In this chapter, you’ll learn about clean ABAP guidelines for a fundamental building block in your ABAP code: methods. You’ll see how methods work with an object-oriented design and follow best practices for method parameters, readability, and invocation.

Clean ABAP: A Style Guide for Developers
Klaus Haeuptle, Florian Hoffmann, Rodrigo Jordão, Michel Martin, Anagha Ravinarayan, Kai Westerholz
351 Pages, 2021, $79.95
ISBN 978-1-4932-2026-7
www.sap-press.com/5190
Methods are the ultimate containers of executable code in ABAP Objects. Other types of structures that directly contain code are function modules, programs, and subroutines, but as discussed in Chapter 2, Section 2.3, methods are the preferred low-level building blocks for clean ABAP code.

Many other chapters in this book describe detailed clean code patterns and recommendations for ABAP code within methods. This chapter deals with the design, declaration, and usage of methods, high-level control flow within methods, method body guidelines; and invoking methods. We’ll start with an in-depth discussion on methods in the context of object-oriented programming in Section 4.1. Then, in Section 4.2, we’ll dive deeply into method parameters and their use before turning, in Section 4.3, to clean code principles to apply to the method body. To finish this chapter, we’ll show you how to call your methods cleanly in Section 4.4.

4.1 Object-Oriented Programming

In this section, we’ll discuss how to make methods work better within an object-oriented design, picking up on our earlier discussion of object-oriented programming in Chapter 3, Section 3.1.

4.1.1 Static and Instance Methods

Static methods are methods declared with the CLASS-METHODS keyword and are attached to the class itself, not to instances of the class. To call a static method, you’ll use the class name directly, without needing to create an instance of that class. In this way, static methods are not truly object-oriented. They do not participate in the class inheritance hierarchy and do not participate in dynamic dispatch (thus, the term “static”). Listing 4.1 shows you how to declare a static method.
4 Methods

CLASS blog_post DEFINITION
  PUBLIC.
PUBLIC SECTION.
CLASS-METHODS:
  publish.
ENDCLASS.

Listing 4.1 Static Method Definition

You can call this method using the class name directly, followed by a fat arrow (=>):
blog_post=>publish( ).

Since static methods constrain the behavior to a specific implementation, they are not flexible.

Instance methods, meanwhile, are attached to instances of the class. Instance methods are declared with the METHODS keyword, as shown in Listing 4.2.

CLASS blog_post DEFINITION
  PUBLIC.
PUBLIC SECTION.
METHODS:
  publish.
ENDCLASS.

Listing 4.2 Instance Method Definition

To call an instance method, you need an instance of the class, followed by a thin arrow (->):
DATA my_blog_post TYPE REF TO blog_post.
...
my_blog_post->publish( ).

Consider Using Instance Methods by Default

The key to the flexibility of instance methods is the fact that they are attached to instances of classes. At runtime, a specific instance could refer to any subclass in the hierarchy of the declared class variable (including itself), which makes the method call “dynamic” or “virtual.”

Also, the behavior of instance methods can be redefined and more easily mocked in unit tests. Their data scope is the scope of the class instance, and resources can be configured per instance. For these reasons, we recommend using instance methods by default, rather than static methods.

Look at the related discussion involving classes in Chapter 3, Section 3.1.2.

Methods should be static in the following cases: for static creation methods and for utility methods.

Static creation methods are methods that work as specialized constructors (as described earlier in Chapter 3, Section 3.3.3). These methods return an instance of the owning class or a subclass, as shown in Listing 4.3, where the create_as_copy method serves as a static creation method by creating a copy of the passed in source_blog_post instance.

CLASS blog_post DEFINITION
  PUBLIC.
PUBLIC SECTION.
CLASS-METHODS:
  create_as_copy
IMPORTING
  source_blog_post TYPE REF TO blog_post
RETURNING
  VALUE(result) TYPE REF TO blog_post.
...
ENDCLASS.

Listing 4.3 Static Creation Method Example

Utility methods are methods that don’t depend on any underlying resources and perform a static operation that is not reasonably expected to change and does not need any parameterized behavior. These methods should usually be part of a utility class that provides related operations (see Chapter 3, Section 3.1.2). An example of the definition of a utility method is shown in Listing 4.4, where the fahrenheit_to_celsius method performs a simple temperature conversion. This kind of operation does not require any variation and is perfectly fine as a utility method.

CLASS temperature_conversion DEFINITION
  PUBLIC
ABSTRACT
FINAL.
4.1 Object-Oriented Programming

Public Instance Methods Should Be Part of an Interface

For a class to serve as a good citizen in an overall design, the class should implement interfaces that define its abstract behavior. Consequently, most, if not all, of its public instance methods should be part of an interface.

In this way, the class itself becomes a mere implementation of the concepts represented by the interfaces it implements, and consumers don’t need to depend directly on the class at all. This important step enables the dependency inversion principle, the D in SOLID.

Not all classes need to implement interfaces. Exception classes, value objects (see Chapter 3, Section 3.1.2), and enumeration classes (see Chapter 6, Section 6.2.2), for example, usually don’t need to present an abstraction over their operations and, as such, aren’t usually backed by interfaces. In general, classes that represent values or entities that only contain data, and not services, won’t implement interfaces.

For more information about interfaces and their advantages, refer to Chapter 3, Section 3.1.1.

Let’s revisit and improve the design of the thermal_switch class presented in Chapter 3, Section 3.1.2, specifically in Listing 3.15. The thermal_switch class’s main service is to protect a wrapped device from overheating. Its methods are trip (to trip the switch and turn the device off) and reset (to reset the switch to a normal state), which were implemented as direct public instance methods on the class.

To make those methods into interface methods and better abstract consumers of a thermal protector from the specific implementation present in the thermal switch, we can extract the interface thermal_protector, as shown in Listing 4.6.

```plaintext
INTERFACE thermal_protector
   PUBLIC.
   METHODS:
      trip
         FOR EVENT critical_temperature_reached OF thermal_sensor, reset.
   ENDINTERFACE.
```

Don’t Call Static Methods through Instance Variables

It is possible to call a static method through an instance of a class. However, a static method is attached to the class itself, and calling it through an instance is redundant and a potential source of confusion.

Make the fact that you’re calling a static method clear by always qualifying the method call with the class name.

4.1.2 Public Instance Methods

Public instance methods in a class effectively define its interface to outside consumers. Anyone with access to the class can call that class’s public instance methods.

To better manage dependencies and improve isolation, you should decouple from concrete classes and focus on the abstraction that the class presents. In this way, replacing implementation details with alternative implementations, like mock instances for testing, will be easier.

```plaintext
CLASS temperature_conversion IMPLEMENTATION.
   METHOD fahrenheit_to_celsius.
   ENDMETHOD.
ENDCLASS.
```

Listing 4.5 Static Method Implementation

PUBLIC SECTION.

CLASS METHODS fahrenheit_to_celsius
   IMPORTING
      temperature_in_fahrenheit TYPE temperature
   RETURNING
      VALUE(result) TYPE temperature.

ENDCLASS.

Listing 4.4 Utility Method Example

Both static and instance methods are implemented with the METHOD ... ENDMETHOD construct in the class implementation area, as shown in Listing 4.5 for the static method temperature_conversion=>fahrenheit_to_celsius.

```plaintext
CLASS METHODS fahrenheit_to_celsius
   IMPORTING
      temperature_in_fahrenheit TYPE temperature
   RETURNING
      VALUE(result) TYPE temperature.

ENDCLASS.
```

Listing 4.4 Utility Method Example

To better manage dependencies and improve isolation, you should decouple from concrete classes and focus on the abstraction that the class presents. In this way, replacing implementation details with alternative implementations, like mock instances for testing, will be easier.
CLASS thermal_switch DEFINITION
FINAL
PUBLIC.

PUBLIC SECTION.

INTERFACES:
  switchable,
  thermal_protector.

ALIASES:
  trip FOR thermal_protector~trip,
  reset FOR thermal_protector~reset.

METHODS:
  constructor
    IMPORTING
      managed_device TYPE REF TO switchable
      sensor TYPE REF TO thermal_sensor.

...
The step_fan class shown in Listing 4.8 inherits from fan and redefines the methods increase_intensity and decrease_intensity to execute them in a certain number of steps. Note how the methods are declared again with the REDEFINITION keyword. The redefined methods can then call the corresponding superclass version with the super-> syntax. Note also that the constructor is required to call the superclass constructor as its first statement.

Be Careful When Redefining Methods
As explored in Chapter 3, Section 3.1.2, inheritance is a powerful tool that is often hard to get right.
When you have method redefinitions, some behavior present in the superclass is adjusted to fulfill a new requirement. This adjustment could break clients' assumptions about the superclass, which violates the Liskov substitution principle, the L in SOLID.
Minimize the number of methods that are redefined. Strive to only implement methods that are abstract in the superclass. This approach makes the design easier to reason about since each class has only a single version of each method. This kind of hierarchy is called a normalized hierarchy in the book Working Effectively with Legacy Code (Prentice Hall, 2004).

4.2 Parameters
Method parameters fall broadly into three kinds or categories: input, output, and input-output parameters.

Input parameters are declared with the IMPORTING keyword. Output parameters can be declared as EXPORTING parameters or as a single RETURNING parameter. Input-output parameters are declared with the CHANGING keyword.

One common source of confusion for novice ABAP developers is that, apart from the CHANGING keyword, these keywords have other keywords as counterparts when invoking methods, as listed in Table 4.1.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORTING</td>
<td>EXPORTING</td>
</tr>
<tr>
<td>EXPORTING</td>
<td>IMPORTING</td>
</tr>
<tr>
<td>RETURNING</td>
<td>RECEIVING</td>
</tr>
<tr>
<td>CHANGING</td>
<td>CHANGING</td>
</tr>
</tbody>
</table>

Table 4.1 Parameters Declaration and Invocation Counterparts

Some invocation keywords are optional depending on how the method is declared and called. In this section, we'll cover guidelines on parameter declarations. Section 4.4 will discuss guidelines for method invocation.

4.2.1 How Many Input Parameters Are Too Many?
Let's look at the input parameters for the method declaration shown in Listing 4.9.

METHODS add_item
IMPORTING:
    product_id  TYPE product_id
    product_group  TYPE product_group
    amount  TYPE int4
    unit  TYPE unit_of_measure.

Listing 4.9 Too Many Input Parameters?
The add_item method shown in Listing 4.9 has many parameters that describe separate concepts, all lumped together as input parameters. Creating new structures or classes for these concepts makes the method clearer and easier to test and improves the overall design, as shown in Listing 4.10.

METHODS add_item
IMPORTING:
    product  TYPE REF TO product
    quantity  TYPE quantity.

Listing 4.10 Minimized Number of Input Parameters
Too many input parameters may also be an indication that the method does more than one thing. More information on this topic is found in Section 4.3.1.

Aim for as Few Input Parameters as Possible
The number of input parameters for a method can significantly affect how usable and understandable the method is to callers, the complexity of its internal implementation, and how hard it is to test it. In general, the less input parameters a method has, the easier it is to consume it, understand its operation, and test it.

Minimize the number of IMPORTING parameters to your methods. Most methods should have less than three input parameters. Three or more parameters makes the sheer number of possible parameter value combinations harder to use and test.

