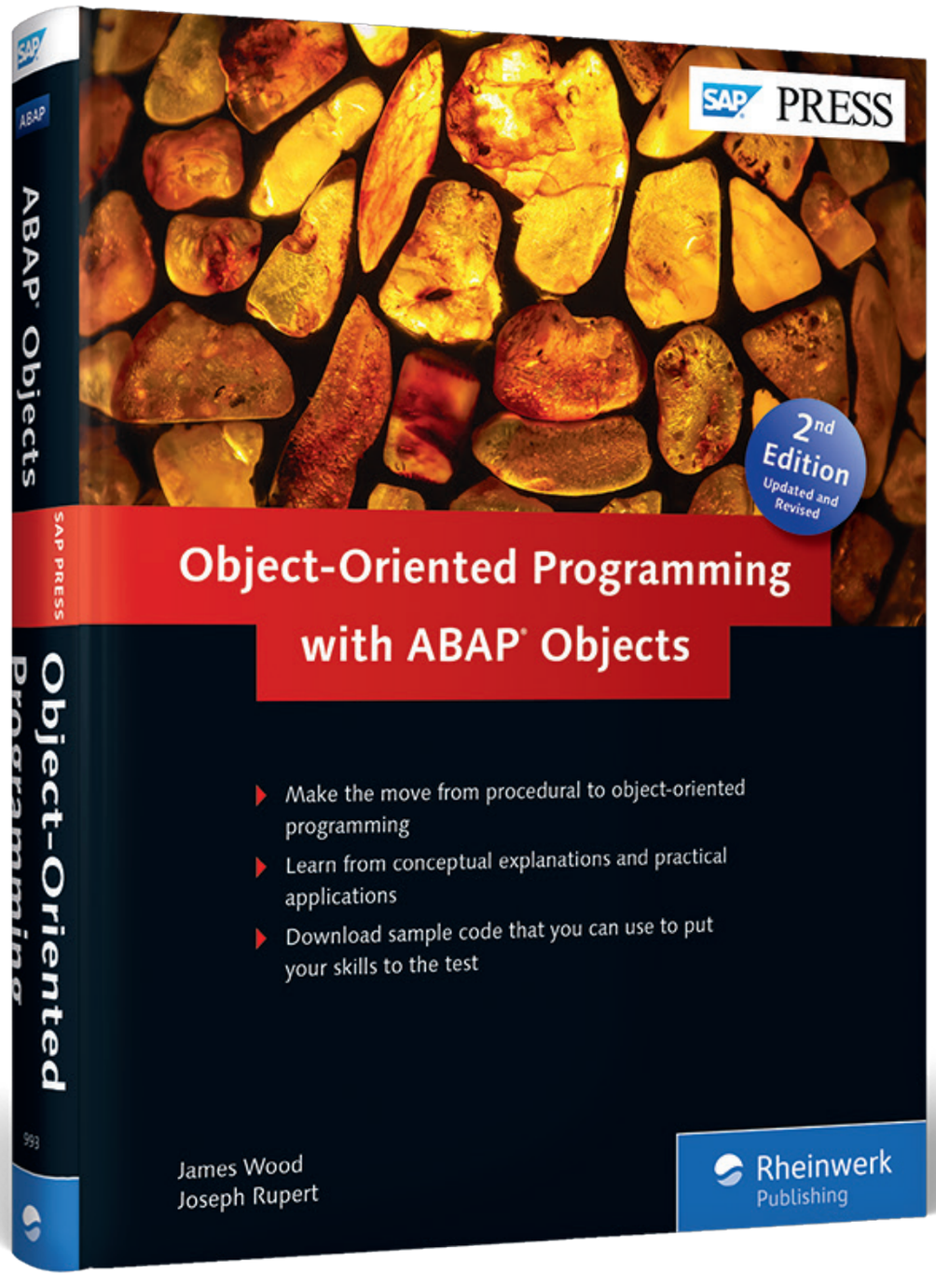


Reading Sample

This sample chapter explores encapsulation and implementation hiding techniques, which are used to simplify the way that developers interface with classes, making object-oriented program designs easier to understand, maintain, and enhance.

James Wood, Joseph Rupert

Object-Oriented Programming in ABAP Objects

470 Pages, 2016, $69.95/€69.95
ISBN 978-1-59229-993-5

www.sap-press.com/3597

Classes are abstractions that are used to extend the functionality of a programming language by introducing user-defined types. Encapsulation and implementation hiding techniques are used to simplify the way that developers interface with these user-defined types, making object-oriented program designs easier to understand, maintain, and enhance.

3 Encapsulation and Implementation Hiding

One of the most obvious ways to speed up the software development process is to leverage pre-existing code. However, while most projects strive to create reusable source code artifacts, few actually succeed in delivering modules that can be classified as *reusable*. In most cases, this lack of (re)usability can be traced back to the fact that the module(s) become too tightly coupled with their surrounding environment. With so many "wires" getting in the way, it's hard to pick up a module and drop it in somewhere else. Therefore, in order to improve reusability, we need to cut the cords and figure out ways of building autonomous components that can *think* and *act* on their own.

In this chapter, we'll learn how to breathe life into objects by exploring the benefits of combining data and behavior together under one roof. Along the way, we'll explore the use of access control mechanisms and see how they can be used to shape the *interfaces* of the defining classes to make them easier to modify and reuse in other contexts.

3.1 Lessons Learned from Procedural Programming

Contrary to popular belief, many core object-oriented programming concepts are based on similar principles rooted in the procedural programming paradigm. In both paradigms, the basic goal is to provide developers with the tools they need to translate requirements from the physical world into software-based solutions. However, while both programming models share in this goal, they go about

achieving it in vastly different ways. In this section, we'll take a closer look at the procedural approach and consider some of the limitations which ultimately caused many language designers to move in the direction of an object-oriented approach.

3.1.1 Decomposing the Functional Decomposition Process

Typically, procedural developers formulate their program designs using a process called *functional decomposition*. The term "functional decomposition" is taken from the world of mathematics, where mathematical functions are broken down into a series of smaller discrete functions that are easier to understand on their own. From a development perspective, functional decomposition refers to the process of *decomposing* a complex program into a series of smaller modules (e.g. procedures or subroutines).

One common approach for discovering these procedures is to scan through the functional requirements and highlight all the verbs used to describe the actions a program must take to meet its objectives. After all of the steps have been identified, they are then *composed* into a main program that's responsible for making sure that the procedures are executed in the right sequence. This process of organizing and refining the main program is sometimes called *step-wise refinement*.

