 New Table-Maintaining Object at the Tree — Airplanes ..... 238
    10.1.2 New Dynpro Screen: Airplane Details ......................... 239
    10.1.3 Authorization Checks ..................................................... 243
    10.1.4 Toggle Display/Edit ......................................................... 244
    10.1.5 Locking Mechanism for Multiuser Applications ............. 245
  10.2 Benefits of Combining Objects with DPs ................................. 245
  10.3 Closing Word by the Author ................................................... 247

The Author ....................................................................................... 249

Index ................................................................................................. 251
The “gang of four” stated that the Decorator’s purpose was to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extended functionality.

You can see in Figure 9.1 that each Decorator can contain one component of the same superclass (a closed “has a” 1:1 loop). This is the most important feature of the Decorator — the ability to form a chain of components of the same superclass. Each one of them adds its behavior to the behavior of the entire chain. The chain always starts with a component that serves as an entry point for a client (ConcreteComponent in Figure 9.1). If the client calls a method of the first component (operation()), the method implements some functionality, or in other words, the program runs through the method’s code. The last (or the first) statement in the method must call the same method belonging to the preceding Decorator in the chain. It does what it has to do and again calls the method of its own preceding Decorator. This way, the method call propagates sequentially through the entire chain.

Figure 9.1 The Decorator

---

1 Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
Each component adds something of its own to the implementation and passes the control to the next component. The trick is that the client (usually another object in the program) doesn’t know that. The client only calls the method once, and doesn’t see the chain at all. The flexibility is excellent, because you can control the chain’s behavior just by plugging the components in and out, and the client doesn’t have to know anything about it. Moreover, you can make the first component a dummy that doesn’t have any implementation, except for passing the method call to the preceding component in the chain. It has no functionality but to serve as an entry point for the client. You can also create generic components as reusable repository components (Z* classes) and your modular system will get even more generic and flexible.

**Inconsistencies about Preceding and Succeeding**

Naming conventions can be somewhat confusing here. Is the dummy Decorator the first or the last component? Does a Decorator call a method of its preceding or succeeding Decorator component? It is important to make this decision once and then stick to it while naming components, variables, and parameters. Otherwise, it might greatly confuse matters. However, no approach is more intuitive than the other.

If we followed the method-call propagation through the chain, the dummy Decorator would have to be the first. The program execution starts from the client, which calls a method of the first component (the dummy), which in turn propagates the call to the next (or the successor) in the chain.

However, when instantiating the object, the dummy is always the last one to be instantiated, and it would be intuitive for the parameter passed to the constructor to be its predecessor.

Whichever paradigm is chosen, one factor will be intuitive and the other quite the opposite.

To stay consistent to the models in other literature, the predecessors were chosen (for example, the constructor parameter is called \_i_prec_dec). Therefore, the dummy is the last component rather than the first (although its method is the first one to be called).
9.1 Case-Study Problem: Lookup Data for Internal Tables

How can we apply this theory in practice? Let’s suppose that we have an internal table, and we want users to edit it (see Figure 9.2). We provide a user interface (for example, an Advanced List Viewer (ALV) grid) and create an editing mechanism. However, not all of the fields should be editable. If we have, for example, an airline code and two airport codes, this is enough for a developer, because it identifies those items uniquely and this is sufficient for further program flow.

<table>
<thead>
<tr>
<th>ID</th>
<th>Depart</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>JFK</td>
<td>BFO</td>
</tr>
<tr>
<td>AZ</td>
<td>FCO</td>
<td>FRA</td>
</tr>
</tbody>
</table>

Figure 9.2 Internal Table with Item Codes

However, no user would be happy with this. Users want to see more. They need some lookup data in the same line with the codes, such as airline name, airport name, and time zone. We would make those fields display-only (as in Figure 9.3).

<table>
<thead>
<tr>
<th>ID</th>
<th>airline</th>
<th>Depart</th>
<th>airport name</th>
<th>timeZone</th>
<th>Target</th>
<th>airport name</th>
<th>timeZone</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>American Airlines</td>
<td>JFK</td>
<td>New York JFK Kennedy, USA</td>
<td>UTC-5</td>
<td>SFO</td>
<td>San Francisco Int-Airport, USA</td>
<td>UTC-8</td>
</tr>
<tr>
<td>AZ</td>
<td>Airbaltic</td>
<td>FCO</td>
<td>Rome Leonardo Da Vinci, Italy</td>
<td>UTC+1</td>
<td>FRA</td>
<td>Frankfurt, Germany</td>
<td>UTC+1</td>
</tr>
</tbody>
</table>

Figure 9.3 Internal Table with Item Codes and Lookup Data

How do we do this? It’s easy — for airport, we’ll select data from the `SAIRPORT` table (let’s keep it simple for clarity and assume no performance issues). We will loop through a target internal table, find the corresponding airport, and pass those lookup fields:

```sql
SELECT id name FROM sairport
  INTO TABLE mt_sairport.
LOOP AT itab ASSIGNING <line>.
  READ TABLE mt_sairport ASSIGNING <sairport>
    WITH KEY id = <line>-airport_id.
  IF sy-subrc = 0.
    <line>-airport_name = <sairport>-name.
  ENDIF.
ENDLOOP.
```

We would do something similar for other lookups. We must make sure to put all of the `READ` statements in the same loop. It is a simple working solution, but what
will we do if we want to use it in another application and another, and so on? How many times have you copied and pasted your own programs or program parts? It’s boring and frustrating manual work. And what if it gets a lot more complicated than simple airport master data? You could spend days or even weeks developing and testing it, so it would be nice to make it reusable.

There’s an even better reason to make it reusable — maintenance. We want to resolve a problem only once and in one place. And if something goes wrong, or if an improvement opportunity occurs, we’d want to fix it at only one place, not at all of the places where we have implemented it by the copy-paste method.