4.2.2 Optional Input Parameters
ABAP does not support method overloads, which occur when methods with the same name exist that only differ in their parameters. As a result, optional parameters, which
are parameters marked with the modifier `OPTIONAL`, have historically been used to control different behaviors within methods.

A typical example is a method that processes either a table of items or a single item, as shown in Listing 4.11.

```java
METHODS process
  IMPORTING
    item_to_process TYPE item OPTIONAL.
    items_to_process TYPE item_table OPTIONAL.
```

Listing 4.11 Mutually Exclusive Optional Parameters

This kind of method signature is confusing to the caller and harder to implement. What is not clear is which parameters are required, which combinations of parameters are valid, and which parameters are mutually exclusive. The implementation must check for all valid combinations and fail or branch accordingly.

**Split the Behavior Instead of Adding Optional Parameters**

Different behaviors should be implemented in different methods (see Section 4.3.1). Separate methods with properly chosen names and specific parameters for each case should be created. Using separate methods provides clear guidance for which parameter combinations are valid and expected and are easier to implement.

For the example shown in Listing 4.11, separating the functionality results in the methods shown in Listing 4.12.

```java
METHODS:
  process_item
    IMPORTING
      item_to_process TYPE item,

  process_items
    IMPORTING
      items_to_process TYPE item_table.
```

Listing 4.12 Separate Methods for Separate Functionalities

Default parameter values, which are accomplished by annotating a parameter with the `DEFAULT` modifier and including a default value, should be used to advertise that the parameter is optional for the caller, but not for the method, which will use the default value if one is not provided. For an example of a default parameter, look at the `to_csv` method in Listing 4.13. The parameter `separator` is optional and, if not assigned by the caller, will have the value `";"` (a comma) by default.

```java
METHODS to_csv
  IMPORTING
    separator TYPE string DEFAULT ";".
  RETURNING...
```

Listing 4.13 Parameter with Default Value

**4.2.3 Preferred Input Parameters**

If all input parameters to a method are optional, either by marking them as `OPTIONAL` or using a default value, then one of them can be annotated as a `PREFERRED PARAMETER`, as shown in Listing 4.14.

```java
METHODS to_csv
  IMPORTING
    separator TYPE string DEFAULT ";" PREFERRED PARAMETER
    quote type string DEFAULT "".
  RETURNING...
```

Listing 4.14 Method with Preferred Parameter

When calling a method with a parameter marked as a `PREFERRED PARAMETER` and providing a single parameter, you don’t need to name the parameter at the call site; the preferred parameter will automatically be used. Thus, when calling the `to_csv` method defined in Listing 4.14 with a single parameter, that parameter, if not named, will be bound to the preferred parameter, for instance, in the following call:

```java
DATA(item_as_csv) = item->to_csv( ";" ).
```

Although this style makes the call more concise, which parameter is being supplied might not be clear, thus making the code harder to understand. Even if unnecessary, naming the parameter can increase readability, such as in the following code:

```java
DATA(item_as_csv) = item->to_csv( separator = ";" ).
```

If the parameter is not marked as a `PREFERRED PARAMETER`, the caller will be forced to name the parameter when calling it.

**Use a Preferred Parameter Sparingly**

Only use a `PREFERRED PARAMETER` if the method name and its preferred parameter are clear and easy to understand when the method is being called, especially when omitting the parameter name (see also Section 4.4.4).
4.2.4 Boolean Input Parameters

Imagine you have a credit score service class that provides a method for calculating the credit score of a customer. The definition of this class is shown in Listing 4.15, where the `calculate_credit_score` method uses a Boolean parameter to control whether to use strong heuristics in the calculation.

CLASS credit_score_service DEFINITION
  PUBLIC FINAL.
  PUBLIC SECTION.
  METHODS:
    constructor IMPORTING customer TYPE REF TO customer,
    calculate_credit_score IMPORTING use_strong_heuristics TYPE abap Bool
      RETURNING VALUE(credit_score) TYPE int4.
  PRIVATE SECTION.
  DATA:
    customer TYPE REF TO customer.
  ENDCLASS.
Listing 4.15 Credit Score Service with a Method with a Boolean Parameter

The `calculate_credit_score` method can be called in the following way:

```
DATA(my_credit_score) =
  my_credit_score_service->calculate_credit_score( abap_true ).
```

In this code, what that Boolean parameter means in the call is not clear. You can alleviate this problem by naming the parameter (see Section 4.4.4), as in the following code:

```
DATA(my_credit_score) =
  my_credit_score_service->calculate_credit_score( use_strong_heuristics = abap_true ).
```

This version improves things a bit even though the burden is on the client to use this style. Moreover, an issue with the semantics of the method still exists.

It’s not hard to imagine that the implementation of `calculate_credit_score` will look something like the code shown in Listing 4.16.

```
METHOD calculate_credit_score.
  IF use_strong_heuristics = abap_true.
    " use a slower but more robust algorithm....
  ELSE.
    " use a faster but less robust algorithm....
  ENDIF.
ENDMETHOD.
Listing 4.16 Branching Implementation

When a method takes a Boolean input parameter, usually the method does two things instead of one (see also Section 4.3.1).

Create Separate Methods Instead of Using Boolean Parameters

Using Boolean parameters to control decisions and branching inside of a method’s body makes the method more complex and harder to use and maintain. Different behaviors should be implemented in different methods with properly chosen names.

In our example, the `calculate_credit_score` method should be broken up into two methods, each focused on a different algorithm, as shown in Listing 4.17.

```
METHODS:
  calculate_simple_credit_score
    RETURNING VALUE(credit_score) TYPE int4,
  calculate_complex_credit_score
    RETURNING VALUE(credit_score) TYPE int4.
Listing 4.17 Separate Methods Instead of a Branching Method

In this way, not only is implementing each method individually easier, consuming those methods separately is also easier. If needed, common code within the original method could be refactored into supporting private methods.

You can even go a step further and create separate classes for each scenario. For our credit score service, we could have an interface with a simple `calculate_credit_score` method with a separate class implementing each algorithm. Thus, even the decision of which algorithm to use is lifted from client code. The client code itself would only depend on the interface and its simpler method.
4.2.5 EXPORTING Parameters

Results that a method provides to the caller can be declared either as EXPORTING parameters or a single RETURNING parameter.

EXPORTING parameters allow you to define multiple outputs from a method, as shown in Listing 4.18.

METHODS check_business_partners
  IMPORTING
    business_partners TYPE business_partners
  EXPORTING
    result TYPE result_type
    failed_keys TYPE /bobf/t_frw_key
    messages TYPE /bobf/t_frw_message.

Listing 4.18 Multiple EXPORTING Parameters

The check_business_partners method can validate a list of business partners provided in the input parameter business_partners. The outcome of this validation is returned in three output parameters: result is the overall status of the validation, failed_keys is a table of the business partner keys that failed validation, and messages is a list of messages generated as part of the validation.

EXPORTING parameters are optional for the caller, so when calling check_business_partners, you can capture the output parameters that you’re interested in, as shown in Listing 4.19.

    check_business_partners(
      EXPORTING
        business_partners = my_business_partners
    IMPORTING
      result = DATA(business_partners_check_result)
      messages = DATA(business_partner_messages) ).

Listing 4.19 Invoking a Method with EXPORTING Parameters

Note how you need to provide the counterpart keywords to the declaration of IMPORTING and EXPORTING parameters, which flips them to EXPORTING and IMPORTING, respectively.

Minimize the Number of EXPORTING Parameters

The presence of many EXPORTING parameters is an indication that your method might be doing more than one thing, which makes the method harder to implement and harder to consume. A good method does one thing (Section 4.3.1), and that sharp focus should be reflected by the outputs of the method as well.

For the check_business_partners method, the output parameters form a logical entity, a check_result, which could be made explicit as a check_result structure. Then, the check_result structure can be used to type the now single EXPORTING parameter of the method, as shown in Listing 4.20.

    TYPES:
      BEGIN OF check_result,
        result TYPE result_type,
        failed_keys TYPE /bobf/t_frw_key,
        messages TYPE /bobf/t_frw_message,
      END OF check_result.

    METHODS check_business_partners
      IMPORTING
        business_partners TYPE business_partners
      EXPORTING
        result TYPE check_result.

Listing 4.20 Minimizing EXPORTING Parameters

Note that this approach is still not the cleanest way to declare check_business_partners, which we’ll explore next.

4.2.6 RETURNING Parameters

RETURNING parameters allow you to declare a single RETURNING parameter for a method, which can be thought of as the result of the method call.

The last iteration of our check_business_partners method shown in Listing 4.20 ended up with a single EXPORTING parameter. Calling that method is shown in Listing 4.21.

    check_business_partners(
      EXPORTING
        business_partners = my_business_partners
      IMPORTING
        result = DATA(business_partners_check_result) ).

Listing 4.21 Calling a Method with a Single EXPORTING Parameter
Prefer RETURNING to EXPORTING

As you minimize the number of EXPORTING parameters, when you end up with a single EXPORTING parameter, make this parameter a RETURNING parameter.

A RETURNING parameter is a more conventional way to declare an output parameter. This kind of parameter makes calls to the method shorter and allows for method chaining (when you chain a method call to the result of another method call without assigning the return value to any intermediate variable).

Some performance recommendations advise using EXPORTING parameters passed by reference for large tables of data (described in more detail in Section 4.2.8). However, using RETURNING parameters for these cases is usually acceptable. If you have proof of poor performance and of a tangible benefit, you can resort to the more cumbersome procedural style that comes with using EXPORTING. As usual with performance issues, measure and adjust accordingly.

Listing 4.22 shows the final refactoring of the definition of the check_business_partners method, in which the EXPORTING parameter is changed to a RETURNING parameter.

METHODS check_business_partners
  IMPORTING
  business_partners TYPE string
  RETURNING
  VALUE(result) TYPE check_result.

Listing 4.22 RETURNING Parameter Declaration

Calls to check_business_partners are much cleaner, as in the following code:

DATA(business_partners_check_result) =
  check_business_partners( my_business_partners ).

Consider Using result as the Name of RETURNING Parameters

The ability to name a RETURNING parameter is not a common capability in many mainstream programming languages. In those languages, a method simply returns the output with a special statement, for example, return 42.

Good method names obviate the need to name the RETURNING parameter. The RETURNING parameter name would do little more than parrot the method name or repeat something obvious, perhaps even create conflict with other members of the class.

Consumers also rarely refer to the RETURNING parameter name when invoking your method. They either assign the result of the method to a variable, use it as part of a larger expression, or ignore the result altogether.

Therefore, we recommend simply calling the RETURNING parameter result or something similar. What your methods are returning will be obvious and at the same time invisible, almost like a keyword of the language itself.

Use a meaningful name for the RETURNING parameter only if what it stands for is not obvious, for example, in methods that return me for method chaining or in methods that create an entity but don’t return the created entity, only its key.

4.2.7 CHANGING Parameters

You can use CHANGING parameters when a value is both an input and an output to a method. In this case, the method can change a component of the parameter as part of its operation. This approach only makes sense for structures and internal tables, which are values with internal components. A structure has fields that can be changed, while an internal table has rows that can be changed.

For example, let’s say you have many entities that need to be validated, and you want to collect all their validation messages in a single message table. You can use the example method shown in Listing 4.23 in your entity classes.