For small to medium-sized programs, this strategy works pretty well. However, as programs start to branch out and grow in complexity, the design tends to become unwieldy as the main program becomes saddled with too many responsibilities. Here, besides keeping track of all of the different procedures and making sure that they're processed in the right order, the main program is also normally responsible for managing all of the data used by the various procedures. For this reason, such programs are often referred to as "God programs".

Note

In his book, *Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd Edition*, Alan Shalloway suggests that the term "God program" stems from the fact that only God can understand these programs.

With functional decomposition, the level of abstraction is the *subroutine*. Within a given subroutine definition, we can implement logic to perform a particular task using data that's provided from one of two places:

- Parameters that are passed into the subroutine from the calling program
- Global variables which are visible from within the subroutine

Regardless of the approach we use to supply subroutines with data, the reality is that there's no clean way of doing this without introducing some undesirable dependencies. For example, if we make liberal use of global variables, we open ourselves up to the possibility of data corruption errors. Here, imagine the impacts of switching out the call sequence of a pair of subroutines which make changes to the same global variable(s). If subroutine `b` depends on subroutine `a` to initialize the data and the call sequence gets flipped based on a requirements change, it's very likely that we'll start seeing strange data-related errors in the processing (see Figure 3.1).

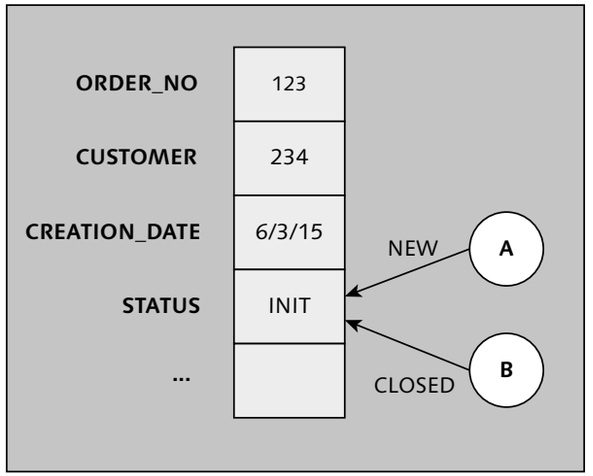


Figure 3.1 Data Collision Errors between Subroutines and Global Variables

Conversely, replacing global variables by passing around lots of parameters places additional burden on the main program to keep track of the parameters. Plus, we end up cluttering up the subroutine's parameter interface, which in turn leads to the tight coupling problem we described earlier.

Ideally we'd like for our modules to assume more responsibilities internally so that they are less reliant on controlling programs/modules when carrying out their tasks. Think of it this way, if we were to compare the organization of a software program with organizational (org) structures in an enterprise, which of the

two org structures depicted in Figure 3.2 and Figure 3.3 would we want our programs to look like? In the case of the flat org structure depicted in Figure 3.2, we have one centralized module that's responsible for (micro)managing lots of submodules. On the other hand, the tall org structure shown in Figure 3.3 is much more balanced with higher level modules delegating responsibilities down to specialized submodules.

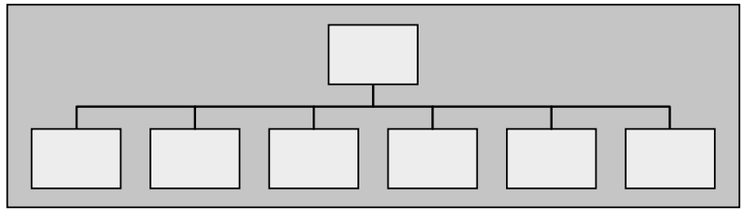


Figure 3.2 Example of a Flat Organizational Structure

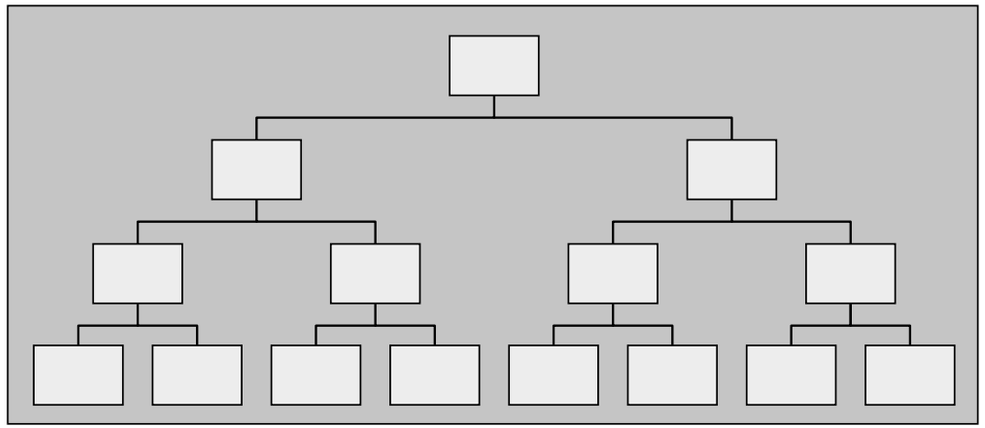


Figure 3.3 Example of a Tall Organizational Structure

In programming, just like business, it's important that we delegate responsibilities so that our programs remain flexible. In order for that to happen, the (sub)modules need to be smart enough to figure certain things out on their own—and that requires data. In the sections to come, we'll find that combining data and behavior together within a class helps us develop modules that can attain the kind of autonomy we're looking for.

3.1.2 Case Study: A Procedural Code Library in ABAP

To better illustrate some of the procedural programming challenges noted in Section 3.1.1, let's consider an example. In this section, we'll sketch out the development of a date utility library using ABAP function modules.

If you've worked with function modules before, then you know that they're defined within the context of a *function group*. In some respects, function groups bear some similarities to classes in that you can use them to define data and behaviors together within a self-contained unit (called a *function pool*). However, this analogy breaks down when you consider the fact that you cannot load multiple instances of a function group inside your program. This limitation makes it difficult for developers to work with the (global) data inside of a function group since additional logic is required to partition the data into separate work areas (or instances).

Because of this shortcoming, most function module developers tend to design their functions as *stateless* modules which operate on data that's maintained elsewhere. In this context, the term "stateless" implies that the function modules have no recollection of prior invocations and don't maintain any sort of internal state. As a result, function module developers need only worry about implementing the procedural logic—keeping track of the data/sessions is someone else's problem.

Note

Whenever you call a function module from a particular function group inside your program, the global data from the function group is loaded into the memory of the internal session of your program. Any subsequent calls to function modules within that function group will share the same global data allocated whenever the first function module was called.