For the airport example, we can resolve this problem with only one line of code:

```abap
CREATE OBJECT lo_dec_airport.
```

In fact, there’s more. But it’s generic and abstractable. First, we get data by

```abap
CALL METHOD lo_dec_airport->refresh_data( ).
```

Then we loop at a target table and pass fields by calling a method that dynamically detects which values need to be passed to which fields:

```abap
LOOP AT t_outtab ASSIGNING <line>.
  CALL METHOD lo_dec_airport->get_field
  CHANGING ch_struc = <line>.
ENDLOOP.
```

Although it’s a bit strange to resolve this by creating an object, it is nothing special. It’s easy to guess that the method `refresh_data` encapsulates the `SELECT` statement, and `get_fields` encapsulates the `READ` statement and passing field values.

There is nothing special about reusability either; it has been around for quite some time. For example, we could easily create a function module that would encapsulate airport master-data lookup. However, there is something better and far more flexible.
Reusability

Let’s suppose that within our application we have two internal tables with different needs for lookup.

- **Table A**
  - Carrier master
  - Airplane master

- **Table B**
  - Carrier master
  - Airport master 1
  - Airport master 2 (two different airport codes in one line)
  - Some lookup specific for the particular application

These tables are displayed in two ALV grids. In the context of the Decorator pattern, ALV grids will be the clients, and each lookup will be a Decorator. It will look like Figure 9.4.

![Figure 9.4 Two Chains of Decorators for Two Internal Tables](image)

Note here that the order of the components in the chain is not consistent with the naming convention that was chosen. Although the \texttt{dummy\_final} appears to be \textit{the first} component in the chain, we always refer to it as \textit{the final, or the last} one. As explained in Chapter 8, The Composite Pattern, the order of instantiation is the opposite of the direction of method-call propagation, and therefore it’s impossible to have fully consistent naming conventions.

In any event, the ALV grid only communicates with the last-in-chain dummy component with no functionality, and this is why this can be done generically. If we put this functionality into a repository component, we only have to find a way to plug in the rest of the components.

So, this part of the code is generic and belongs to a repository component:

```plaintext
FIELD-SYMBOLS: <table> TYPE TABLE,
                <line> TYPE ANY.
...
    o_dec_dummy_final->refresh_data( ).
LOOP AT <table> ASSIGNING <line>.
        CALL METHOD o_dec_dummy_final->get_field
            CHANGING ch_struc = <line>.
    ENDLOOP.
```

Note that everything is generic here: The `<table>` field symbol can refer to any internal table, the `<line>` accepts any type of line, and `o_dec_dummy_final` is an empty generic object with no specific implementation.

But so far we only have the client (ALV grid) and the dummy. This combination doesn’t do anything. Now we have to do the work specific to our application. We need to plug in some components to add some functionality. For internal table A, it will be:

```plaintext
CREATE OBJECT:
    o_dec_airplane,
    o_dec_carrier
    EXPORTING i_prev_dec = o_dec_airplane,
    o_dec_dummy_final
    EXPORTING i_prev_dec =
    o_dec_carrier.
```

Table B has different requirements, and therefore it must have a different chain:

```plaintext
CREATE OBJECT:
    o_dec_airport,
    o_dec_carrier
    EXPORTING i_prev_dec = o_dec_airport,
    o_dec_app_spec
    EXPORTING i_prev_dec =
```

```plaintext
...
Case-Study Problem: Lookup Data for Internal Tables

9.1

This is all we have to do locally. The entire implementation is encapsulated in these components, and only one CREATE OBJECT statement is sufficient to include a Decorator in a specific chain.

In the case of some commonly used features, such as airport master or carrier master lookup (let's pretend that they are part of real life), we can create those components as repository classes to use them in other applications, too. If it is some specific lookup that we will only use in that particular application, at one or more places, it probably pays to implement it as a Decorator local to the application.

On the other hand, if we have something that we're absolutely sure we will use once and only once, maybe you'd be better off using a classical solution. We can redefine the refresh method of the ZCL_DPD_MOD_DBT class in its subclass, call the superclass method, and then do the specific SELECT ... LOOP ... READ part. However, having a simple local Decorator for this purpose would be more consistent with the entire concept.

You have probably noticed that there are two different airports in table B (departure and arrival), and just one dec_airport. Technically, it is possible to create two Decorators of the same class, one for each airport. And it would probably be more intuitive and consistent to the model. However, multiplicity in one Decorator was enabled. We will see how and why in the following sections, where implementation of particular decorators will be described. We will also be able to see how the fields are assigned dynamically and generically.

The decorator classes can be both repository and local. Which option will be chosen depends on the need for reusability. If it is likely that the decorator will be used by a lot of programs or repository classes, then it is wise to create them as repository Z classes. If they only implement some specific one-time functionality, then it doesn’t make sense to pile repository objects and make a mess in the system.

In the demo, only one decorator was made local, while the others are repository Z classes.
9.2 Class ZCL_DPD_ITAB_DEC

ZCL_DPD_ITAB_DEC is a superclass for all other decorator classes, serving for instantiation of dummy_final at the same time. Figure 9.1 above shows that all of the components were inherited from an abstract class called Component. This is where we will deviate a little from the given model. This class will be made concrete rather than abstract, and when instantiated, it will become a o_dec_dummy_final. It is very simple, because it has no implementation. There are three methods, three attributes, and no events.

9.2.1 Attributes

Figure 9.5 shows the attributes of class ZCL_DPD_ITAB_DEC.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Level</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MO_PREC_DECORATOR</td>
<td>Inst</td>
<td>Proc</td>
<td>A reference to the preceding Decorator object in the chain. It is of the same type as the containing object (hence recursion in the diagram) and it is the object whose method will be called as the first statement of the same method within the current object.</td>
</tr>
<tr>
<td>MDT_REF</td>
<td>Inst</td>
<td>Proc</td>
<td>A data-reference object referencing a table that contains reference values for data selection to improve performance of the refresh_data method. For example, this is where the key field values for the FOR ALL ENTRIES clause can be found. MDT_REF is usually the same table that needs a lookup.</td>
</tr>
<tr>
<td>MTT_FNAME_MAPPINGS</td>
<td>Inst</td>
<td>Proc</td>
<td>A deep table (a table whose field is another table) with fieldname mappings between fields of an output table and fields of a DDIC table. It is typed to a class’ public type, but before we explain it in detail (see Section</td>
</tr>
</tbody>
</table>
9.2.2, Public Class’ Types), we should understand the motivation for having such an attribute.

"A table of tables with fieldname mappings" — sounds a bit weird and abstract, so let's explain. What does this mean exactly? Let's start our explanation by asking ourselves: How can we match the fields of an internal table to the fields of a database table? The easiest and most intuitive way would be to name them the same (for example, the database table field `SPF1I-CITYFROM` should get the corresponding field `CITYFROM` in our internal table). And it would probably work well in many situations. However, if we rely only on this principle, it would impose limitations. We might get into situations where we are supposed to have more fields with the same name in one table, which is technically impossible. For example, in the airports master data table `SAIRPORT`, the name of the airport is stored in the field `NAME`. If we wanted to have both departure and arrival airports we couldn't use the same name for both.

Also, the fields might get unwanted values if they get populated by the wrong Decorator, which might “think” that the particular field belongs to it. For example, it is common for master data tables to use generic field name `NAME1` to describe their data (vendor master: `LFA1-NAME1`, customer master: `KNA1-NAME1`). If we had `NAME1` in our internal table, how would we know who it belongs to? Therefore, it would be good to have different field names in internal tables with some sort of field mapping to the database table fields.

To resolve this situation, explicit field matching was enabled, with two options:

« Hardcoded default fieldname mappings
« Table of fieldname mappings as a constructor parameter

An optional parameter was assigned to the constructor method — a mapping table of fieldname pairs (an internal table fieldname and a dictionary table fieldname). It is optional, so the default mapping table can be hardcoded and used within the component, unless a different table was provided as a parameter.

It sounds more or less clear, but it gets complicated when the same lookup is required more than once at the same time, such as with the departure and arrival airports. There are two airport codes in one internal table (the one that corresponds to the `SPF1I` database table): `AIRPFROM` and `AIRPTO`, and they should have the corresponding lookup fields: `NAME_FROM` and `NAME_TO`. The same applies to time zones. Therefore, a decorator is needed to get lookup data for two groups of
lookup fields. The fields of a display internal table (with line structure ZDPD_SPFL1_DS) should match the fields of the lookup database table SAIRPORT, such as in Figure 9.6.

The most intuitive solution would be to create many (in this case two) Decorator objects of the same type. In this example, it would be one for each airport. And it would work well, except that performance would worsen over time, because the same SELECT statements (or however the data is fetched) would be invoked multiple times unnecessarily. It would be better to do it once for all of them. The solution emerged naturally: a table of tables of field pairs. Each airport from the example would have its own mapping table, as shown in Figure 9.7.
Now that we know the motivation, let us see how can we type the attribute `MTT_FNAME_MAPPINGS`. It will be typed to a public class' type, which we will build in three steps in the next section.

### 9.2.2 Public Class' Types

To declare such a table, we first need a structure with a pair of field names:

```plaintext
TYPES: BEGIN OF s_fname_pair,
    fname_itab TYPE fieldname,
    fname_ddic TYPE fieldname,
END OF s_fname_pair.
```

Then, we need a table type whose line type is exactly the same as the previous structure:

```plaintext
ty_t_fname_mapping
    TYPE STANDARD TABLE OF s_fname_pair
    WITH DEFAULT KEY.
```

And then we need another table type whose line is the previous table type:

```plaintext
ty_tt_fname_mappings
    TYPE STANDARD TABLE OF
        ty_t_fname_mapping
    WITH DEFAULT KEY.
```
These types must be accessible by both the ZCL_DPD_ITAB_DEC class and other components that will use the decorator, such as other global classes, programs, etc. To achieve global visibility, the types could have been created as global dictionary objects (Z data elements and structures). However, they naturally belong to the class ZCL_DPD_ITAB_DEC, so it makes sense to create them as its public types (as in Figure 9.8), while still enabling the outside world to access them.

![Figure 9.8 Public Types of ZCL_DPD_ITAB_DEC Class](image)

And, finally, we can declare an attribute based on these types — a table of field-name mappings MTT_FNAME_MAPPINGS, as in Section 9.2.1, Attributes, and Figure 9.5 above.

### 9.2.3 constructor Method

Figure 9.9 shows the parameters of method constructor.

![Figure 9.9 Parameters of Method constructor](image)

constructor doesn’t do much here. It only accepts parameter values and passes them to the corresponding attributes (see Listing 9.1). There are three parameters that correspond to the class attributes: \_prec_dec, \_ref, and \_tt fname mappings.

METHOD constructor.

```
me->mo_prec_decorator = \_prec_dec.
GET REFERENCE OF \_ref INTO me->\_mdt_ref.
me->\_tt fname mappings = \_tt fname mappings.
ENDMETHOD.
```

Listing 9.1 Method constructor
### 9.2.4 refresh_data Method

The parameterless method `refresh_data` fetches data from the database all at once, and the data is read later by the `GET_FIELDS` method. In most cases it consists of one or more `SELECT` statements, but it can be anything that actually gets data, such as a function module or some more complex solution.

We could have the same functionality with no `refresh_data` at all by using `SELECT` in the `get_fields` method. However, we would leave the user without the possibility of refreshing data at any time.

The true implementation of the method was left to the inheriting subclass. In this class, it doesn’t do anything but call the same method of the preceding Decorator in the chain (see Listing 9.2).