METHODS validate
  CHANGING
  messages TYPE message_table.

Listing 4.23 CHANGING Parameter Declaration

This instance method will validate the current entity and append any generated validation messages to the collecting parameter messages, which is an internal table of messages. The messages parameter could already contain messages (perhaps added earlier by other validation methods), and as a result, by using the same variable in a number of validate calls, all the generated messages will be collected in the parameter, as shown in Listing 4.24.

Use CHANGING Parameters Sparingly

CHANGING parameters should be reserved for cases where an existing structure or internal table that is potentially already filled with data is changed, like messages in the validate method example.

Use input parameters for object references that you work with, since your method can still call instance methods on the parameter to change its internal state.

If you fill up or replace the parameter completely (i.e., you assign a new value to the parameter), an EXPORTING parameter or a RETURNING parameter is more appropriate.
Methods

DATA messages TYPE message_table.
LOOP AT entities INTO DATA(entity).
  entity->validate(CHANGING messages = messages).
ENDLOOP.

Listing 4.24 CHANGING Parameter Usage

To eliminate the CHANGING parameter of the validate method, you can introduce a class that represents a message container, which will collect messages and also provide many more convenient methods for a list of messages. As an example, look at the message_container class in Listing 3.16, discussed in Chapter 3, Section 3.1.3.

The refactored validate method should look like Listing 4.25.

METHODS validate
  IMPORTING
    message_container TYPE ref to message_container.

Listing 4.25 Refactored CHANGING Parameter to IMPORTING Reference

This version is also easier to consume since the method now doesn’t have a CHANGING parameter, as in Listing 4.26.

DATA(message_container) = NEW message_container( ).
LOOP AT entities INTO DATA(entity).
  entity->validate(message_container).
ENDLOOP.

Listing 4.26 Using the Refactored Method

Avoid Using More Than One Kind of Output Parameter

Avoid combining IMPORTING, EXPORTING, and RETURNING parameters in the same method. Different kinds of output parameters indicate that the method might do more than one thing, which is confusing to readers and makes calling the method needlessly complicated.

4.2.8 Pass-by-Value and Pass-by-Reference

Apart from the input or output category of parameters, you can also declare parameters to be pass-by-value or pass-by-reference.

Pass-by-value is when a copy of the parameter value is made before calling the method, and the method parameter is assigned to that copy. Note that, when you pass an object reference by value, a copy of the reference is made, which is just a pointer that points to the same object in memory.

ABAP has important optimizations for value copy semantics. When a value is assigned to another variable or parameter, a copy is only made when needed, usually when changing something in the copy. This approach is commonly referred to as copy-on-write optimization.

Pass-by-reference is when a reference to the original value is passed to the method. If the method can reassign the value, the calling method will also be impacted by the change.

Pass-by-reference is the default for IMPORTING, EXPORTING, and CHANGING parameters. Pass-by-value is the only accepted option for RETURNING parameters. As a result, you’ll always see the VALUE modifier in RETURNING parameters. This modifier can also be used for other parameter categories to make them pass-by-value.

Although rarely used, the syntax for pass-by-reference can also be made explicit with the REFERENCE modifier. For example, the code shown earlier in Listing 4.10 can be rewritten into the code shown in Listing 4.27.

METHODS add_item
  IMPORTING
    REFERENCE(product) TYPE ref to product
    REFERENCE(quantity) TYPE quantity.

Listing 4.27 Explicit Pass-by-Reference Syntax

Do Not Reassign IMPORTING Parameters Passed by Value

Whenever an input parameter is passed by reference, which is the default, the parameter cannot be reassigned inside the method.

A pass-by-value IMPORTING parameter can be reassigned within the method. But, since this parameter is a copy, its new value will not be visible to the caller. This approach is a poor practice that you should avoid. IMPORTING parameters are meant to be inputs to a method, and reassigning them doesn’t make sense. Use proper local variables instead if you need to use the input as a starting value for a computation and then update the local variables instead. Use a pass-by-reference input parameter if you don’t need the parameter to be pass-by-value.

To make the quantity parameter shown in Listing 4.27 pass-by-value, you can end up with the declaration shown in Listing 4.28.

METHODS add_item
  IMPORTING
    REFERENCE(product) TYPE ref to product
    VALUE(quantity) TYPE quantity.

Listing 4.28 Pass-by-Value Input Parameter
Mind the Semantics of EXPORTING Parameters, Prefer Pass-By-Value

Parameters passed by reference ultimately point to existing memory areas that may have already been filled. EXPORTING parameters that are passed by reference will retain their original value before the method was called. This makes them work like CHANGING parameters (Section 4.2.7).

To make an EXPORTING parameter passed by reference behave like a true output parameter, it’s important to actually assign a value to it. It might make sense to clear the value at the beginning of the method if the parameter could end up unassigned, perhaps because of the method logic or an exception that is raised early on.

Another option is to make the EXPORTING parameter into a pass-by-value parameter, which will make it behave like a RETURNING parameter. EXPORTING parameters that are passed by value are handed over as new, separate memory areas that are empty. There’s no need to preemptively clear them, just like there’s no need to clear RETURNING parameters.

Therefore, prefer pass-by-value EXPORTING parameters.

Pass-by-reference EXPORTING parameters should be used when there’s a measurable performance hit when using pass-by-value. One such case is when you process large internal tables in batches. You can reuse an existing memory area repeatedly so as not to incur the cost of allocation for each batch. Cases like these should be rare. Aim for clarity and correctness first and improve the performance only when it becomes an issue.

As an example of EXPORTING parameters, look at the declaration of the try_parse_int method in Listing 4.29.

METHODS try_parse_int
  IMPORTING
    text TYPE string
  EXPORTING
    success TYPE abap_bool
    result TYPE int4.

Listing 4.29 Method with EXPORTING Parameters

To make sure that the pass-by-reference EXPORTING parameters to try_parse_int are initialized before the method returns, the method body starts by setting and clearing the parameters, as shown in Listing 4.30.

METHOD try_parse_int.

  success = abap_false.
  clear result.

Listing 4.30 Initializing EXPORTING Parameters

If you change the EXPORTING parameters to pass-by-value, as shown in Listing 4.31, the parameters will be initialized by default when entering the method, and there’ll be no need to clear or assign them to their initial value.

METHODS try_parse_int
  IMPORTING
    text TYPE string
  EXPORTING
    VALUE(success) TYPE abap_bool
    VALUE(result) TYPE int4.

Listing 4.31 Pass-by-Value EXPORTING Parameters

Another advantage of EXPORTING parameters that are pass-by-value is that they prevent a situation where the same variable could be assigned to an input and an output parameter by the caller at the same time, and the method could clear the IMPORTING parameter value indirectly by clearing the output EXPORTING parameter before doing its work.

CHANGING Parameters Passed by Value Don’t Make Sense

CHANGING parameters are both inputs and outputs to a method. Even though you can make a CHANGING parameter a pass-by-value parameter with the VALUE modifier, doing so does not change the way the CHANGING parameter works and generally does not make sense. Leave CHANGING parameters with their default pass-by-reference semantics.

4.3 Method Body

Code is more often read than it is written, and thus the code within a method should be optimized for reading. Not only will the method be easier to read, it will also be easier to consume, change, adapt, and test. You’ll find reasoning about the method easier and will spot defects more quickly. In this section, we’ll explore some important guidelines to ensure the code within your methods is clear, understandable, and flexible.

4.3.1 Do One Thing

To be easier to read, test, and maintain, a method should not only have a single responsibility, but it should perform that responsibility in the simplest way possible. A method should do one thing, and only one thing.
Many clean ABAP rules we discuss in this book help make your methods more focused and closer to the goal of doing only one thing. A method probably only does one thing if it follows the following rules:

- It has a small number of input parameters.
- It doesn’t include Boolean parameters.
- It has exactly one output parameter.
- It only throws one type of exception.
- It is small.
- It stays at a single level of abstraction.
- You cannot extract other meaningful methods.
- You cannot meaningfully group its statements into sections.

Let’s start by looking at the definition of a method for logging exceptions, as shown in Listing 4.32.

METHODS
   log_exception
      IMPORTING
         exception_instance TYPE REF TO cx_root.

Listing 4.32 Definition of a Method to Log Exceptions

The log_exception method will be used by some batch processing code whenever the batch processing program encounters an exception. The code will call log_exception to write information about the exception_instance to a log file and then abort the batch process. Log files only accept a maximum number of characters per line. A user can later read these logs and get more information about the exception to determine what went wrong with the processing.

Our first implementation of the log_exception method is shown in Listing 4.33.

METHOD log_exception.
   DATA(current_exception) = exception_instance.
   WHILE current_exception IS BOUND.
      " log a header with the exception class name
      DATA(class_name) =
         cl_abap_classdescr=>get_class_name( current_exception ).
      log_line( |
         ---- Exception occurred: { class_name } |
      ).
      " split the exception text into lines and log them
      DATA(exception_text) = current_exception->get_text( ).
      DATA(lines) = split_text_into_lines( exception_text ).
      LOOP AT lines into data(line).
         log_line( line ).
      ENDLOOP.
      " move to the previous exception
      current_exception = current_exception->previous.
      IF current_exception IS BOUND.
         log_line( `---- Exception has previous exception` ).
      ENDIF.
   ENDWHILE.
ENDMETHOD.

Listing 4.33 log_exception First Implementation

The supporting log_line method logs a single line of text to the log file, and the split_text_into_lines method splits a long text into multiple lines using a fixed number of columns for each line. This method returns a table of strings representing the lines.

Even though the log_exception method has a single responsibility, many steps are needed to perform its tasks, for instance, the following:

1. Store the exception in a local variable.
2. Start a loop while the exception is not null.
3. Get the class name of the exception.
4. Log a header including the class name.
5. Split the exception text into lines.
6. Loop and log each line.
7. Move to the previous exception.
8. Repeat the first loop.

These steps hint at many things that the method does: It loops to log the previous exception, queries the exception class name, and splits the exception text into lines. The next section will discuss a clean ABAP rule that improves the log_exception method until the method finally does only one thing in as few steps as possible.

4.3.2 Descend One Level of Abstraction

A method should tell a story that’s easy to follow and understand. Its body should stay at the same level of abstraction, which is just below the responsibility denoted by its name.

If more details of the story are needed, you can delve into the methods called by the first method until one of two things happen. Either you’ll reach basic statements of the language that implement some low-level algorithm, or you’ll reach a method call that belongs to an interface.
Either way, you should have a complete understanding of how a part of your code does its work and whether the code coordinates with other dependencies. For more information on class design and the crucial role of interfaces, check out Chapter 3.

That’s the beauty of well-crafted abstractions. The design is decoupled enough that you only need to understand the parts of the system that you currently require. You won’t need to read the whole codebase to understand what a certain part of the system does.

### Descend One Level of Abstraction

Statements in a method should be one level of abstraction below the method name itself. All these statements should be on the same level of abstraction.

This rule is called the “Stepdown Rule” in the book *Clean Code: A Handbook of Agile Software Craftsmanship* (Pearson, 2008) and called the “Single Level of Abstraction Principle (SLAP)” in the book *The Productive Programmer* (O’Reilly, 2008). The concept probably first appeared in the book *Smalltalk Best Practice Patterns* (Prentice Hall, 1996), where it was called the “Composed Method Pattern.”