Blame it on the BAPIs

The stateless approach to function module development increased in popularity quite a bit in the late 1990s/early 2000s whenever SAP started introducing *BAPIs* (the term "BAPI" stands for Business Application Programming Interface). At that time, SAP rolled out loads of function modules which promoted a stateless architecture. To call these BAPIs, one would generally have to define a slew of (global) variables that would be used to process BAPI calls. This is illustrated with the commonly used BAPI_USER_GET_DETAIL used to read user details. In the function signature shown in Figure 3.4, you can see that there's quite a bit of data about a user that has to be maintained outside of the function module. It's also interesting to note that the same variables would be needed to perform other operations on users such as create, change, and so forth.

Function Builder: Display BAPI_USER_GET_DETAIL

Pattern | Insert | Replace

Function module: BAPI_USER_GET_DETAIL — Active

Attributes | Import | Export | Changing | Tables | Exceptions | Source code

```
function bapi_user_get_detail.
*"----------------------------------------------------------------------
*"*"Lokale Schnittstelle:
*"  IMPORTING
*"     VALUE(USERNAME) LIKE  BAPIBNAME-BAPIBNAME
*"     VALUE(CACHE_RESULTS) TYPE  FLAG_X DEFAULT 'X'
*"  EXPORTING
*"     VALUE(LOGONDATA) LIKE  BAPILOGOND STRUCTURE  BAPILOGOND
*"     VALUE(DEFAULTS) LIKE  BAPIDEFAUL STRUCTURE  BAPIDEFAUL
*"     VALUE(ADDRESS) LIKE  BAPIADDR3 STRUCTURE  BAPIADDR3
*"     VALUE(COMPANY) LIKE  BAPIUSCOMP STRUCTURE  BAPIUSCOMP
*"     VALUE(SNC) LIKE  BAPISNCU STRUCTURE  BAPISNCU
*"     VALUE(REF_USER) LIKE  BAPIREFUS STRUCTURE  BAPIREFUS
*"     VALUE(ALIAS) LIKE  BAPIALIAS STRUCTURE  BAPIALIAS
*"     VALUE(UCLASS) TYPE  BAPIUCLASS
*"     VALUE(LASTMODIFIED) TYPE  BAPIMODDAT
*"     VALUE(ISLOCKED) TYPE  BAPISLOCKD
*"     VALUE(IDENTITY) TYPE  BAPIIDENTITY
*"     VALUE(ADMINDATA) TYPE  BAPIUSERADMIN
*"  TABLES
*"      PARAMETER STRUCTURE  BAPIPARAM OPTIONAL
*"      PROFILES STRUCTURE  BAPIPROF OPTIONAL
*"      ACTIVITYGROUPS STRUCTURE  BAPIAGR OPTIONAL
*"      RETURN STRUCTURE  BAPIRET2
*"      ADDTEL STRUCTURE  BAPIADTEL OPTIONAL
*"      ADDFAX STRUCTURE  BAPIADFAX OPTIONAL
*"      ADDTTX STRUCTURE  BAPIADTTX OPTIONAL
*"      ADDTLX STRUCTURE  BAPIADTLX OPTIONAL
*"      ADDSMTP STRUCTURE  BAPIADSMTP OPTIONAL
*"      ADDRML STRUCTURE  BAPIADRML OPTIONAL
*"      ADDX400 STRUCTURE  BAPIADX400 OPTIONAL
*"      ADDRFC STRUCTURE  BAPIADRFC OPTIONAL
*"      ADDPRT STRUCTURE  BAPIADPRT OPTIONAL
*"      ADDSSF STRUCTURE  BAPIADSSF OPTIONAL
*"      ADDURI STRUCTURE  BAPIADURI OPTIONAL
*"      ADDPAG STRUCTURE  BAPIADPAG OPTIONAL
*"      ADDCOMREM STRUCTURE  BAPICOMREM OPTIONAL
*"      PARAMETER1 STRUCTURE  BAPIPARAM1 OPTIONAL
*"      GROUPS STRUCTURE  BAPIGROUPS OPTIONAL
*"      UCLASSSYS STRUCTURE  BAPIUCLASSSYS OPTIONAL
*"      EXTIDHEAD STRUCTURE  BAPIUSEXTIDHEAD OPTIONAL
*"      EXTIDPART STRUCTURE  BAPIUSEXTIDPART OPTIONAL
*"      SYSTEMS STRUCTURE  BAPIRCVSYS OPTIONAL
*"----------------------------------------------------------------------
```

Figure 3.4 An Example of a Stateless BAPI Function

For the purposes of our date library example, we'll build our utility functions as stateless function modules. Within these functions, we'll operate on a date value represented by the SCALS_DATE structure shown in Figure 3.5. Here, though we

could have just as easily used the internal ABAP date (D) type, we elected to use a structure type so that we could clearly address the individual components of a date (e.g. month, day, or year) without using offset semantics.

Dictionary: Display Structure

Hierarchy Display

Structure: SCALS_DATE — Active

Short Description: General Date Structure

Attributes | Components | Entry help/check | Currency/quantity fields

Predefined Type — 1 / 4

Component	Typing Method	Component Type	Data Type	Length	Deci...	Short Description
YEAR	Types	SCAL_YEAR	NUMC	4	0	Calendar Year
MONTH	Types	SCAL_MONTH	NUMC	2	0	Calendar Month
DAY	Types	SCAL_DAY	NUMC	2	0	Calendar Day
CONTAINER	Types	SCAL_CONTAINER	CHAR	20	0	Data Container for Additional Date Entries

Figure 3.5 Modeling the Data Used for the Date Library

The code excerpt contained in Listing 3.1 sketches out the date API in a function group called ZDATE_API. Here, we've defined a handful of utility methods that can be used to perform date calculations, format dates according to different locales, and so forth.

```
FUNCTION-pool zdate_api.
FUNCTION z_add_to_date.
  * Local Interface IMPORTING VALUE (iv_days) TYPE i
  *                 CHANGING (cs_date) TYPE scals_date
  ...
ENDFUNCTION.
FUNCTION z_subtract_from_date.
  * Local Interface IMPORTING VALUE (iv_days) TYPE i
  *                 CHANGING (cs_date) TYPE scals_date
  ...
ENDFUNCTION.
FUNCTION z_get_day_name.
  * Local Interface IMPORTING VALUE (is_date) TYPE scals_date
  *                 EXPORTING ev_day TYPE string
  ...
ENDFUNCTION.
FUNCTION z_get_week_of_year.
  * Local Interface IMPORTING VALUE (is_date) TYPE scals_date
  *                 EXPORTING ev_week TYPE i
  ...
ENDFUNCTION.
```

```
FUNCTION z_format_date.
  * Local Interface IMPORTING VALUE (is_date) TYPE scals_date
  *                           VALUE (iv_format) TYPE csequence
  *                 EXPORTING ev_formatted TYPE string
  ...
ENDFUNCTION.
```