```plaintext
METHOD refresh_data.
  IF NOT me->mo_prec_decorator IS INITIAL.
    me->mo_prec_decorator->refresh_data( ).
  ENDIF.
ENDMETHOD.
```

**Listing 9.2 Method refresh_data**

### 9.2.5 get_fields Method

Figure 9.10 shows the parameter of method `get_fields`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>P</th>
<th>O</th>
<th>Tp</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH_STRUC</td>
<td>Change</td>
<td>[ ]</td>
<td>[ ]</td>
<td>Type ANY</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 9.10 Parameter of Method get_fields**

This method is called when the client wants lookup data. It takes the changing parameter `CH_STRUC` and finds the needed key fields in it. With the key fields’ values, it then finds the corresponding lookup data and fills it into the lookup fields of the `CH_STRUC` parameter.

The `CH_STRUC` parameter is a structure of a generic type `ANY`, because many different clients can use the same Decorator and the output structure is not known until runtime. Therefore, a special set of statements must be used to enable the handling of generic data.
However, this is to be implemented in the inheriting subclasses. Similar to the `refresh_data` method, in this class we only pass control to the preceding Decorator in the chain (see Listing 9.3).

```java
METHOD get_fields.
  IF NOT me->mo_prec_decorator IS INITIAL.
    me->mo_prec_decorator->get_fields(
      CHANGING ch_struc = ch_struc).
  ENDIF.
ENDMETHOD.

Listing 9.3 Method get_fields
```

In the following subsections, which describe a few of the Decorators, we’ll see that concrete implementations of `refresh_data` and `get_fields` always begin with the call of the superclass methods:

```java
super->refresh_data( ).
```

and

```java
super->get_fields( CHANGING ch_struc = ch_struc ).
```

Because those Decorators are inherited from the superdecorator, this actually means we are calling the previously mentioned methods, which pass control to the preceding Decorator.

### 9.3 Class ZCL_DPD_ITAB_DEC_AIRPORTS

The Airport Decorator provides lookup data from the airport master data table `SAIRPORT` for a given airport code (fieldname `ID`). Here, it would be good to take care of performance when fetching data. Let us pretend that the `SAIRPORT` table has 200 fields and 100,000 records. In such conditions, a `SELECT *` without a `WHERE` clause would be a real performance killer. As a way to reduce the load, the `refresh_data` method will be a bit more complicated.

The class is inherited from the superdecorator `ZCL_DPD_ITAB_DEC`, the public methods are redefined, and selection data is stored in a private `mt_sairport` table attribute.
9.3.1 Attribute

The table MT_SAIRPORT is the only attribute in the class, and this is where the refresh_data method stores the data that it selects from the SAIRPORT table, to be used later by the get_fields method.

9.3.2 refresh_data Method

It was said in the introduction to the Airport Decorator that we will pay special attention to performance because we pretended that the SAIRPORT table was too big in both dimensions to afford a simple SELECT * without a WHERE. The logic is simple: we usually won’t need all of the (imagined) 200 fields of SAIRPORT. Instead, we will provide a field list to be selected, based on the fieldname mappings. To do so, we could create an internal table dynamically to select data into. However, a statically declared internal table will be used for the demo, containing all of the fields of SAIRPORT, filled by INTO CORRESPONDING FIELDS OF addition.

Then we will create a WHERE clause based on key field values (ID) provided in the IT_REF parameter of the constructor, and invoke the SELECT statement.

Let’s get to the code next. First, we must not forget to pass the control to the preceding Decorator in the chain.

    super->refresh_data( ).

Remember the superdecorator’s constructor, it takes the generic table IT_REF parameter (whatever type) and packs it into a D_REF (TYPE REF TO data) attribute. We first need to unpack it into a generic field-symbol <t_ref> TYPE STANDARD TABLE. If this fails, then the rest of the method makes no sense either.

    ASSIGN me->mdt_ref->* TO <t_ref>.
    IF sy-subrc NE 0. RETURN. ENDF.

Now we need the fieldname mappings to fill two tables of fieldnames:

- /L50776 lt_keyfield_fnames will be needed to build the WHERE clause for the SELECT statement. It should contain the fieldnames of an internal table that correspond to the key field ID of the SAIRPORT dictionary table.
- /L50776 lt_select_fnames will be a dynamic field list of a SELECT statement. It should contain all of the fields that need to be selected from the SAIRPORT table.
If the `lt_keyfield_fnames` stays empty, it means that the target table doesn't have any ID fields, and that there's no job for the Airport Decorator. It doesn't necessarily mean that it was plugged into a chain by mistake. Maybe it serves a table that is created dynamically, or there's some other reason to plug it in. Besides, we must take care not to break the chain. So instead of raising an exception, we'd better do nothing and leave the method quietly (see Listing 9.4).

```
LOOP AT me->mtt_fname_mappings
  ASSIGNING <t_fname_mapping>.
LOOP AT <t_fname_mapping> ASSIGNING <fname_mapping>.
  IF <fname_mapping>-fname_ddic = 'ID'.
    APPEND <fname_mapping>-fname_itab TO lt_keyfield_fnames.
  ENDIF.
  APPEND <fname_mapping>-fname_ddic TO lt_select_fnames.
ENDLOOP.
ENDLOOP.
```

Listing 9.4 Preparing Key Fields and the Fields to be Selected

The field names are here. Let's get rid of the duplicates.

```
SORT lt_keyfield_fnames.
DELETE ADJACENT DUPLICATES FROM lt_keyfield_fnames.
SORT lt_select_fnames.
DELETE ADJACENT DUPLICATES FROM lt_select_fnames.
```

Now we need to fill the `lt_airport_ids` table that will be used by the `FOR ALL ENTRIES IN ... WHERE` clause of the `SELECT` statement. We will loop at the reference table, and for each line we will dig out all of the airport codes (there can be more than one). We have filled the `lt_keyfield_fnames` table just for this purpose.