How can you keep a method focused on a single level of abstraction? One approach is to explain what the method does in a few sentences or steps, which should all be related to its goal. These steps should not go into more detail than is needed to understand the responsibility of the method. The steps then should become the downstream methods the method calls or the statements it executes.

### Extract Method Refactoring

The extract method refactoring is a fundamental refactoring approach presented in the book *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999). You extract cohesive parts of a method into new methods, which are then called by the original method in the place from which the functionality was extracted.

This refactoring is supported by ABAP Development Tools (ADT). Select the section of a method you want to extract and press `Alt`+`Shift`+`M` on Windows or `Cmd`+`Alt`+`M` on macOS to start the extract method refactoring wizard.

The `log_exception` method shown previously in Listing 4.33 is divided into sections of statements with separate responsibilities, delimited by clarifying comments. The section comments in the method describe the following steps:

1. Log a header with the exception class name.
2. Split the exception text into lines and log them.
3. Move to the previous exception (and repeat).

Not all these sections are at the same level of abstraction. We want to know at a high-level what it means to log an exception. Details about getting the exception class name and splitting the exception text into lines is at a level of abstraction below the level we’re interested in. We’re talking about details we’re not ready to delve into just yet.

Taking a step back and thinking about what the method should accomplish, we might come up with the following steps, which are at the same level of abstraction:

1. Log a header for the exception.
2. Log a body for the exception.
3. Log the previous exception, if any.

This series of steps will translate into the final methods shown in Listing 4.34.

```
METHOD log_exception.
  log_exception_header( exception_instance ).
  log_exception_body( exception_instance ).
  log_previous_exception( exception_instance->previous ).
ENDMETHOD.

METHOD log_exception_header.
  DATA(class_name) = cl_abap_classdescr->get_class_name( exception_instance ).
  log_text( 
    "---- Exception occurred: { class_name }" ).
ENDMETHOD.

METHOD log_exception_body.
  DATA(exception_text) = exception_instance->get_text( ).
  log_text( exception_text ).
ENDMETHOD.

METHOD log_previous_exception.
  CHECK exception_instance IS BOUND.
  log_text( 
    "---- Exception has previous exception" ).
  log_exception( exception_instance ).
ENDMETHOD.
```

```
METHOD log_text.
  DATA(lines) = split_text_into_lines( text ).
  log_lines( lines ).
ENDMETHOD.

METHOD log_lines.
  LOOP AT lines INTO DATA(line).
  log_line( line ).
ENDLOOP.
```

Listing 4.34 Each Method at a Single Level of Abstraction
These methods are now each at the same level of abstraction and one level below the responsibility implied by their names.

Apart from making your methods simple and easy to understand, breaking down methods in this way might unearth interesting possibilities in the design. Whole new classes and subfeatures that were tangled up in the code may come to light and provide you with an enriched vocabulary for your project. Your design becomes less coupled, more cohesive, and much cleaner.

You can start thinking about logging the exception body, not just logging the exception text. This approach might mean, later on, that you’ll also include the exception stack trace with the text. You might start to think how to separate the formatting from the logging, and a whole new `exception_formatter` class or interface, with its own set of responsibilities, might emerge.

4.3.3 Keep Methods Small

ABAP is a notoriously verbose language. Coupled with a ton of legacy code, you may find gargantuan methods or function modules spanning thousands of lines of code. These sprawling methods are hard to understand, never mind debug and change. When you’re confronted with a data declaration section as big as the code shown in Listing 4.35, you already know this method does way more than one thing.

```
DATA:
  class TYPE vseoclass,
  attributes TYPE seoo_attributes_r,
  methods TYPE seoo_methods_r,
  events TYPE seoo_events_r,
  types TYPE seoo_types_r,
  aliases TYPE seoo_aliases_r,
  implementings TYPE seor_implementings_r,
  inheritance TYPE vseoextend,
  friendships TYPE seof_friendships_r,
  typepuses TYPE seot_typepusages_r,
  clsdeferdds TYPE vseocdefer,
  intdeferdds TYPE vseoidfer,
  new_clskey_save TYPE seoclskey.
```

Listing 4.35 Too Many Data Declarations: A Prelude to a Huge Method

Look at the original `log_exception` method shown earlier in Listing 4.33. You might have thought this method was reasonably sized. However, as shown earlier in Listing 4.34, still many cohesive parts could be made into simpler methods. The resulting methods were much smaller than the original and much cleaner. The resulting `log_exception` method reads like the steps it implements. You can read it and understand what it does at a high-level quickly without even delving into the other methods.

Now, look at the `log_exception_body` method, shown in Listing 4.36.

```
METHOD log_exception_body.
  DATA(exception_text) = exception_instance->get_text( ).
  log_text( exception_text ).
ENDMETHOD.
```

Listing 4.36 The `log_exception_body` Method

You might think you can still extract another method from it, resulting in the code shown in Listing 4.37.

```
METHOD log_exception_body.
  log_exception_text{ exception_instance }.
ENDMETHOD.
```

Listing 4.37 The Extracted `log_exception_text` Method Is Redundant
A new `log_exception_text` method could have a place, perhaps if `log_exception_body` did more than just log the exception text. At this point, however, the new method is simply a reiteration of the old one and doesn’t add much semantic value over the existing method.

Stop extracting methods when you feel comfortable that the method expresses its intent clearly and simply through its statements.

### 4.3.4 Fail Fast

Often, when an error occurs or an exception is thrown, the root cause of the problem can be traced to some parameter having an invalid value introduced earlier in the program flow. The closer that exception is to the place the error was introduced, the easier the error is to find and fix.

**Fail Fast**

Document and check the input parameters to your methods and fail with an exception as early as possible if they are not valid for the execution of the method.

This recommendation is especially important for public and protected methods. Private methods could also benefit from validation (or using `ASSERT`) should the program and the assumptions that they make change, but usually you can rely on the validation already performed by the methods that call them.

Validating later is not only hard to spot and understand but might waste important resources. Not validating might make your code fail in hard-to-predict ways and might result in invalid data and undesirable behaviors.

The `step_fan` class, shown earlier in Listing 4.8, is not validating its `step` input parameter to the constructor. This value is used to step over a fixed amount when changing the fan intensity. A negative value or zero wouldn’t work, and a value that’s bigger than the maximum intensity wouldn’t make sense. The code should validate this value and fail at the earliest possible spot, which is on construction, as shown in Listing 4.38.

```csharp
METHOD constructor.
    super->constructor( ).
    IF step < 1 OR step > max_intensity.
        RAISE EXCEPTION NEW invalid_argument_exception(
            argument_name = `step`,
            argument_value = step ).
    ENDIF.
    me->step = step.
ENDMETHOD.
```

**Listing 4.38 Constructor Validation**

The `step_fan` constructor first calls the superclass constructor and then validates its input parameter and raises an exception right away if it’s invalid. It’s important to add as much information to the exception as possible. In this case, the custom `invalid_argument_exception` includes the argument name and the invalid argument value.

**Focus on the Happy Path or Error Handling**

A method body should emphasize the happy path it’s built for. If it needs to perform error handling, it’s important that this error handling code doesn’t divert attention from the main intent of the method. If it does, put the error handling code in a dedicated method. For more information on this topic, refer to Chapter 11.

As it stands, the `step_fan` constructor error handling code dominates its logic. It makes sense to break it down so that it’s more readable, as shown in Listing 4.39.

```csharp
METHOD constructor.
    super->constructor( ).
    validate_step( step ).
    me->step = step.
ENDMETHOD.

METHOD validate_step.
    IF step < 1 OR step > max_intensity.
        RAISE EXCEPTION NEW invalid_argument_exception(
            argument_name = `step`,
            argument_value = step ).
    ENDIF.
ENDMETHOD.
```

**Listing 4.39 Separate Error Handling Method**

Note that this kind of validation is meant to aid other developers fix their calling code, and not for end users, since this validation is very technical in nature. End users could get business-relevant validation errors in the form of messages that are presented to them. Earlier in Section 4.2.7, Listing 4.26, we presented the code to validate a list of entities. If this is part of a larger method that does some processing with the entities, failing early might mean failing if any of those validations generated an error message. Adding in some refactoring, we might end up with the code shown in Listing 4.40.

```csharp
DATA(message_container) = NEW message_container( ).
    validate_entities( message_container ).
    message_container->raise_if_in_error( ).
...
```

**Listing 4.40 Raising When a Message Container Has Error Messages**
In this case, the message_container class has a method that raises an exception if it contains any error messages. Validation messages are added to the exception, which could be caught by an upper layer and then shown to end users in a suitable manner. For example, they could be OData error messages that are then displayed in a message pop-over in an SAPUI5 application.

### 4.3.5 CHECK versus RETURN

Another type of validation is when you simply return early if the arguments to a method don’t meet its expectations. In these cases, it’s fine for the method to simply return instead of raising an exception.

Earlier in Listing 4.34, you saw an example of this “return early” behavior for the method log_previous_exception. If the exception_instance parameter is not bound (which means that the value is null), the method will return early without completing its work.

Another method that would benefit from this kind of validation is the main log_exception method itself. If the exception instance is not bound, the method simply returns without doing any work, as shown in Listing 4.41.

```abap
METHOD log_exception.
  CHECK exception_instance IS BOUND.
  log_exception_header( exception_instance ).
  log_exception_body( exception_instance ).
  log_previous_exception( exception_instance->previous ).
ENDMETHOD.
Listing 4.41 Check If the Exception Instance Is Bound
```

Both methods now use the CHECK statement. This statement reads perfectly fine but is not a common statement and might cause some confusion. What happens when the check condition evaluates to false might not be clear.

Another way to implement this kind of validation is to use a condition check followed by a RETURN statement, as shown in Listing 4.42.

```abap
METHOD log_exception.
  IF exception_instance IS NOT BOUND.
    RETURN.
  ENDIF.
  log_exception_header( exception_instance ).
  log_exception_body( exception_instance ).
  log_previous_exception( exception_instance->previous ).
ENDMETHOD.
Listing 4.42 Return If the Exception Instance Is Not Bound
```

No matter what you choose, you should not use CHECK outside of the initialization section of a method. The statement behaves differently in different positions and may lead to unclear, unexpected effects.

For example, a CHECK statement inside a LOOP ends the current iteration of the loop and proceeds to the next iteration. In this context, a reader might mistakenly expect the CHECK statement to end the method or exit the loop. In this case, we recommend using an IF statement in combination with CONTINUE instead, since CONTINUE can only be used inside loops.

### 4.4 Calling Methods

At this point, we’ve explored many aspects of calling a method in ABAP, from calling static methods, where you’ll specify the class name using a fat arrow (=>), to calling instance methods, where you need a reference to an instance of the class and use the thin arrow syntax (->).

Interface methods can be called the same way as class methods if you have a reference pointing to the interface type. If, however, you have a reference to a class and want to call an interface method on it, you need to add the interface name and a tilde (~) to the start of the method call. For example, if you have an instance of a thermal_switch (shown earlier in Listing 4.7) in the my_thermal_switch variable, you can call its interface method switchable~is_on in the following way:

```abap
DATA(is_on) = my_thermal_switch->switchable~is_on( ).
```

The class can also alias the interface method to make the call look like a normal class instance method call. An example of an aliased interface method and its usage can be found in Chapter 3, Section 3.1.2.