Listing 3.1 Building a Date Utility Library Using Function Modules

Within an ABAP program, we might use functions in the `ZDATE_API` function group to operate on date values being evaluated as part of a data processing routine like the contrived reporting example contained in Listing 3.2. With this kind of scenario in mind, in the upcoming sections we'll think about how our date API might stand up to maintenance requests that might pop up over time. This analysis will set the stage for Section 3.1.3 when we begin thinking about objects.

```
REPORT zsome_report.
START-OF-SELECTION.
  PERFORM get_data.

FORM get_data.
  DATA ls_date TYPE scals_date.
  DATA lt_itab TYPE STANDARD TABLE OF ...
  FIELD-SYMBOLS <ls_wa> LIKE LINE OF lt_itab.

  SELECT *
    INTO TABLE lt_itab ...

  LOOP AT lt_itab ASSIGNING <ls_wa>.
    ls_date = ...

    CALL FUNCTION 'Z_ADD_TO_DATE'
      EXPORTING
        iv_days = <ls_wa>-work_days
      CHANGING
        cs_date = ls_date.
    ...
    CALL FUNCTION 'Z_SUBTRACT_FROM_DATE'
      EXPORTING
        iv_days = <ls_wa>-offset
      CHANGING
        cs_date = ls_date.
    ...
    CALL FUNCTION 'Z_FORMAT_DATE'
      EXPORTING
        is_date = ls_date
        iv_format = `MM/DD/YYYY`
      IMPORTING
        ev_formatted = lv_formatted.
```



```
    ...
  ENDLOOP.
ENDFORM.
```

Listing 3.2 Incorporating the Date API into an ABAP Report Program

Expanding the Scope of the Date API

For the first scenario, imagine that we discover a need to expand the date API to also keep track of time. While this seems easy enough in principle, this could prove challenging since the structure used to model the date value doesn't contain components to capture a time stamp.

Looking at the `SCALS_DATE` structure in the ABAP Dictionary (in Figure 3.6), we discover that this structure cannot be enhanced/appended to. Maybe we could get away with using the unused `CONTAINER` field, but this wouldn't be obvious to developers who weren't intimately familiar with the internal workings of our date API.

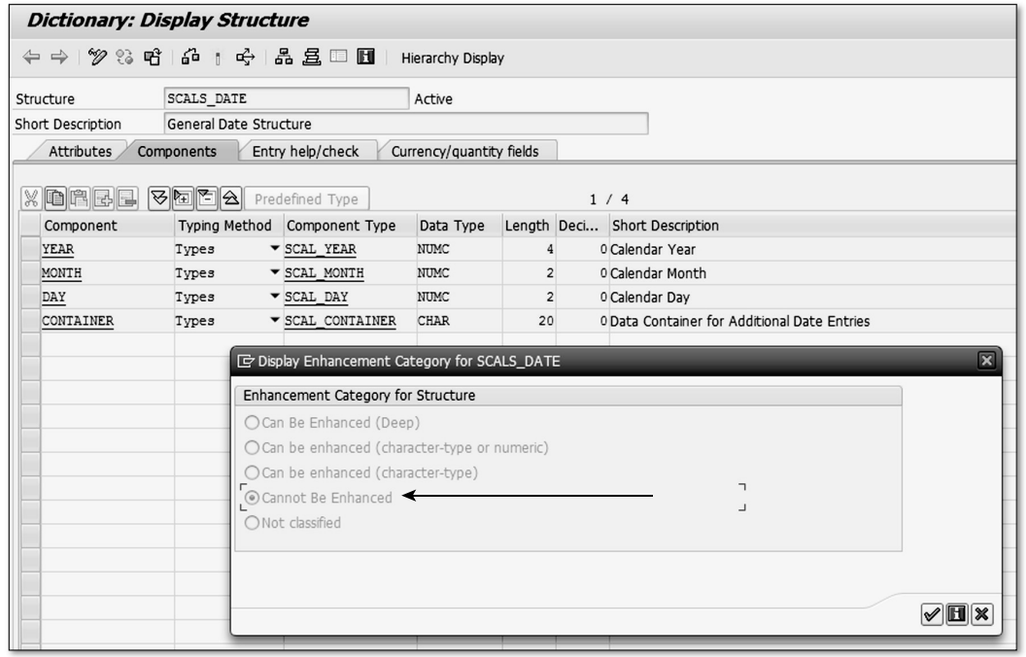


Figure 3.6 Looking at the Enhancement Category of the SCALS_DATE Structure

To implement this change correctly, we'd probably have to change the signature of our function modules to utilize a new structure. Besides requiring a fair

amount of rework within the functions themselves, this also requires that we make wholesale changes to the programs that call them.

Though you might be saying to yourself that the choice of the `SCALS_DATE` structure for the date API's data model was a poor one (and you're right to say so), that's really not the issue here. The point of this demonstration is to illustrate the fact that our date API exposes way too much information about its internal representation. Consumers of our date API shouldn't know (or care) whether we use the native ABAP date type (`D`), a structure, or something else entirely.

By exposing this kind of information in the function signatures, we've effectively coded ourselves into a corner. For better or worse, we have to stick with the design choices we've made and try our best to enhance around them. With stateless modules, this is about the best we can hope for.

Dealing with External Data Corruption

For the next scenario, imagine that you receive a defect report which indicates that the `Z_FORMAT_DATE` function is producing invalid output. After much investigation, you determine that the invalid output isn't a function of the logic in `Z_FORMAT_DATE`, but rather due to fact that an invalid day value has been specified in the `SCALS_DATE` structure's `DAY` field. Here, you discover that the invalid value is set within the calling report program which is accessing the `SCALS_DATE` structure outside of the `ZDATE_API` function group.

Though such errors might be easy to fix once you find them, they can be difficult to find. Since the `ZDATE_API` function group doesn't technically own the data, there's nothing stopping other modules from overwriting and/or corrupting the API's data model. In a perfect world, we'd like all accesses to the date API's data model to go through functions in the `ZDATE_API` function group so that we can isolate them and enforce the necessary validation rules (e.g. you can't have a date value of `20160231`). However, this is something the procedural model simply can't guarantee. To really enforce these rules, we need some support from the underlying language implementation to control access.

3.1.3 Moving Toward Objects

The `ZDATE_API` function group introduced in Section 3.1.2 is an example of an *abstract data type* (ADT). As the name suggests, ADTs are data types which provide an abstraction around some entity or concept (e.g. a date). Included in this abstraction is the data itself as well as a set of operations that can be performed on that data.