```
CHECK <t_ref> IS ASSIGNED.
LOOP AT <t_ref> ASSIGNING <line>.
  LOOP AT lt_keyfield_fnames ASSIGNING <fname>.
    ASSIGN COMPONENT <fname> OF STRUCTURE <line> TO <fld>.
    CHECK sy-subrc = 0.
    IF NOT <fld> IS INITIAL.
      APPEND <fld> TO lt_airport_ids.
```

226
Now we only need to select the data.

IF lt_airport_ids IS INITIAL. EXIT. ENDIF.
SORT lt_airport_ids.
DELETE ADJACENT DUPLICATES FROM lt_airport_ids.
SELECT (lt_select_fnames) FROM sairport
  INTO CORRESPONDING FIELDS OF TABLE me->mt_sairport
  FOR ALL ENTRIES IN lt_airport_ids
  WHERE id = lt_airport_ids-table_line.
SORT me->mt_sairport BY id.

Listing 9.5 provides the entire code for a better overview.

METHOD refresh_data.
  TYPES: ty_fname_tab TYPE STANDARD TABLE OF fieldname.
  DATA:
    lt_airport_ids TYPE STANDARD TABLE OF sairport-id,
    lt_keyfield_fnames TYPE ty_fname_tab,
    lt_select_fnames TYPE ty_fname_tab.
  FIELD-SYMBOLS:
    <line> TYPE ANY,
    <fld> TYPE ANY,
    <fname> TYPE ANY,
    <t_ref> TYPE STANDARD TABLE.
    <t_fname_mapping> TYPE ty_t_fname_mapping,
    <fname_mapping> TYPE LINE OF ty_t_fname_mapping.
  *SUPER method call (REFRESH_DATA of prec dec in chain)
    super->refresh_data( ).
  *assign reference table or struc (passed to CONSTRUCTOR)
    ASSIGN me->mdt_ref->* TO <t_ref>.
    IF sy-subrc NE 0. RETURN. ENDF.
  *define fname(s) which represent airport and flds to select
    LOOP AT me->mtt_fname_mappings
      ASSIGNING <t_fname_mapping>.
      LOOP AT <t_fname_mapping> ASSIGNING <fname_mapping>.
      IF <fname_mapping>-fname_ddic = 'ID'.
        APPEND <fname_mapping>-fname_itab TO lt_keyfield_fnames.
      ENDF.
APPEND \texttt{<fname_mapping>\textunderscore ddic} TO \texttt{lt\_select\_fnames}.

ENDLOOP.

IF \texttt{lt\_keyfield\_fnames} IS INITIAL. RETURN. ENDIF.

SORT \texttt{lt\_keyfield\_fnames}.

DELETE ADJACENT DUPLICATES FROM \texttt{lt\_keyfield\_fnames}.

SORT \texttt{lt\_select\_fnames}.

DELETE ADJACENT DUPLICATES FROM \texttt{lt\_select\_fnames}.

*define \texttt{lt\_airport\_ids} for 'FOR ALL ENTRIES FROM sairport'

CHECK \texttt{<t\_ref>} IS ASSIGNED.

LOOP AT \texttt{<t\_ref>} ASSIGNING \texttt{<line>}

LOOP AT \texttt{lt\_keyfield\_fnames} ASSIGNING \texttt{<fname>}

ASSIGN COMPONENT \texttt{<fname>} OF STRUCTURE \texttt{<line>} TO \texttt{<fld>}

CHECK sy\textunderscore subrc = 0.

IF NOT \texttt{<fld>} IS INITIAL.

APPEND \texttt{<fld>} TO \texttt{lt\_airport\_ids}.

ENDIF.

ENDLOOP.

ENDLOOP.

*SELECT data from LFA1

IF \texttt{lt\_airport\_ids} IS INITIAL. EXIT. ENDIF.

SORT \texttt{lt\_airport\_ids}.

DELETE ADJACENT DUPLICATES FROM \texttt{lt\_airport\_ids}.

SELECT \texttt{(lt\_select\_fnames) FROM sairport}

INTO CORRESPONDING FIELDS OF TABLE \texttt{me\textunderscore mt\_sairport}

FOR ALL ENTRIES IN \texttt{lt\_airport\_ids}

WHERE \texttt{id = lt\_airport\_ids\textunderscore table\_line}.

SORT \texttt{me\textunderscore mt\_sairport} BY \texttt{id}.

ENDMETHOD.

Listing 9.5 Method refresh_data

9.3.3 get\textunderscore fields Method

get\textunderscore fields method is called by the client within its LOOP at the main output table that needs its lookup fields to be filled. The table line is then passed as a \texttt{CH\textunderscore STRUC} parameter to the method get\textunderscore fields of a dummy decorator, which propagates the call through all of the decorators in chain. Let us see how it works, specifically in the Airports Decorator.

After calling the supermethod, the loop on the field mappings starts.
* SUPER method call (calls GET_FIELDS of next dec in chain)
  CALL METHOD super->get_fields
  CHANGING ch_struc = ch_struc.
* loop at field mappings
LOOP AT me->mtt_fname_mappings
  ASSIGNING <t_fname_mapping>.
Now that we have the first mapping table, we have to see whether it makes sense
to proceed. That is, is there a key field in it? If not, we let the loop pass; that is, we
go to the next mapping table.

  READ TABLE <t_fname_mapping> ASSIGNING <fname_mapping>
  WITH KEY fname_ddic = 'ID'.
  IF sy-subrc = 0.
    ASSIGN COMPONENT <fname_mapping>-fname_itab
    OF STRUCTURE ch_struc TO <field>.
  ELSE.
    CONTINUE.
  ENDIF.

  However, if the field symbol <field> is assigned, it means that it contains the
  value of the key field ID and that we can go on and read the data table mt_sairport.

  UNASSIGN <lookup_line>.
  READ TABLE me->mt_sairport ASSIGNING <lookup_line>
  WITH KEY id = <field>
  BINARY SEARCH.

  Whether we found something here or not, we will now loop at the field pairs of
  the mapping and do something with the lookup fields of the structure ch_struc,
  which is passed as a changing parameter. As a reminder, this structure is a line of
  the target internal table that we need to lookup data for.

  If some lookup data was found, and we check that with IF <line> IS ASSIGNED,
  we will pass each value from a line <line> of an internal table mt_sairport to the
  corresponding fields of the structure ch_struc. If no lookup data was found, it
  can only mean that the key was invalid (perhaps as the result of user error), and
  we'll simply clear those fields.
LOOP AT <t fname_mapping> ASSIGNING <fname_mapping>.
 ASSIGN COMPONENT <fname_mapping>-fname_itab
 OF STRUCTURE ch_struc TO <targ fld>.
 CHECK sy-subrc = 0.
 IF <lookup_line> IS ASSIGNED.
  ASSIGN COMPONENT <fname_mapping>-fname_ddic
  OF STRUCTURE <lookup_line> TO <field>.
  CHECK sy-subrc = 0.
  <targ fld> = <field>.
 ELSE.
  CLEAR <targ fld>.
 ENDIF.
 ENDLOOP.

Now we have everything settled for the first mapping (or the first airport). With
another loop pass at the TT_FNAME_MAPPINGS table, all of the same steps are done
for the next mapping (second airport), and so on.

Listing 9.6 provides the entire code for a better overview.

METHOD get_fields.
 FIELD-SYMBOLS:
     <targ fld> TYPE ANY,
     <field> TYPE ANY,
     <lookup_line> TYPE ANY,
     <t fname_mapping> LIKE LINE OF me->mtt_fname_mappings,
     <fname_mapping> LIKE LINE OF <t fname_mapping>.
* SUPER method call (calls GET_FIELDS of next dec in chain)
 CALL METHOD super->get_fields
  CHANGING ch_struc = ch_struc.
* loop at field mappings
 LOOP AT me->mtt_fname_mappings
    ASSIGNING <t fname_mapping>.
* find key field airport (if none, exit loop pass)
 READ TABLE <t fname_mapping> ASSIGNING <fname_mapping>
  WITH KEY fname_ddic = 'ID'.
 IF sy-subrc = 0.
  ASSIGN COMPONENT <fname_mapping>-fname_itab
  OF STRUCTURE ch_struc TO <field>.
 ELSE.
  CONTINUE.

The class lcl_itab_dec_fl_sched_comment maintains flight schedules, with the ability to enter a comment for each record. The comments are not edited in the ALV grid because the cells can accept a limited number of characters and do not support paragraph breaks. It would be nice, however, to have a part of the text (say, the first 100 characters) in a read-only field next to the airport code and name. This is a lookup field, isn’t it? So why not stay consistent and implement it as the Decorator pattern?

This component is excellent for showing a diversity of implementations of Decorators. Up to now we had Decorators as components that SELECT data from dictionary tables and then fill lookup fields of the target table given at runtime. However, this is just the most usual situation, not standard procedure, and not a part of the pattern.
The data selection can be implemented in literally any way, not only by selecting from tables. For example, this Decorator doesn’t have a single SELECT statement. It doesn’t even have the refresh_data or constructor method redefined. How can that be? As explained earlier, the refresh_data method fetches data from the database all at once to improve performance. However, in this case it is not possible because the data is fetched in a very specific way, by using the function module READ_TEXT, which only fetches one text at a time. We can do it in the get_fields method, and therefore we don’t need refresh_data at all. It doesn’t mean that it doesn’t exist. It is just not redefined, which means that it runs in the superclass ZCL_DPD_ITAB_DEC, where it only passes control to the preceding Decorator in the chain. Similarly, we don’t have a constructor and we don’t use field name pairs. On the contrary, this Decorator has a very simple implementation because it is one-time and very specific. The field names are hardcoded and there are no complex constructions because we didn’t have to create anything generic.

9.4.1 Definition
Listing 9.7 shows the definition of class lcl_itab_dec_fl_sched_comment.

CLASS lcl_itab_dec_fl_sched_comment
  DEFINITION INHERITING FROM zcl_dpd_itab_dec.
  PUBLIC SECTION.
  METHODS get_fields REDEFINITION.
ENDCLASS.

Listing 9.7 Definition of Class lcl_itab_dec_fl_sched_comment

9.4.2 Attributes
There is no refresh_data method to fetch data so there’s no need for attributes either, because attributes are usually used as storage for the data to be used later by the get_fields method. This method now does all of the work alone.

9.4.3 get_fields Method
The get_fields method logic is completely different than in all other decorators. The field names are hardcoded because this one is specific and for one-time use. We simply collect the fields of the changing structure that form the name of the
standard text, get the text, and put it into the corresponding field of the structure (see Listing 9.8).

METHOD get_fields.
  DATA: l_tname TYPE tdobname,
       lt_lines TYPE STANDARD TABLE OF tline.
  FIELD-SYMBOLS:
    <carrid> TYPE sflight-carrid,
    <connid> TYPE sflight-connid,
    <comment> TYPE zdpd_spfli_ds-comment,
    <line> LIKE LINE OF lt_lines.
* SUPER method call (calls GET_FIELDS of next dec in chain)
  CALL METHOD super->get_fields 
    CHANGING 
      ch_struc = ch_struc.
  ASSIGN COMPONENT:
    'CARRID' OF STRUCTURE ch_struc TO <carrid>.
    'CONNID' OF STRUCTURE ch_struc TO <connid>.
    'COMMENT' OF STRUCTURE ch_struc TO <comment>.
  CHECK <carrid> IS ASSIGNED AND <connid> IS ASSIGNED AND <comment> IS ASSIGNED.
* get the lookup data
  CONCATENATE 'ZDPD_FL_' <carrid> <connid> INTO l_tname.
  CALL FUNCTION 'READ_TEXT' 
    EXPORTING
      id = 'ST'
      language = sy-langu
      name = l_tname
      object = 'TEXT'
    TABLES
      lines = lt_lines
    EXCEPTIONS
      OTHERS = 8.
  CHECK sy-subrc = 0 AND lt_lines IS NOT INITIAL.
* pass value for lookup fld
  READ TABLE lt_lines ASSIGNING <line> INDEX 1.
  <comment> = <line>-tdline.
ENDMETHOD.

Listing 9.8  Method get_fields
9.5 Other Decorators in the Application

The demo application contains more Decorators:

- Carriers Decorator (class ZCL_DPD_ITAB_DEC_CARRIERS)
- Planes Decorator (class ZCL_DPD_ITAB_DEC_PLANES)

Because the logic is the same for all of the Decorator components, there is no need to describe them in detail. Their lookup functionality was based on selecting data from dictionary tables, and they are more similar to the Airport Decorator than to the Standard Text Comment Decorator. The implementation is simpler and there’s no need to explain them in detail.

However, it could be interesting to pay a little more attention to the get_fields methods of all Decorators except the Standard Text Comment Decorator.

They are so similar that a question arises naturally: Would it be possible to create a fully generic get_fields method and implement it in superdecorator to spare development and maintenance effort?

The only difference between the methods is in the way the data is found. Particularly, the following READ TABLE statement is specific for each Decorator, and it would be good to make it generic:

```plaintext
READ TABLE me->mt_saairport ASSIGNING <line>
    WITH KEY id = <field>
    BINARY SEARCH.
```

The first problem is that the READ TABLE statement does not have a full dynamic version. You can try with the most dynamic version:

```plaintext
READ TABLE <generic_tab>
    WITH KEY (v_fname) = 'Value'...
```

While you can provide each key fieldname dynamically, you can only have a fixed number of key fields. You cannot use something like the fully dynamic WHERE (t_where) clause and this is a significant constraint.

But also, why would all of the Decorators be limited to only one dictionary table? You can have complex solutions with more tables or other ways of getting data
Implementation of the Decorator Pattern in the Application

So far, you have only seen how particular Decorators are created. But how and where does the whole thing work? The primary goal was to provide lookup data for an internal table that already has main data that includes certain key fields, such as the airport code. An internal table that needs to be filled and displayed to a user is contained in the "supermodel" (no, it’s not a beautiful woman) class ZCL_DPD_MOD_DBT. This is a generic superclass inherited by concrete classes that actually hold the data.

The whole concept of models is explained in Chapter 6, The Model-View-Controller (MVC) Pattern. From the Decorator developer’s perspective, it is important to see that this class contains the data-reference object MDT_OUTTAB, which represents the output internal table (created dynamically by the inheriting objects), and the MO_DEC_FINAL dummy Decorator. The refresh method of class ZCL_DPD_MOD_DBT (called by a controller when it needs fresh data) contains the generic implementation of the lookup functionality:

```plaintext
me->mo_dec_final->refresh_data( ).
LOOP AT <outtab> ASSIGNING <table_line>.
   me->mo_dec_final->get_fields(
      CHANGING ch_struc = <table_line> ).
ENDLOOP.
```

However, instantiating the Decorator chain is specific and can’t be implemented in the generic superclass. It is up to each object to decide for itself how the chain should look.

The lcl_mod_dbt_flight_schedules class (inherited from lcl_mod_dbt) contains a large chain instantiation. Note that the chain ends with the dummy mo_dec_final, which is an object attribute (denoted by prefix me->), rather than a method’s local variable.
The Decorator Pattern

CREATE OBJECT:
lo_dec_airports
  EXPORTING it_ref = <outtab>
  itt_fname_mappings = tt_fname_mappings.
lo_dec_carriers
  EXPORTING i_prec_dec = lo_dec_airports.
lo_dec_comment
  EXPORTING i_prec_dec = lo_dec_carriers.
me->mo_dec_final
  EXPORTING i_prec_dec = lo_dec_comment.

Demonstrating reusability, the lcl_mod_dbt_flights class uses the Carriers and Planes Decorators as well.

CREATE OBJECT:
lo_dec_planes,
lo_dec_carriers
  EXPORTING i_prec_dec = lo_dec_planes.
me->mo_dec_final
  EXPORTING i_prec_dec = lo_dec_carriers.

9.7 Summary

This is, in the author’s opinion, one of the most exciting patterns. What do we gain from it?

In the demo, it was used to form a chain of different reusable components that perform lookup data functionalities for internal tables. The components can be plugged in the chain and reordered by just one statement of code!

We can have an infinite and expandable set of objects that we can freely combine in any way we want, depending on the particular need. Once we create the new member, no matter how complex it may be, we can plug it in and out by just one statement of code! That is, if we code at all.

Say you have a voice synthesizer created by some sort of visual application composer. It has a chain of two boxes. One box says “Hello,” and another says “how are you doing?” You create the new box that says “Mr. Barbarič,” just drag and drop it in between the existing two, and you’re done. Now the machine says “Hello, Mr. Barbarič, how are you doing?”
Index

A

ABAP procedural 237
Adapter 53, 64
add_tree_objects 90
Aggregation 20
Alexander, Christopher 12
alv 112
ALV grid 111, 112, 116, 121, 122, 124, 166, 213, 215, 216
ALV_GRID_VIEW 154
Application controllers 83
Application model 38, 83
Authorization check 243

B

BASIC 237
Bowline knot 13
Business Application Programming Interface (BAPI) 32
Business Objects Repository (BOR) 32
Business Server Pages (BSP) 83

C

C# 32
CFW 19, 68
CL_DD_DOCUMENT 82, 129
CL_GUI_ALV_GRID 67, 82, 83, 129
CL_GUI_DOCKING_CONTAINER 67, 83
class_pai 139
class_pbo 140
Command button 63
Command tree 33, 53, 69
authorization control 243
COMPONENTCONTROLLER, ZDPD_APP_CON_WEBDYNPRO 143
Composite 191, 209
constructor 49
lcl_con_alv 113
lcl_con_dd 122
lcl_con_dynpro_screen 138
constructor (cont.)
lcl_main_program 104
lcl_mod_dbt_carriers 98
lcl_mod_dbt_flights 199
lcl_mod_dbt_planes 239
ZCL_DPD_APP_MOD 88
ZCL_DPD_ITAB_DEC 222
ZCL_DPD_MOD_DBT 93
ZCL_DPD_ST_TEXT_EDITOR 180
Controller method 49
controller 112
Context 143
Controller 79, 81, 82
Controls Framework (CFW) 19, 68
create_controller 138

D

Decorator 41, 202, 211, 236, 245
Class ZCL_DPD_ITAB_DEC_CARRIERS 234
Implementation 235
Naming conventions 212
ZCL_DPD_ITAB_DEC_PLANES 234
Demo application 29
Design patterns
Benefits 245
Definition 12
in software design 14
destroy_container 71
Docking containers 67, 68
Dynamic Document (DD) 34, 41
Dynpro screens 129

E

Empty containers 36, 69
Expanding tree nodes 62, 109, 149

F

Façade 135, 171, 188
Web Dynpro 188
Factory 65, 135
Index

flight_schedule_st_text_edit 119, 164
Flow logic 130, 241
free
lcl_con_alv 118
lcl_con_dd 128
ZCL_DPD_ST_TEXT_EDITOR 188

G

Gang of four 14, 47, 53, 79, 171, 191, 211
get_app_mod 89
get_attribute 58
get_commands 59
get_container 71
get_fcat 118
get_fields
  lcl_itab_dec_fl_sched_comment 232
  ZCL_DPD_ITAB_DEC 223
  ZCL_DPD_ITAB_DEC_AIRPORTS 228
get_sub_objects 58, 240
get_tree_object 89
get_tree_objects 90
Global program variables 129
GUI status 131

H

handlein_alv 155
hide_all 72

I

Inheritance 20
Instantiate only once 47

J

Java 32, 49

L

lcl_con_alv 76, 111
lcl_con_dd 121
lcl_con_dynpro_screen 65, 135
lcl_itab_dec 224
lcl_itab_dec_fl_sched_comment 231
lcl_main_program 49, 102, 242
lcl_mod_dbt_carriers 97
lcl_mod_dbt_flights 196, 236
lcl_mod_dbt_planes 239
Local classes 38
Locking mechanism 245
Lookup data, internal table 213

M

M_CAPTION 93, 180
M_DISP_STRUC_NAME 92
m_dynnr_main 137
m_gui_status_main 137
M_TABNAME 92
m_title 137
MAIN_VIEW 149
MAIN_WINDOW 148
mdt_outtab 92, 137
message 147
MO_APP_MOD 88
MO_CONTAINER 180
mo_dd 122
MO_DEC_FINAL 92
mo_model 137
MO_PRC_DECORATOR 218
MO_TEXTEDIT 180
Model 79, 81
Model state 79
ms_node 56
MS_THEAD 180
MT_ACTIVE_OBJECTS 87
MT_MAIN_WHERE 93
mt_scr_contr 137
MTT_FNAME_MAPPINGS 218
Multiuser applications 245
MVC 41, 79, 165, 235

N

Naming conventions 17
Node selection 62, 106, 153
Index

**O**

OBJECT 144
Object orientation 32
on_clicked 123
on_double_click 106, 242
on_expand_no_children 109
on_menu_request 110
on_menu_select 110
on_toolbar 116
on_toolbar_func_sel 186
on_user_command 117
on_action_button_click 160
on_actiontree_load_children 149
on_actiontree_node_click 153
Open Object — Close Object 166

**P**

PAI 140
PBO 48, 141
PDF form 174
Polymorphism 65
Process After Input (PAI) 140
Process Before Output (PBO) 48
Proxy 171

**R**

Recursion 192
refresh
   ALV_GRID_VIEW 161
   ZCL_DPD_MOD_DBT 94
refresh_data
   ZCL_DPD_ITAB_DEC 223
   ZCL_DPD_ITAB_DEC_AIRPORTS 225
refresh_display 124
refresh_total_size 73
resize_containers 74
Return of Investment (ROI) 11
run_main_program 103
Running program only once 48

**S**

S_THEAD 145
SAP GUI application controller 41
SAP R/3 237
save
   ALV_GRID_VIEW 162
   ZCL_DPD_MOD_DBT 95
   ZCL_DPD_ST_TEXT_EDITOR 186
Sequence diagram 20, 167, 197
show ZCL_DPD_ST_TEXT_EDITOR 181
Single-record maintenance 36, 129
Singleton 51, 66
   in application model 50
SmartForm 174
ST_TEXT_EDITOR 145
Static (class) UML diagram 20
Static diagram 19, 40, 42, 87, 101

**T**

Table maintenance 34, 91
Text editor 177
Toggle display 244
toolbar_add_buttons 163
Total Cost of Ownership (TCO) 11
Transaction
   SE38 32
   SE80 32, 44
   SO01 174
 TREE_NODE 144

**U**

UML 19, 246
unhide_all 72
Unified Modeling Language (UML) 19, 246
Usage 20
user_command 61

**V**

Version compatibility 31
View 79, 81, 82
Index

W

wdoinit COMPONENTCONTROLLER 145
Web Dynpro 18, 42, 81, 142, 176
Web Dynpro application controller 42
WRITE statement 83

Z

ZCL_DPD_APP_MOD 50, 81, 82, 83, 85, 86, 238
ZCL_DPD_ITAB_DEC 92, 218, 222, 224, 232
ZCL_DPD_ITAB_DEC_AIRPORTS 224
ZCL_DPD_ITAB_DEC_CARRIERS 202, 234
ZCL_DPD_ITAB_DEC_PLANES 202, 234
ZCL_DPD_MOD_DBT 82, 85, 91, 217, 235
ZCL_DPD_ST_TEXT_EDITOR 178, 179
ZCL_DPD_ST_TEXT_EDITOT 119
ZDPD_APP_CON_GUI 81, 82, 83, 85, 100, 241
ZDPD_APP_CON_WEBDYNPOR 44, 142, 143, 188
ZDPD_ST_TEXT_EDITOR 119, 188
ZIF_DPD_TREE_OBJECT 55, 61, 91, 113, 144
zif_dpd_tree_object-get_attribute 96, 113, 123, 151, 155
ZIF_DPD_TREE_object~GET_COMMANDS 116, 125, 163, 178
zif_dpd_tree_object-get_commands 96
zif_dpd_tree_object-get_sub_objects 95, 99, 109, 151, 205
zif_dpd_tree_object-user_command 94, 106, 111, 117, 124, 141, 156, 161, 169