With regard to how parameters are passed to methods, many options are available, depending on how the method is declared. In this section, we’ll explore some guidelines to make your method invocations clearer and more readable.

#### 4.4.1 Passing Input Parameters

Listing 4.43 shows a possible declaration of the method split_text_into_lines, which we used earlier in Listing 4.33.

This step adds another indentation level, which might break the visual flow of the code. However, how you or your team feel about using CHECK statements is up to you. Community discussion might suggest that the statement is so unclear that few people will understand its intended behavior.

This method is critical for ensuring the correct setup of the thermal switch in the example. It sets the switchable variable to a new value and uses the logical operator ‘&’ to assign the result of the logical expression. This approach is essential for performing logical comparisons and assignments in ABAP.

In summary, the example demonstrates the importance of using meaningful and consistent naming conventions for variables and methods. Precise naming allows for better code readability and maintainability, ensuring that the code is easier to understand and work with over time.
4.4 Calling Methods

METHODS:
  split_text_into_lines
    IMPORTING
text TYPE string
  columns_per_line TYPE int4 DEFAULT 40
RETURNING
  VALUE(lines) TYPE string_table.

Listing 4.43 split_text_into_lines Declaration

When only providing input parameters to a method and assigning its RETURNING parameter to a variable, you can simply call the method, as shown in Listing 4.44.

DATA(lines) =
  split_text_into_lines(
    text = text
    columns_per_line = 80 ).

Listing 4.44 Calling a Method with Input Parameters and a RETURNING Parameter

You can also explicitly annotate the parameter category before the parameter list. In the case of IMPORTING parameters, the caller must use the counterpart keyword EXPORTING (Section 4.2.5), as shown in Listing 4.45.

DATA(lines) =
  split_text_into_lines( 
    EXPORTING
    text = text
    columns_per_line = 80 ).

Listing 4.45 Explicitly Using EXPORTING

Omit the Optional EXPORTING Keyword
When all you have are input parameters and possibly a returning parameter, the EXPORTING keyword is optional in a method call.

However, this style is unnecessarily verbose and redundant. Omit the optional EXPORTING keyword in these cases.

4.4.2 Capturing Output Parameters
RETURNING parameters can be simply assigned as a value coming out of the method call itself, as we’ve seen in Listing 4.44.

A RETURNING parameter from a method can also be captured alongside the parameter list by using the RECEIVING keyword on a method call, as shown in Listing 4.46.

Listing 4.46 Using the RECEIVING Keyword

Omit the RECEIVING Keyword
The call style using the RECEIVING keyword is unnecessarily verbose and forces you to specify the EXPORTING keyword for input parameters.

To make the code more readable, omit the RECEIVING keyword and capture the return value of the method directly.

The EXPORTING parameters from a method are captured by using the IMPORTING keyword in a call. You’ve already seen an example of a call like this earlier in Listing 4.19. Another example is shown in Listing 4.47, which calls the try_parse_int method declared earlier in Listing 4.29.

try_parse_int(
  EXPORTING
  text = '42'
  IMPORTING
    success = DATA(parse_succeeded)
    result = DATA(parsed_int) ).

Listing 4.47 Capturing EXPORTING Parameters with the IMPORTING Keyword

Any time an EXPORTING or CHANGING parameter is used, you must declare the parameter category with the proper keyword in the method call. As described earlier in Section 4.2, EXPORTING parameters are called with the IMPORTING keyword, and CHANGING parameters are called with the CHANGING keyword, as shown earlier in Listing 4.24. This rule also forces you to explicitly define the keyword EXPORTING for IMPORTING parameters.

4.4.3 The CALL METHOD Construct
Another way to call a method in ABAP is to use a style reminiscent of the way function modules are called, with the CALL METHOD construct, as shown in Listing 4.48.

This call style doesn’t use parentheses around the parameters. With this approach, you are forced to declare all keywords for capturing parameters, including the RECEIVING keyword for a RETURNING parameter.
CALL METHOD split_text_into_lines
EXPORTING
text = text
RECEIVING
result = DATA(lines).

Listing 4.48 Using CALL METHOD

Reserve CALL METHOD for Dynamic Method Calls
The CALL METHOD construct for calling methods is obsolete and unnecessarily verbose. Use the call style with parentheses to make the code more consistent and easier to read and to better support other clean code patterns discussed in this chapter. Only use the CALL METHOD construct for dynamic method calls, where the class or the method name is resolved at runtime, as in Listing 4.49.

CALL METHOD modify->(method_name)
EXPORTING
node = my_bo_c=>node-item
key = item->key
data = item
changed_fields = changed_fields.

Listing 4.49 CALL METHOD for Dynamic Method Calls

However, we recommend you design your application around interfaces and avoid using dynamic method calls whenever possible.

4.4.4 Optional Parameter Name
When a method has a single non-optional input parameter or a PREFERRED PARAMETER, the caller can provide a single unnamed argument for that parameter. This call style, when using a properly named variable or constant, can make the calling code more concise and still clean and readable, as in the following code:

DATA(lines) = split_text_into_lines( text ).

Naming the parameter in this case, as in the following code, is unnecessarily verbose and redundant:

DATA(lines) = split_text_into_lines( text = text ).

At times, however, a method name alone fails to help the reader understand what the supplied argument means. In this case, adding the parameter name increases readability, as in the following code:

car->drive( speed = 50 ).

4.4.5 Self-Reference
In the code inside instance methods, the self-reference me is implicitly set by the system and can be used to access the current class instance. This self-reference is equivalent to the self-reference this in languages like Java and C#.

Picking up on the version of the log_exception method shown earlier in Listing 4.41, we could implement the same code by explicitly using the self-reference me, as shown in Listing 4.50.

METHOD log_exception.
CHECK exception_instance IS BOUND.
me->log_exception_header( exception_instance ).
me->log_exception_body( exception_instance ).
me->log_previous_exception( exception_instance->previous ).
ENDMETHOD.

Listing 4.50 Using the Self-Reference me to Call Instance Methods

Omit the Self-Reference When Not Needed
You should omit the self-reference me when calling instance methods or accessing instance members of the containing class. Using me is redundant and doesn’t add any clarity to the code.

However, if a parameter or a variable name shadows a class instance member, then you’re forced to use me to disambiguate, for example, in the step_fan constructor shown in Listing 4.51.

METHOD constructor.
super->constructor( ).
validate_step( step ).
me->step = step.
ENDMETHOD.

Listing 4.51 Using me to Assign to an Instance Member of the Class

Only use me when you need to reference a shadowed class instance member. Otherwise, the self-reference is redundant and only adds clutter to your code.
4.5 Summary

In this chapter, we covered some important guidelines on how to think about, design, and use methods in clean ABAP code. Let’s recap the clean ABAP rules for methods we covered.

We started with some guidance on object-oriented design, including the following recommendations:

- Consider using instance methods by default.
- Don’t call static methods through instance variables.
- Public instance methods should be part of an interface.
- Be careful when redefining methods.

Then, we took you on a grand tour of how parameters work and how to design them, while exploring the following recommendations:

- Aim for as few input parameters as possible.
- Split the behavior instead of adding optional parameters.
- Use a preferred parameter sparingly.
- Create separate methods instead of using Boolean parameters.
- Minimize the number of EXPORTING parameters.
- Prefer RETURNING to EXPORTING.
- Consider using result as the name of RETURNING parameters.
- Use CHANGING parameters sparingly.
- Avoid using more than one kind of output parameter.
- Do not reassign IMPORTING parameters passed by value.
- Mind the semantics of EXPORTING parameters, prefer pass-by-value.
- Pass-by-value CHANGING parameters don’t make sense.

In Section 4.3, we delved into some important considerations when designing the body within methods:

- A method should do one thing, and only one thing.
- Descend one level of abstraction.
- Keep methods small.
- Fail fast.
- Focus on the happy path or error handling.

Finally, we ended this chapter with a discussion of several clean ABAP aphorisms related to invoking methods:

- Omit the optional EXPORTING keyword.
# Contents

Preface ........................................................................................................................ 17

1 Introduction ............................................................................................................... 23

1.1 What Is Clean ABAP? .......................................................................................... 23
  1.1.1 What Is Readability? ...................................................................................... 23
  1.1.2 What’s the Story behind Clean ABAP? ......................................................... 25

1.2 How to Get Started with Clean ABAP ................................................................. 26

1.3 How to Handle Legacy Code ................................................................................. 26

1.4 How to Check Code Automatically ....................................................................... 28

1.5 How Does Clean ABAP Relate to Other Guides? ................................................ 29

1.6 How to Engage with the Clean ABAP Community .............................................. 30

1.7 Summary ................................................................................................................ 31

2 The ABAP Language ................................................................................................. 33

2.1 Mind Legacy Code ............................................................................................... 33

2.2 Mind Performance ............................................................................................... 35
  2.2.1 Mostly Harmless ........................................................................................... 35
  2.2.2 Start Clean, Degrade Only Where Needed .................................................... 36
  2.2.3 Measure, Don’t Assume ................................................................................ 37

2.3 Prefer Object Orientation to Procedural Programming ....................................... 37
  2.3.1 ABAP’s Programming Paradigm .................................................................. 37
  2.3.2 The Difference between Function Groups and Classes .............................. 38
  2.3.3 Differences in ABAP’s Object Orientation ................................................ 41
  2.3.4 If You Have No Choice ............................................................................... 43

2.4 Favor Functional Language Constructs over Procedural Language Constructs .... 43

2.5 Avoid Obsolete Language Elements .................................................................... 46

2.6 Use Design Patterns Wisely ................................................................................ 48
  2.6.1 Too Many Singletons .................................................................................... 48
  2.6.2 Too Many Patterns Mixed .......................................................................... 49

2.7 Summary ................................................................................................................ 50
3 Classes and Interfaces

3.1 Object Orientation .......................................................... 51
3.1.1 Interfaces .............................................................. 52
3.1.2 Classes and Objects .................................................. 57
3.1.3 State ........................................................................ 75
3.2 Scope and Visibility .......................................................... 76
3.2.1 Global and Local Scope ............................................ 76
3.2.2 Visibility ................................................................. 81
3.3 Constructors .................................................................. 83
3.3.1 Scope and Visibility .................................................. 83
3.3.2 Dependency Injection .............................................. 85
3.3.3 Static Creation Methods ........................................... 86
3.3.4 Creational Patterns .................................................. 89
3.3.5 Instantiation .............................................................. 91
3.4 Summary ...................................................................... 93

4 Methods

4.1 Object-Oriented Programming .......................................... 95
4.1.1 Static and Instance Methods ...................................... 95
4.1.2 Public Instance Methods .......................................... 98
4.1.3 Method Redefinition ................................................. 100
4.2 Parameters .................................................................... 102
4.2.1 How Many Input Parameters Are Too Many? ............... 103
4.2.2 Optional Input Parameters ....................................... 103
4.2.3 Preferred Input Parameters ...................................... 105
4.2.4 Boolean Input Parameters ........................................ 106
4.2.5 EXPORTING Parameters ....................................... 108
4.2.6 RETURNING Parameters ....................................... 109
4.2.7 CHANGING Parameters ......................................... 111
4.2.8 Pass-by-Value and Pass-by-Reference ......................... 112
4.3 Method Body .................................................................. 115
4.3.1 Do One Thing ......................................................... 115
4.3.2 Descend One Level of Abstraction ......................... 117
4.3.3 Keep Methods Small .............................................. 120
4.3.4 Fail Fast ................................................................. 122
4.3.5 CHECK versus RETURN ....................................... 124