In order for ADTs to be effective, we must keep the data and operations as close to one another as possible. As we observed in Section 3.1.2, such cohabitation is virtually impossible to achieve with procedural programming techniques. Because of this divide, our date API (though admittedly contrived) was awkward to use and quite error prone. These problems become even more pronounced as the size and complexity of such code libraries expand.

In many ways, all of the problems we've considered in this section can be traced back to one central theme: poor support for data. While it would seem obvious that data is the foundation upon which any successful computer program runs, the stark reality is that data takes a back seat to actions in the procedural programming paradigm. As a result, procedural programs tend to decay at a much faster pace than programs built using programming models which place a greater emphasis on the data.

3.2 Data Abstraction with Classes

Recognizing many of the limitations outlined in Section 3.1, software researchers developed the OOP paradigm from the ground up with a strong emphasis on data *and* behavior. As you've already learned, classes are the vehicle that drives this equilibrium, encapsulating data (attributes) and behavior (methods) together inside a self-contained unit.

Encapsulation improves the organization of the code, making object-oriented class libraries much easier to understand and use than their procedural counterparts. To put this into perspective, consider the clumsiness of the function module-based date library we created in Section 3.1.2. Each time we accessed one of the API functions, we had to pass in an externally-managed structure which contained all of the date information needed to handle the request. Plus, if we wanted to work with multiple dates, then we had to define multiple variables and track those variables manually outside of the function group.

Let's compare that experience with a reimagined date API built using an ABAP Objects class. In Listing 3.3, we've created a class called `LCL_DATE` which provides

the same functionality of the ZDATE_API function group. As you look over the class definition, notice the simplification in the signature of the API methods. Instead of passing around an SCALS_DATE structure, the date information is being stored internally in an instance attribute called MS_DATE_INFO. Besides simplifying the interface, this design change also allows us to get out of the business of tracking date information externally. Now, our date API is truly an ADT which provides a *complete* abstraction around a date value as opposed to a loosely associated set of stateless function modules.

```
CLASS lcl_date DEFINITION.
  PUBLIC SECTION.
    DATA ms_date_info TYPE scals_date.
    METHODS:
      add IMPORTING iv_days TYPE i
          RETURNING VALUE(ro_date) TYPE REF TO lcl_date,
      subtract IMPORTING iv_days TYPE i
               RETURNING VALUE(ro_date) TYPE REF TO lcl_date,
      get_day_name RETURNING VALUE(rv_day) TYPE string,
      get_week_of_year RETURNING VALUE(rv_week) TYPE i,
      format IMPORTING iv_pattern TYPE csequence
             RETURNING VALUE(rv_date) TYPE string.
      ...
ENDCLASS.
```

Listing 3.3 Reimagining the Date Utilities API as an ABAP Objects Class

The code excerpt contained in Listing 3.4 demonstrates how we can work with our refactored date library. Once an LCL_DATE instance is created, we no longer have to worry about handling the date value. Instead, we can use methods like add() and subtract() to apply the changes in-place. From a code readability standpoint, this is much easier to follow because the context of an operation like add() is clearly the object referenced by lo_date.

```
DATA lo_date TYPE REF TO lcl_date.
DATA lv_message TYPE string.

CREATE OBJECT lo_date
  EXPORTING
    iv_date = '20150913'
lo_date->add( 30 ).
lv_message = |{ lo_date->subtract( 15 )->format( 'YYYYMMDD' ) }|.
```

Listing 3.4 Working with an OO-Based API

Ultimately, objects created in reference to encapsulated classes take on their own *identity*, allowing developers to start thinking about their designs in more conceptual terms (e.g. a date). Consumers of these classes don't have to worry about low-level implementation details; to the end user the LCL_DATE class is like a black box which performs various date manipulations. We don't have to supply the LCL_DATE class with lots of data/context/instructions; it intrinsically *knows* how to do its job.

In the next section, we'll learn how to round out ADTs like the LCL_DATE class by closing off access to internal components such as the MS_DATE_INFO attribute. This safeguard ensures that *all* operations on date values are mediated through API methods which rigorously validate incoming requests to ensure that the integrity of date values is maintained. As we'll see, this approach offers several important benefits.

3.3 Defining Component Visibilities

The term "encapsulation" refers to the idea of enclosing something inside of a *capsule*. The verbal imagery associated with words like "capsule" implies that we're setting some kind of boundary between the internal components of a class and the outside world. The purpose of this boundary is to protect (or hide) the inner mechanisms of the object that are sensitive to change. Most of the time, the most vulnerable parts of an object are its attributes since these define the object's state. However, in this book, we'll look at ways to hide *any* design decisions that are subject to change.

In this section, we'll describe the ABAP Objects language constructs that you can use to establish boundaries within your classes. Then, in the section that follows, we'll consider how to use these boundaries to build robust classes that can easily be adapted to ever-changing functional requirements.

3.3.1 Working with Visibility Sections

ABAP Objects provides three visibility sections for controlling access to the components defined within a class: the PUBLIC SECTION, the PROTECTED SECTION, and the PRIVATE SECTION. Within a CLASS DEFINITION statement, all component declarations must be defined within one of these three visibility sections. The code

excerpt contained in Listing 3.5 demonstrates the syntax used to define components within these sections.

```
CLASS lcl_visibility DEFINITION.
  PUBLIC SECTION.
    DATA x TYPE i.
  PROTECTED SECTION.
    DATA y TYPE i.
  PRIVATE SECTION.
    DATA z TYPE i.
ENDCLASS.
```

Listing 3.5 Working with Visibility Sections

As you might expect, components defined within the `PUBLIC SECTION` of a class are accessible from any context in which the class itself is visible (i.e., anywhere you can use the class type to declare an object reference variable). These components make up the *public interface* of the class.

Components defined within the `PRIVATE SECTION` of a class are only accessible from within the class itself. Note that this is more than just a mere suggestion; this is something that's strictly enforced by the ABAP compiler/runtime environment. For example, the code excerpt contained in Listing 3.6 would produce a compilation error because the `z` attribute of the `LCL_VISIBILITY` class is defined as a private attribute. The only way to get our hands on `z` is through a method defined in the `LCL_VISIBILITY` class.

```
DATA lo_visible TYPE REF TO lcl_visibility.
CREATE OBJECT lo_visible.
IF lo_visible->z GT 0.
  ...
ENDIF.
```

Listing 3.6 Attempting Access to Private Components of a Class

For now, we'll defer a discussion on the `PROTECTED SECTION` until we have a chance to cover inheritance in Chapter 5. For now, simply note that components defined in the `PROTECTED SECTION` are only accessible within a class and its subclasses.