4.4 Calling Methods ............................................................... 125
4.4.1 Passing Input Parameters ........................................ 125
4.4.2 Capturing Output Parameters .................................. 126
4.4.3 The CALL METHOD Construct ............................... 127
4.4.4 Optional Parameter Name ....................................... 128
4.4.5 Self-Reference ......................................................... 129

4.5 Summary ...................................................................... 130

5 Names

5.1 Good Naming ................................................................ 133
5.1.1 Descriptive Names ................................................... 133
5.1.2 Domain Terms ........................................................ 134
5.1.3 Plural or Singular? .................................................... 135
5.1.4 Abbreviations .......................................................... 136
5.1.5 Naming Classes and Methods ................................. 136
5.1.6 Noise Words ............................................................. 137
5.1.7 Term Consistency ...................................................... 137
5.1.8 Design Patterns in Code ......................................... 138
5.1.9 Hungarian Notation and Prefixes ......................... 138

5.2 ABAP Peculiarities .......................................................... 138
5.2.1 Group of Objects with Possible Name Collisions .......... 139
5.2.2 snake_case versus camelCase ............................... 139

5.3 Affixes: Prefixes, Suffixes, and Infixes ......................... 140
5.3.1 The Musts ................................................................. 140
5.3.2 Helpful Affixes ........................................................ 141
5.3.3 Affixes That Don’t Bring Value ............................... 142

5.4 Dealing with Legacy Code .............................................. 142
5.5 Summary ...................................................................... 143

6 Variables and Literals .......................................................... 145

6.1 Variables ...................................................................... 145
6.1.1 Declaring Variables ................................................ 146
6.1.2 Branches and Scope .............................................. 148
6.1.3 Declaration Chaining ............................................. 149
6.1.4 Looping Variables .................................................. 151
## 10 Formatting

10.1 Consistency in Coding Style ........................................ 216
10.2 Optimizing for Reading .............................................. 216
10.3 The Pretty Printer ..................................................... 217
10.4 Number of Statements Per Line ................................... 218
10.5 Line Length ............................................................. 219
10.6 Condensing the Code ................................................ 220
10.7 Blank Lines ............................................................ 220
10.8 Alignment of Assignment Statements ....................... 221
10.9 Alignment of Variable Declarations .......................... 222
10.10 Placement of Closing Brackets .................................. 223
10.11 Formatting Method Parameters ................................. 223
  10.11.1 Single Parameter Calls ........................................ 224
  10.11.2 Line Break Multiple Parameters ......................... 224
  10.11.3 Placement of Parameters .................................... 224
  10.11.4 Indenting Parameters ........................................ 225
  10.11.5 Vertical Alignment of Parameter Values ............... 226
  10.11.6 Breaking Calls to a New Line ............................ 226
  10.11.7 Indents and Using the Tab Key ............................ 227
  10.11.8 Indentation of Inline Declarations ...................... 227
10.12 Summary ............................................................. 227

## 11 Error Handling

11.1 Messages ............................................................... 229
11.2 Return Codes ........................................................ 232
  11.2.1 Exceptions versus Return Codes ....................... 233
  11.2.2 Handling Failures ............................................. 234
11.3 Exceptions ........................................................... 236
  11.3.1 Exceptions Are for Errors ................................. 236
  11.3.2 Class-Based Exceptions .................................... 237
  11.3.3 CX_STATIC_CHECK ........................................ 242
  11.3.4 CX_NO_CHECK ............................................. 243
  11.3.5 CX_DYNAMIC_CHECK ..................................... 244

## 12 Unit Testing

12.1 Test Classes .......................................................... 254
  12.1.1 Test Class Properties ....................................... 254
  12.1.2 Test Class Scope ............................................ 257
  12.1.3 Test Helper Classes ........................................ 259
  12.1.4 Executing Tests .............................................. 260
12.2 Test Methods ........................................................ 262
  12.2.1 Test Fixture Methods ....................................... 262
  12.2.2 Given-When-Then Style .................................. 265
12.3 Class under Test .................................................... 266
12.4 Naming ............................................................... 267
12.5 Assertions ........................................................... 269
  12.5.1 Writing Effective Assertions ............................ 270
  12.5.2 Checking Expected Exceptions ....................... 272
  12.5.3 Unexpected Static Exceptions ......................... 274
  12.5.4 Custom Assertions ........................................ 275
  12.5.5 Constraints .................................................. 277
12.6 Test Doubles ........................................................ 279
  12.6.1 Dependency Inversion and Test Doubles ............. 279
  12.6.2 ABAP Object-Oriented Test Double Framework .... 284
  12.6.3 More ABAP Test Tools ................................... 288
12.7 Test Seams ........................................................... 289
12.8 Principles ............................................................ 291
  12.8.1 Test-Driven Development ............................... 291
  12.8.2 Clean Test Properties .................................... 292
  12.8.3 Test Coverage .............................................. 294
12.9 Summary ............................................................. 294
13 Packages

13.1 General Package Concepts
13.1.1 Use Cases
13.1.2 Reuse Levels
13.1.3 Cohesion

13.2 Package Concepts in ABAP
13.2.1 Package Types
13.2.2 Encapsulated Packages
13.2.3 Package Interfaces
13.2.4 Best Practices

13.3 Package Design Options
13.3.1 Side-by-Side Application Hierarchies
13.3.2 Layer-Based Hierarchies
13.3.3 Translation Relevance Split Hierarchies

13.4 Package Checks
13.4.1 What Are Package Checks?
13.4.2 Manual Execution of Package Checks
13.4.3 Automated Execution of Package Checks
13.4.4 How to Fix Package Check Errors
13.4.5 Best Practices

13.5 Consequences of Poor or No Package Strategy

13.6 Summary

14 How to Implement Clean ABAP

14.1 Common Understanding among Team Members
14.2 Collective Code Ownership
14.3 Clean Code Developer Initiative

14.4 Tackling the Broken Window Effect
14.4.1 Static Code Check
14.4.2 Metrics
14.4.3 Code Coverage

14.5 Code Review and Learning
14.5.1 Code Review Prefix
14.5.2 Style Guide

14.6 Clean Code Advisor

14.7 Learning Techniques
14.7.1 Kata
14.7.2 Dojo
14.7.3 Code Retreat
14.7.4 Fellowship
14.7.5 Pair Programming
14.7.6 Mob Programming
14.7.7 Habits

14.8 Continuous Learning in Cross-Functional Teams
14.8.1 Profile of a Team Member
14.8.2 Cross-Functional Teams
14.8.3 Multipliers: Need for Topic Experts in the Team
14.8.4 Need for Community of Practice

14.9 Summary

The Authors

Index

Index
Index

A
ABAP ......................................................... 23, 33
  comments ............................................... 212
  exceptions ........................................... 233, 236
  messages .............................................. 229
  object orientation .................................. 41
  packages ........................................... 299
  programming paradigm .......................... 37
  test tools ............................................ 288
ABAP Authority Check Test Helper API .......... 289
ABAP CDS Test Double Framework ................. 288
ABAP Data Dictionary ............................... 51, 53, 64
ABAP Development Tools (ADT) ................. 77, 118, 138, 217, 219, 239, 246, 254, 260, 310
ABAP Docs ................................................ 212, 258, 268
ABAP Objects .......................................... 51–52, 95
  implementation inheritance ...................... 65
ABAP Programming Guidelines .................... 30
ABAP Runtime Type Services (RTTS) ............. 86
ABAP SQL Test Double Framework .................. 288
ABAP Test Cockpit ................................... 28, 243, 325
ABAP Test Double Framework ...................... 284
  configure calls .................................... 287
  when to use ...................................... 288
ABAP Unit ................................................ 253, 256, 262
  call sequence ..................................... 262
ABAP Workbench ...................................... 308
  abap_bool type .................................... 162
  abap_false ........................................ 162–163
  abap_true .......................................... 162
  abap_undefined .................................. 162
  abapGet ........................................... 29
  abapinit ............................................ 29
ABAP-Managed Database Procedures (AMDPs) 35
  abapOpenChecks ................................ 29
Abbreviations ......................................... 136
Abstract classes ...................................... 59, 63
  rules ............................................... 61
  Abstract factory .................................. 91
  Abstract final ..................................... 155
  Abstract types ................................... 53
  Abstraction ....................................... 24, 52, 55, 81, 270
  levels ................................................ 117
  Accessor methods ................................ 82
  Accidental architecture .......................... 82
  Accounts .......................................... 53
  Actions ............................................. 57
  Aliases ............................................ 140–141
  Alignment .......................................... 221
  assignment statements ......................... 221
  Arguments ......................................... 224
  Array-like tables .................................. 172
  ASSERT statements .............................. 251
  Assertions ......................................... 255, 269
  expected exceptions ............................ 275
  failure ............................................. 269, 271, 274
  methods .......................................... 269, 277
  writing ............................................ 270
  Assignment operators ............................ 221
  Automated executions ............................ 312
  Automated testing ................................ 253, 292, 353
  Avoidable errors .................................. 247

B
Backlog items .......................................... 27
Base classes ........................................... 59
Black grade ............................................ 322
Blank lines .......................................... 220
Blank spaces ........................................ 220
Blank operations .................................... 182
Blank processing .................................... 181
Booleans .............................................. 161, 163
  conditions ....................................... 191
  naming ............................................ 137
  parameters ....................................... 106
Bottlenecks ............................................ 320
Boy Scout Rule ...................................... 27, 201, 212
Branch coverage .................................... 294, 326
Branches ............................................. 148
Branches ............................................. 148
  IF statements ................................... 188
  nesting depth .................................... 190
  Broken window effect ........................... 323
  Examples .......................................... 324