When working in the form-based view of the Class Builder tool, you can assign components of global classes to visibility sections using the Visibility column highlighted in Figure 3.7.

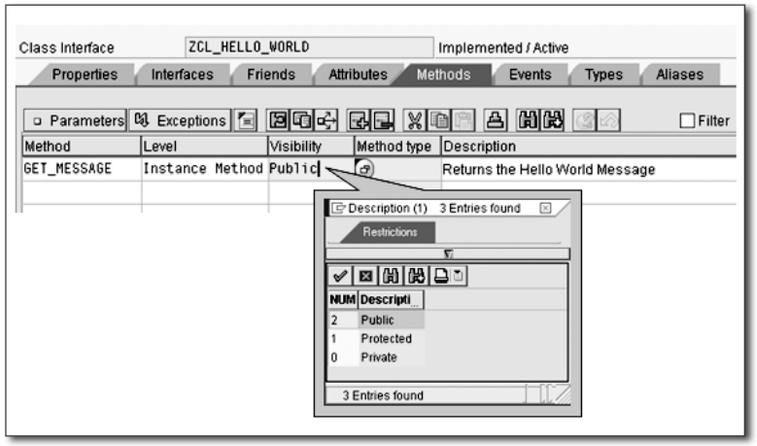


Figure 3.7 Setting the Visibility of Components Using the Form-Based View of the Class Builder Tool

Designing Across Multiple Dimensions

Choosing the right visibility section for a given component can be tricky, and it requires a fair amount of thought. Here, rather than thinking about the individual components, we need to think in terms of the class's overall interface. If we want to make our class simple and easy to use, then we'll need to strip down the public interface to just the essentials. This makes the interface less busy and therefore easier to consume.

In general, clients of a class should be on a "need-to-know" basis. In other words, if a client doesn't require direct access to a component, then there's no need for them to even be aware of its existence. Declaring such components within the `PRIVATE SECTION` of a class makes life easier for everyone: clients get to work with a simplified interface and the owners of the class have the freedom to change/improve the internal implementation of a class without fear of breaking existing client code.

With this concept in mind, we'd suggest that most attributes should be defined within the `PRIVATE SECTION` of a class. The primary reason for hiding attributes is to ensure that the state of the object cannot be tampered with haphazardly. If a client needs to update the state of an object, then they can do so through a method defined in the `PUBLIC SECTION`. The advantage of this kind of indirection

is that we can control the assignment of the attribute using business rules that are defined inside the method. This eliminates a lot of the guesswork in troubleshooting data-related errors since we know that any and all changes to an attribute are brokered through a single method. Methods that update the value of private attributes are sometimes called *setter* (or *mutator*) methods. To access these values (or formatted versions of these values), clients can invoke *getter* (or *accessor*) methods which broker access in the other direction.

This getter/setter method approach to indirect data access is demonstrated in the `LCL_TIME` class contained in Listing 3.7. Here, the state of the time object is being represented by three private attributes called `mv_hour`, `mv_minute`, and `mv_second`. Any updates to these attributes are controlled through setter methods such as `set_hour()` or `set_minute()`. Within these methods, we've included logic to ensure that the attributes remain consistent (e.g. we don't have an hour value of 113). Clients can obtain copies of these values by calling the corresponding getter methods (e.g. `get_hour()`).

```
CLASS lcl_time DEFINITION.
  PUBLIC SECTION.
    METHODS:
      set_hour IMPORTING iv_hour TYPE i,
      get_hour RETURNING VALUE(rv_hour) TYPE i,
      set_minute IMPORTING iv_minute TYPE i,
      get_minute RETURNING VALUE(rv_minute) TYPE i,
      set_second IMPORTING iv_second TYPE i,
      get_second RETURNING VALUE(rv_second) TYPE i.

  PRIVATE SECTION.
    DATA: mv_hour TYPE i,
          mv_minute TYPE i,
          mv_second TYPE i.
ENDCLASS.

CLASS lcl_time IMPLEMENTATION.
  METHOD set_hour.
    IF iv_hour BETWEEN 0 AND 23.
      me->mv_hour = iv_hour.
    ELSE.
      "TODO: Error handling...
    ENDIF.
  ENDMETHOD.

  METHOD get_hour.
    rv_hour = me->mv_hour.
  ENDMETHOD.
  ...
ENDCLASS.
```

Listing 3.7 Working with Getter and Setter Methods

As an alternative to the getter method approach, ABAP also allows us to define read-only attributes within a class definition. This is achieved using the `READ-ONLY` addition to the `DATA` keyword. The code excerpt below demonstrates how we might refactor the LCL_TIME class from Listing 3.7 to use this feature.

```
CLASS lcl_time DEFINITION.
  PUBLIC SECTION.
    DATA: mv_hour TYPE i READ-ONLY,
          mv_minute TYPE i READ-ONLY,
          mv_second TYPE i READ-ONLY.
  ...
ENDCLASS.
```

Listing 3.8 Defining Read-Only Attributes in a Class

While this feature can come in handy for simple classes which are primarily used for transferring data, we'd encourage you to use this option sparingly since it exposes the internal implementation details of your class.

3.3.2 Understanding the Friend Concept

In the previous section, we learned that components defined within the private and protected sections of a class are not visible outside of that class (or subclasses in the case of protected components). However, in some cases, it might be advantageous to be able to grant special access to certain classes of our choosing. Such classes are called *friends* of the class that grants them access.

Listing 3.9 illustrates the syntax used to create friend relationships between a defining class `CL_SOME_CLASS` and its friends: `C1`, `C2`, and so on. Here, the `FRIENDS` addition is added to a `CLASS DEFINITION` statement to declare this relationship up front to the ABAP compiler. As you can see, we can specify multiple friend classes after the `FRIENDS` addition (not to mention interfaces, which are covered in Chapter 6).

```
CLASS cl_some_class DEFINITION FRIENDS c1 c2 i3 i4.
  ...
ENDCLASS.
```

Listing 3.9 Defining Friendship Relationships in Classes

To demonstrate how friendship relationships work between classes, consider the example code contained in Listing 3.10. Here, we have a pair of classes called LCL_PARENT and LCL_CHILD which have entered into a friendship relationship. The LCL_CHILD class is taking advantage of this relationship by accessing the LCL_PARENT class's mv_credit_card_no attribute in a method called buy_toys(). Since mv_credit_card_no is defined as a private attribute, the only way for LCL_CHILD to access this value is through the friendship relationship. Without this addition, the code below would produce a syntax error.

```
CLASS lcl_child DEFINITION DEFERRED.
CLASS lcl_parent DEFINITION FRIENDS lcl_child.
  PRIVATE SECTION.
    DATA mv_credit_card_no TYPE string.
ENDCLASS.

CLASS lcl_child DEFINITION.
  PUBLIC SECTION.
    METHODS buy_toys.
ENDCLASS.
CLASS lcl_child IMPLEMENTATION.
  METHOD buy_toys.
    DATA: lo_parent TYPE REF TO lcl_parent,
          lo_store TYPE REF TO lcl_toy_store.
    lo_parent = ...
    lo_store = ...

    lo_store->checkout( lo_parent->mv_credit_card_no ).
  ENDMETHOD.
ENDCLASS.
```

Listing 3.10 Bypassing Access Control Using Friends

We can achieve the same effect for global classes maintained in the form-based view of the Class Builder tool by plugging the target friend classes on the Friends tab as shown in Figure 3.8.

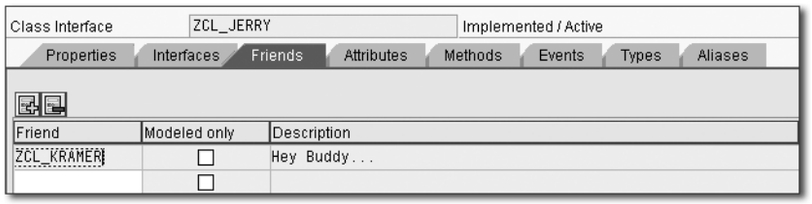


Figure 3.8 Defining Friendship Relationships Between Global Classes

As you begin working with friendship relationships, there are a couple of important things to consider. First of all, it's important to note the direction and nature of the friendship relationship. In Listing 3.10, class LCL_PARENT explicitly granted friendship access to class LCL_CHILD. This relationship definition is not reflexive. For example, it would not be possible for class LCL_PARENT to access the private components of class LCL_CHILD without the LCL_CHILD class granting friendship access to LCL_PARENT first. Secondly, notice that classes cannot arbitrarily declare themselves friends of another class. For instance, it would not be possible for class LCL_CHILD to surreptitiously declare itself a friend of class LCL_PARENT. If this were the case, access control would be a waste of time since any class could bypass this restriction by simply declaring themselves a friend of whatever class they were trying to access.

The example shown in Listing 3.10 also introduced a new addition to the CLASS DEFINITION statement that we have not seen before: the DEFERRED addition. In a scenario like this, the DEFERRED addition used in the first CLASS DEFINITION statement for LCL_CHILD is needed to instruct the compiler of the existence of the LCL_CHILD class in the CLASS DEFINITION statement for the LCL_PARENT class. Without this clause, the compiler would have complained that class LCL_CHILD was unknown whenever we tried to establish the friendship relationship in the definition of class LCL_PARENT.

To Friend or Unfriend

Many purists argue that the use of friends should not be allowed in object-oriented languages since they bypass traditional access control mechanisms. Whether you agree with this sentiment or not, we would recommend that you use friendship relationships sparingly in your designs because it truly is rare that you would need to open up access like this.

3.4 Designing by Contract

As we've learned, encapsulation and implementation hiding techniques can be used to define very precise public interfaces for a class. These interfaces help to form a *contract* between the developer of a class and users of that class. The contract metaphor is taken from the business world, where customers enter into contractual agreements with suppliers providing goods or services. In his book, *Object-Oriented Software Construction*, Bertrand Meyer described how this con-

cept could be adapted into object-oriented software designs in order to improve the reliability of software components that are "...implementations meant to satisfy well-understood specifications."

In this context, objects are subject to a series of *invariants* (or constraints) that specify the valid states for the object. To maintain these invariants, methods are defined using *preconditions* (what must be true before the method is executed) and *postconditions* (what must be true after the method is executed). In Chapter 8, we'll look at ways to deal with exceptions to these rules.

The primary goal when applying the *Design by Contract* approach in your software designs is to produce components that deliver *predictable* results. The boundaries set by the visibility sections ensure that *loopholes* are not introduced into the contract. For instance, the date library that we first introduced in Section 3.1.2 had many loopholes that made it possible to bypass the business rules implemented inside the function module(s). The encapsulation techniques we applied in the class-based reimplementation of this library eliminated these loopholes by encapsulating the date data as a private attribute that's cut off from external tampering.

Client programmers using classes based on these principles know what to expect from the class based on the provided public interface. Similarly, class developers are free to change the underlying implementation so long as they continue to honor the contract outlined in the public interface. Over time, the duel nature of this relationship helps to increase trust as we accumulate reusable modules that clients know will work.

3.5 UML Tutorial: Sequence Diagrams

So far, our study of the UML has been focused on diagrams that are used to describe the static architecture of an object-oriented system. In this chapter, we will introduce the first of several *behavioral diagrams* that are used to illustrate the behavior of objects at runtime. The *sequence diagram* depicts a message sequence chart between objects that are interacting inside a software system.

Figure 3.9 shows a simple sequence diagram that is used to illustrate a cash withdrawal transaction in an ATM machine. A sequence diagram is essentially a graph in two dimensions. The various objects involved in the interaction are aligned along the horizontal axis. The vertical axis represents time. Sequence diagrams

are initiated by a request message from some kind of external source. In the example in Figure 3.9, the external source is a user interfacing with the ATM machine. This initial message is called a *found message*. In object-oriented terms, a message is analogous to a method call. Messages are sent to objects (depicted in the familiar object boxes seen on the object diagrams described in Chapter 2). The dashed line protruding from underneath the object box represents the object's *lifeline*.