341
<table>
<thead>
<tr>
<th>C</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL FUNCTION statement</td>
<td>231</td>
</tr>
<tr>
<td>Call method</td>
<td>127, 189</td>
</tr>
<tr>
<td>case</td>
<td>140</td>
</tr>
<tr>
<td>CASE blocks</td>
<td>198</td>
</tr>
<tr>
<td>CASE keyword</td>
<td>185</td>
</tr>
<tr>
<td>CASE statements</td>
<td>196</td>
</tr>
<tr>
<td>hide</td>
<td>199</td>
</tr>
<tr>
<td>reusing</td>
<td>199</td>
</tr>
<tr>
<td>versus ifs</td>
<td>197</td>
</tr>
<tr>
<td>CATCH blocks</td>
<td>202, 249</td>
</tr>
<tr>
<td>CATCH keyword</td>
<td>249</td>
</tr>
<tr>
<td>Caching exceptions</td>
<td>248</td>
</tr>
<tr>
<td>cleanup</td>
<td>250</td>
</tr>
<tr>
<td>Chaining</td>
<td>150, 166</td>
</tr>
<tr>
<td>ifs</td>
<td>200</td>
</tr>
<tr>
<td>Changing parameters</td>
<td>111</td>
</tr>
<tr>
<td>pass by value</td>
<td>115</td>
</tr>
<tr>
<td>Character limits</td>
<td>219</td>
</tr>
<tr>
<td>CHECK keyword</td>
<td>186</td>
</tr>
<tr>
<td>CHECK statements</td>
<td>124, 186, 200, 203</td>
</tr>
<tr>
<td>checks</td>
<td>148, 164-165, 189, 200</td>
</tr>
<tr>
<td>automatic</td>
<td>28</td>
</tr>
<tr>
<td>basic</td>
<td>165</td>
</tr>
<tr>
<td>packages</td>
<td>308</td>
</tr>
<tr>
<td>result</td>
<td>109</td>
</tr>
<tr>
<td>suppress</td>
<td>213</td>
</tr>
<tr>
<td>Child classes</td>
<td>66, 69</td>
</tr>
<tr>
<td>Circular dependency</td>
<td>309</td>
</tr>
<tr>
<td>CL_OSQ_REPLACE statement</td>
<td>288</td>
</tr>
<tr>
<td>Class declaration statement</td>
<td>83</td>
</tr>
<tr>
<td>Class instances</td>
<td>59</td>
</tr>
<tr>
<td>Class interfaces</td>
<td>266</td>
</tr>
<tr>
<td>Class member visibility</td>
<td>81</td>
</tr>
<tr>
<td>Class under test</td>
<td>109</td>
</tr>
<tr>
<td>naming</td>
<td>268</td>
</tr>
<tr>
<td>Class-based exceptions</td>
<td>233, 235-237, 242, 245</td>
</tr>
<tr>
<td>raising</td>
<td>247</td>
</tr>
<tr>
<td>Classes</td>
<td>51, 57</td>
</tr>
<tr>
<td>abstract</td>
<td>59</td>
</tr>
<tr>
<td>builder</td>
<td>89</td>
</tr>
<tr>
<td>composition</td>
<td>70</td>
</tr>
<tr>
<td>constraints</td>
<td>154</td>
</tr>
<tr>
<td>constraint</td>
<td>278</td>
</tr>
<tr>
<td>definition</td>
<td>59</td>
</tr>
</tbody>
</table>

**Classes (Cont.)**
- dependencies                          | 85   |
- enumeration                           | 154  |
- friendship                            | 81   |
- global                                 | 76   |
- immutable                              | 82   |
- inheritance                           | 39, 59, 65 |
- instance                               | 96   |
- local                                  | 77   |
- messages                               | 230  |
- naming                                 | 136  |
- production                             | 254  |
- small                                   | 36   |
- stateful                                | 75   |
- static                                  | 61   |
- stub                                   | 281  |
- test                                    | 254  |
- test helper                             | 259  |
- utility                                 | 61, 97 |
- visibility                              | 81   |
- wrap with functions                     | 43   |
- classic exceptions                      | 234-235 |
- convert                                | 241  |
- Clean ABAP                             | 23, 29, 33, 35-37 |
- back story                              | 25   |
- community engagement                   | 50   |
- get started                             | 26   |
- repository                             | 31   |
- rules                                   | 26   |
- Clean ABAP open-source style guide     | 327  |
- Clean code                              | 23, 185, 205, 249, 319 |
- messages                               | 232  |
- Clean code advisor                      | 330  |
- Clean Code Developer Initiative         | 322  |
- Clean islands                           | 28   |
- Clean test properties                   | 292  |
- Cleanup                                 | 250  |
- CLEANUP keyword                        | 81, 91, 107 |
- Client code                            | 300  |
- client packages                        | 223  |
- Closing brackets                        | 325-326 |
- Code base                               | 219  |
- Code editor                             | 28, 37 |
- Code Inspector                          | 115  |
- Code rewrite                            | 25, 326 |
- feedback culture                        | 327  |
- managers                               | 329  |
- prefix                                 | 326  |
- rules                                   | 328  |
- visibility                              | 327  |

**Code rewrite**                        | 324  |
**Code steward**                         | 321  |
**Code under test**                      | 266  |
**Isolate**                              | 279  |
**Global**                               | 289, 299 |
**Collecting parameters**                | 111  |
**Collective code ownership**            | 320  |
**Steps**                                | 320  |
**Command**                              | 56   |
**Commented code**                       | 211  |
**Comments**                             | 205  |
**IF branches**                          | 210  |
**Instructive**                          | 211  |
**Issues**                               | 205  |
**Method names**                         | 268  |
**Placement**                            | 208  |
**Pseudo**                               | 213  |
**Remove**                               | 210  |
**Special**                              | 212  |
**What to avoid**                        | 208  |
**When to use**                          | 207  |
**Common understanding**                 | 319, 321 |
**Communication style**                  | 328  |
**Communities of practice**              | 336  |
**Community engagement**                 | 30   |
**Compatible reference variable**        | 91   |
**Complex conditions**                   | 195  |
**Composing object**                     | 74   |
**Composition**                          | 70   |
**Constraints**                          | 279  |
**Versus inheritance**                   | 73   |
**CONDO statements**                     | 187  |
**Condensing**                           | 220  |
**Conditions**                           | 186, 190, 195 |
**Complex**                              | 195  |
**Decompose**                            | 192  |
**Enhance**                              | 192  |
**Inline**                               | 195  |
**Positive formulation**                 | 190  |
**Raising exceptions**                   | 248  |
**Reusing**                              | 193  |
**Consistency**                           | 137, 216 |
**Constant pattern**                     | 155  |
**Constants**                            | 152, 159 |
**Grouping**                             | 159  |
**Handling**                             | 154  |
**Constraints**                          | 277  |
**Classes**                              | 278  |
**Constructors**                         | 64, 68, 83, 87 |
**ADT**                                  | 239  |
**Declare multiple**                     | 86   |
**Default**                              | 84   |

**Constructors (Cont.)**                 | 86   |
**Error handling**                       | 123  |
**Injection**                            | 280  |
**Scope and visibility**                 | 83   |
**Specialized**                          | 97   |
**Static**                               | 49, 90 |
**CONTINUE statement**                   | 201  |
**Control Flow**                         | 185, 193, 203, 273 |
**Pseudo loops**                         | 280  |
**Converting expressions**               | 195  |
**Copy-on-write optimization**           | 113  |
**Core data services (CDS) views**       | 51, 268 |
**Naming**                               | 140  |
**Coupling**                             | 297  |
**Undesired**                            | 309  |
**Coverage report**                      | 260, 294 |
**CREATE OBJECT**                        | 93   |
**Cretional patterns**                   | 89   |
**Credit score service class**           | 106  |
**Cross-functional teams**               | 334-335 |
**Currency conversion**                  | 290  |
**Service**                              | 53   |
**Custom assertions**                    | 275, 276 |
**Constraints**                          | 277  |
**Parameters**                           | 276  |
**Custom stubs**                         | 284  |
**CX_DYNAMIC_CHECK**                     | 244  |
**CX_NO_CHECK**                          | 243  |
**Superclass**                           | 246  |
**CX_STATIC_CHECK**                      | 242, 247 |
**Cyclical dependency**                  | 309  |
**Cyclical complexity metric**           | 190, 325 |

**D**

- Data keyword                           | 146  |
- chaining                               | 150  |
- Database tables                        | 173  |
- Decision trees                         | 190  |
- Declaration                            | 145-146 |
- alignment                              | 222  |
- anonymous                              | 147  |
- chaining                               | 149  |
- inline                                  | 44, 146-127 |
- scope                                   | 149  |
- strings                                 | 160-161 |
- up front                                | 149  |
- Declarative programs                   | 38   |
- Deconstruction                         | 194  |
- Default constructor                    | 84   |
- Default fail method                     | 56, 88 |
### Index

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input-output parameters</td>
<td>102</td>
</tr>
<tr>
<td>Insert statement</td>
<td>175</td>
</tr>
<tr>
<td>Instance methods</td>
<td>96, 111</td>
</tr>
<tr>
<td>Self reference me</td>
<td>129</td>
</tr>
<tr>
<td>Inheritance</td>
<td>38, 91</td>
</tr>
<tr>
<td>Integration tests</td>
<td>265, 289</td>
</tr>
<tr>
<td>Interface segregation principle</td>
<td>54–55</td>
</tr>
<tr>
<td>Interfaces</td>
<td>40, 51–52, 58–59</td>
</tr>
<tr>
<td>Account</td>
<td>52</td>
</tr>
<tr>
<td>Cloning</td>
<td>124</td>
</tr>
<tr>
<td>Constants</td>
<td>154</td>
</tr>
<tr>
<td>Exceptions</td>
<td>241</td>
</tr>
<tr>
<td>Factory</td>
<td>91</td>
</tr>
<tr>
<td>Global</td>
<td>76</td>
</tr>
<tr>
<td>Members</td>
<td>55</td>
</tr>
<tr>
<td>Method takeover</td>
<td>42</td>
</tr>
<tr>
<td>Naming</td>
<td>136</td>
</tr>
<tr>
<td>Option methods</td>
<td>56</td>
</tr>
<tr>
<td>Public</td>
<td>293</td>
</tr>
<tr>
<td>Public instance methods</td>
<td>99</td>
</tr>
<tr>
<td>Static creation methods</td>
<td>87</td>
</tr>
<tr>
<td>Visibility</td>
<td>81</td>
</tr>
<tr>
<td>Internal package interfaces</td>
<td>302</td>
</tr>
<tr>
<td>Internal tables</td>
<td>148, 171</td>
</tr>
<tr>
<td>Append</td>
<td>174</td>
</tr>
<tr>
<td>Categories</td>
<td>171</td>
</tr>
<tr>
<td>Declaring</td>
<td>173</td>
</tr>
<tr>
<td>Default keys</td>
<td>173</td>
</tr>
<tr>
<td>Insert</td>
<td>175</td>
</tr>
<tr>
<td>Looping variables</td>
<td>151</td>
</tr>
<tr>
<td>Number of rows</td>
<td>182</td>
</tr>
<tr>
<td>Populations</td>
<td>46</td>
</tr>
<tr>
<td>Retrieve contents</td>
<td>176–177</td>
</tr>
<tr>
<td>Unnecessary table reads</td>
<td>180</td>
</tr>
<tr>
<td>Invariants</td>
<td>75</td>
</tr>
<tr>
<td>Invoking methods</td>
<td>102</td>
</tr>
<tr>
<td>Irrelevant constructs</td>
<td>34</td>
</tr>
<tr>
<td>IS INITIAL statements</td>
<td>193</td>
</tr>
<tr>
<td>IS NOT INITIAL keyword</td>
<td>165</td>
</tr>
<tr>
<td>IS NOT statements</td>
<td>193</td>
</tr>
<tr>
<td>Is-a relationships</td>
<td>46</td>
</tr>
<tr>
<td>I-shaped profile</td>
<td>335</td>
</tr>
<tr>
<td>Iterations</td>
<td>167</td>
</tr>
<tr>
<td>Iterator pattern</td>
<td>78</td>
</tr>
<tr>
<td>Layer-based hierarchies</td>
<td>306</td>
</tr>
<tr>
<td>Layers</td>
<td>135, 305</td>
</tr>
<tr>
<td>Leadership styles</td>
<td>321</td>
</tr>
<tr>
<td>Learning techniques</td>
<td>330</td>
</tr>
<tr>
<td>Legacy code</td>
<td>26, 39, 120, 218, 289, 219, 219, declarations</td>
</tr>
<tr>
<td>Inheritance</td>
<td>70</td>
</tr>
<tr>
<td>Learning techniques</td>
<td>333</td>
</tr>
<tr>
<td>Naming</td>
<td>142, 269</td>
</tr>
<tr>
<td>Test suite</td>
<td>124, 290</td>
</tr>
<tr>
<td>Line breaks</td>
<td>220, 224</td>
</tr>
<tr>
<td>Line lengths</td>
<td>219</td>
</tr>
<tr>
<td>LINE_EXISTS function</td>
<td>177</td>
</tr>
<tr>
<td>Lines</td>
<td>183</td>
</tr>
<tr>
<td>Liskov substitution principle</td>
<td>55, 69, 102</td>
</tr>
<tr>
<td>Literals</td>
<td>153</td>
</tr>
<tr>
<td>String</td>
<td>160</td>
</tr>
<tr>
<td>Local area</td>
<td>77</td>
</tr>
<tr>
<td>Local classes</td>
<td>77, 79</td>
</tr>
<tr>
<td>Constructors</td>
<td>85</td>
</tr>
<tr>
<td>Local friends</td>
<td>266</td>
</tr>
<tr>
<td>LOOP AT statement</td>
<td>176, 178–180</td>
</tr>
<tr>
<td>Looping variables</td>
<td>149, 151</td>
</tr>
<tr>
<td>Loops</td>
<td>125, 149, 151, 176, 178</td>
</tr>
<tr>
<td>Pseudo</td>
<td>185, 200</td>
</tr>
<tr>
<td>Undefined signature</td>
<td>203</td>
</tr>
<tr>
<td>Magic numbers</td>
<td>153</td>
</tr>
<tr>
<td>Assertion</td>
<td>271</td>
</tr>
<tr>
<td>Main packages</td>
<td>42</td>
</tr>
<tr>
<td>Maintenance</td>
<td>196, 216</td>
</tr>
<tr>
<td>Managers</td>
<td>329</td>
</tr>
<tr>
<td>Meaningful names</td>
<td>133</td>
</tr>
<tr>
<td>Measurements</td>
<td>37</td>
</tr>
<tr>
<td>MESSAGE statements</td>
<td>229, 231</td>
</tr>
<tr>
<td>Messages</td>
<td>160, 202, 229, 237</td>
</tr>
<tr>
<td>Add to exceptions</td>
<td>247</td>
</tr>
<tr>
<td>ADT</td>
<td>239</td>
</tr>
<tr>
<td>Classes</td>
<td>230, 241</td>
</tr>
<tr>
<td>Clean code</td>
<td>232</td>
</tr>
<tr>
<td>Comments</td>
<td>211</td>
</tr>
<tr>
<td>Containers</td>
<td>112</td>
</tr>
<tr>
<td>Error</td>
<td>271</td>
</tr>
<tr>
<td>Handling</td>
<td>229</td>
</tr>
<tr>
<td>ID</td>
<td>230</td>
</tr>
<tr>
<td>Map to exceptions</td>
<td>238</td>
</tr>
<tr>
<td>Number</td>
<td>230</td>
</tr>
<tr>
<td>Raising</td>
<td>161</td>
</tr>
<tr>
<td>Types</td>
<td>229</td>
</tr>
<tr>
<td>Variables</td>
<td>231</td>
</tr>
</tbody>
</table>