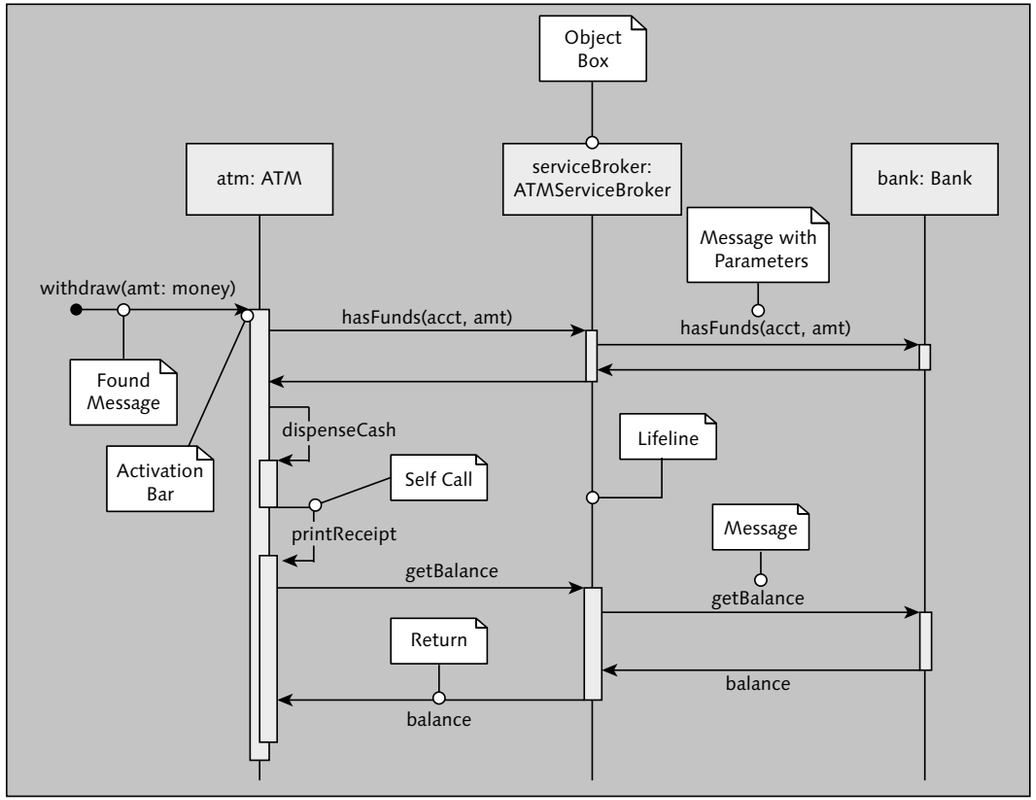


Figure 3.9 Sequence Diagram for Withdrawing Cash from an ATM

The intersection of a message and an object's lifeline is depicted with a thin rectangular box called an *activation bar*. The activation bar shows when an object is active during the interaction. Objects are activated via messages (i.e. method calls). Messages can include parameters that help clarify the operation to be performed by the object. However, it's not a good idea to try and fully specify the

method interface in a sequence diagram—that's what a class diagram is for. Here, we only use parameters for emphasis or clarity. Synchronous method calls can have a *return* message that can also have optional parameters.

In some cases, a method might need to call other local helper methods to complete its task. In this case, a *self call* can be illustrated by drawing a circuitous arrow to another activation bar that is stacked on top of the current activation bar. For example, in Figure 3.9, messages `dispenseCash` and `printReceipt` are both represented as self calls on the `atm` object inside method `withdraw`.

Sequence diagrams are very useful for explaining complex interactions where the order of operations is difficult to follow. One of the reasons that sequence diagrams are so popular is that the notation is very intuitive and easy to read. To maintain this readability, it's important to avoid cluttering a sequence diagram with too many interactions. In the coming chapters, we'll look at other types of interaction diagrams that can be used to illustrate fine-grained behavior within an object or more involved interactions that span multiple use cases.

3.6 Summary

In this chapter, you learned about the many advantages of applying encapsulation and implementation hiding techniques to your class designs. Encapsulating data and behavior in classes simplifies the way that users/clients work with classes. Hiding the implementation details of these classes strengthens the design even further, making classes much more resistant to change and/or data corruption. The combination of these two design techniques helps you to design intelligent classes that are highly self-sufficient. Such classes are easy to reuse in other contexts since they are loosely coupled to the outside world.

In the next chapter, we'll examine the basic lifecycle of an object. We'll also learn about special methods called *constructors* that can be used to ensure that object instances are always created in a valid state.

Contents

PART II Case Studies

Index

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

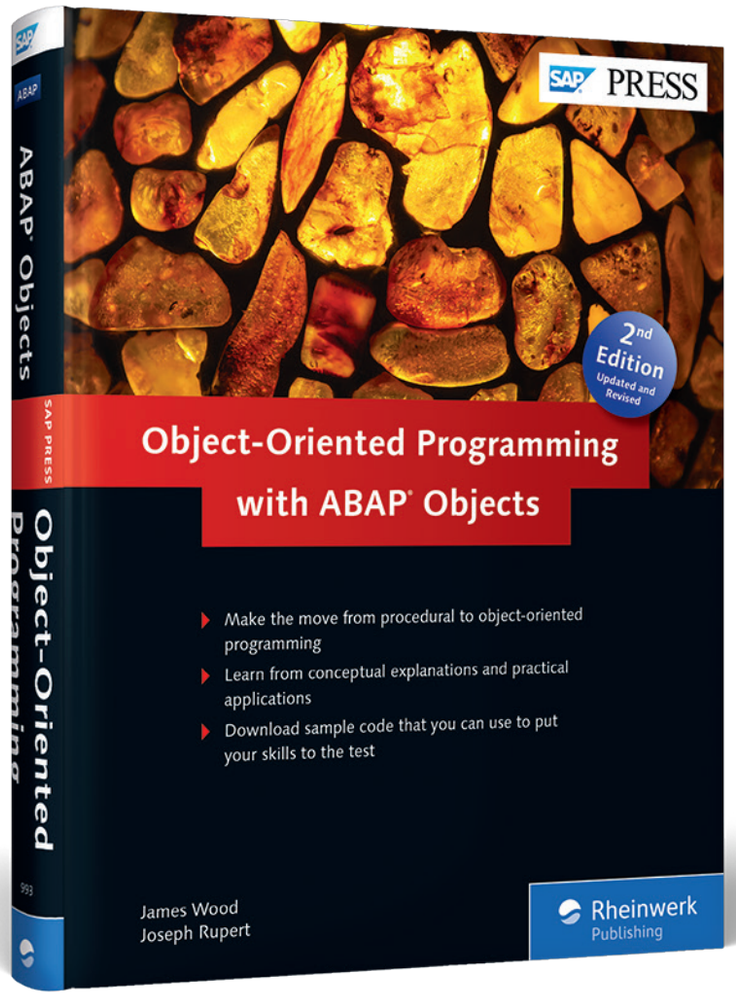
U

V

W

X

Z



James Wood, Joseph Rupert

Object-Oriented Programming in ABAP Objects

470 Pages, 2016, $69.95/€69.95
ISBN 978-1-59229-993-5

www.sap-press.com/3597

James Wood is the founder and principal consultant of Bowdark Consulting, Inc., a consulting firm specializing in technology and custom development in the SAP landscape. Before starting Bowdark in 2006, James was an SAP NetWeaver consultant for SAP America, Inc. and IBM Corporation, where he was involved in many large-scale SAP implementations. James is also an SAP Mentor and author of several best-selling SAP titles. To learn more about James and this book, please check out his website at www.bowdark.com.

Joseph Rupert is a senior technical consultant at Bowdark Consulting, Inc. Before joining Bowdark, Joe worked for several health care technology companies building complex search engines for querying biomedical research, patient lab and clinical data.