### Meta-quality

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>26, 34, 95, 188, 223</td>
</tr>
<tr>
<td>Abstraction levels</td>
<td>117</td>
</tr>
<tr>
<td>Assertion</td>
<td>269, 277</td>
</tr>
<tr>
<td>Body</td>
<td>115, 129</td>
</tr>
<tr>
<td>Boolean parameters</td>
<td>107</td>
</tr>
<tr>
<td>Calling</td>
<td>125</td>
</tr>
<tr>
<td>CASE statements</td>
<td>197</td>
</tr>
<tr>
<td>CHECK statements</td>
<td>187</td>
</tr>
<tr>
<td>Check versus return</td>
<td>124</td>
</tr>
<tr>
<td>Closing brackets</td>
<td>223</td>
</tr>
<tr>
<td>Define outputs</td>
<td>108</td>
</tr>
<tr>
<td>Do one thing</td>
<td>115</td>
</tr>
<tr>
<td>Exceptions</td>
<td>238, 242</td>
</tr>
<tr>
<td>Function module wrapping</td>
<td>236</td>
</tr>
<tr>
<td>IF statements</td>
<td>189</td>
</tr>
<tr>
<td>Inheritance</td>
<td>102</td>
</tr>
<tr>
<td>Line breaks</td>
<td>226</td>
</tr>
<tr>
<td>Namespaces</td>
<td>89</td>
</tr>
<tr>
<td>Naming</td>
<td>110, 137</td>
</tr>
<tr>
<td>Nesting</td>
<td>45</td>
</tr>
<tr>
<td>Object-oriented programming</td>
<td>95</td>
</tr>
<tr>
<td>Overloading</td>
<td>86, 103</td>
</tr>
<tr>
<td>Parameters</td>
<td>102</td>
</tr>
<tr>
<td>Parameters formatting</td>
<td>223</td>
</tr>
<tr>
<td>Redefinition</td>
<td>100</td>
</tr>
<tr>
<td>Refactoring</td>
<td>202</td>
</tr>
<tr>
<td>Remove comments</td>
<td>206</td>
</tr>
<tr>
<td>Size</td>
<td>120</td>
</tr>
<tr>
<td>Small</td>
<td>36</td>
</tr>
<tr>
<td>Split the behaviors</td>
<td>104</td>
</tr>
<tr>
<td>Takeover by interfaces</td>
<td>277</td>
</tr>
<tr>
<td>Test</td>
<td>255, 259, 262</td>
</tr>
<tr>
<td>Test fixture</td>
<td>262</td>
</tr>
<tr>
<td>Metrics</td>
<td>325</td>
</tr>
<tr>
<td>Mob programming</td>
<td>330, 333</td>
</tr>
<tr>
<td>Mocks</td>
<td>285</td>
</tr>
<tr>
<td>Instances</td>
<td>98</td>
</tr>
<tr>
<td>When to use</td>
<td>288</td>
</tr>
<tr>
<td>Multiple parameter calls</td>
<td>224</td>
</tr>
<tr>
<td>Mutation testing</td>
<td>353</td>
</tr>
</tbody>
</table>

### Naming conventions

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>133</td>
</tr>
<tr>
<td>Domain terms</td>
<td>134</td>
</tr>
<tr>
<td>Hungarian notation</td>
<td>138</td>
</tr>
<tr>
<td>Layers</td>
<td>135</td>
</tr>
<tr>
<td>Legacy code</td>
<td>142</td>
</tr>
<tr>
<td>Noise words</td>
<td>137</td>
</tr>
<tr>
<td>Packages</td>
<td>308</td>
</tr>
<tr>
<td>Plural or singular</td>
<td>135</td>
</tr>
<tr>
<td>Reduce operations</td>
<td>167</td>
</tr>
<tr>
<td>Term consistency</td>
<td>137</td>
</tr>
<tr>
<td>Unit testing</td>
<td>267</td>
</tr>
<tr>
<td>Navigators</td>
<td>331, 333</td>
</tr>
<tr>
<td>Next code</td>
<td>215</td>
</tr>
<tr>
<td>Negative conditions</td>
<td>191–192</td>
</tr>
<tr>
<td>Nested IF statement</td>
<td>178–179</td>
</tr>
<tr>
<td>Nested types</td>
<td>159</td>
</tr>
<tr>
<td>Nesting depth</td>
<td>188, 190</td>
</tr>
<tr>
<td>Levels</td>
<td>190</td>
</tr>
<tr>
<td>New operator</td>
<td>92</td>
</tr>
<tr>
<td>No-check exceptions</td>
<td>243</td>
</tr>
<tr>
<td>Noise words</td>
<td>65, 137</td>
</tr>
<tr>
<td>Non-object-oriented code</td>
<td>43</td>
</tr>
<tr>
<td>Normalized hierarchy</td>
<td>102</td>
</tr>
<tr>
<td>NOT IS statements</td>
<td>193</td>
</tr>
<tr>
<td>NOT keyword</td>
<td>193–194</td>
</tr>
<tr>
<td>Notes</td>
<td>136</td>
</tr>
<tr>
<td>Number of statements</td>
<td>218</td>
</tr>
</tbody>
</table>

### O

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object orientation</td>
<td>37, 41</td>
</tr>
<tr>
<td>Object pattern</td>
<td>157–158</td>
</tr>
<tr>
<td>Object references</td>
<td>111</td>
</tr>
<tr>
<td>Object-oriented programming</td>
<td>38, 51, 58</td>
</tr>
<tr>
<td>Characteristics</td>
<td>51</td>
</tr>
<tr>
<td>Methods</td>
<td>95</td>
</tr>
<tr>
<td>Naming</td>
<td>136</td>
</tr>
<tr>
<td>Objects</td>
<td>58</td>
</tr>
<tr>
<td>Allowed</td>
<td>310</td>
</tr>
<tr>
<td>Bundles</td>
<td>303</td>
</tr>
<tr>
<td>Expose</td>
<td>314</td>
</tr>
<tr>
<td>Mock</td>
<td>285</td>
</tr>
<tr>
<td>Naming</td>
<td>136</td>
</tr>
<tr>
<td>Regrouping</td>
<td>297</td>
</tr>
<tr>
<td>Obsolete language constructs</td>
<td>48</td>
</tr>
<tr>
<td>Obsolete language elements</td>
<td>46</td>
</tr>
<tr>
<td>Open communication</td>
<td>143</td>
</tr>
<tr>
<td>Open SQL</td>
<td>51, 288</td>
</tr>
<tr>
<td>Open-closed principle</td>
<td>140</td>
</tr>
<tr>
<td>Optimization</td>
<td>216</td>
</tr>
<tr>
<td>Optional methods</td>
<td>56</td>
</tr>
</tbody>
</table>
Index

Static exceptions ........................................... 274
forward .................................................. 275
multiple .............................................. 275
Static methods .......................................... 55, 63, 95
call ..................................................... 98
Status messages ......................................... 230
Strings .................................................. 160
building ............................................... 161
literals ............................................... 160
static .................................................. 160
Structure packages .................................. 299
Stubs ................................................... 281-282
custom ............................................. 284
when to use ......................................... 288
Style guide ........................................... 327, 329
Subclasses ............................................. 100
Sub-methods .......................................... 190
Subpackages ......................................... 307-308
Subroutines .......................................... 95
Substitution ........................................... 40, 69
Suffixes .............................................. 65, 140
packages .......................................... 308
Superclass ........................................... 61, 65, 68-69
exceptions .......................................... 237, 245
test ................................................... 259
Superfluous statements ......................... 180
SWITCH operations .............. 198
reusing ............................................. 199
Synonyms ........................................... 137
Syntax .................................................. 193
warnings ............................................. 242-244, 246
System administrators ... 307
sy-subrc ............................................. 231, 233-234
field .................................................. 191

T

T100 exception ................................... 238, 242
T100 messages .................................... 238, 241
Tab key .................................................. 227
Table GTADER ...................................... 141
Tables .................................................. 148
categories ......................................... 172
hashed ............................................... 173
naming ............................................... 168
read ................................................... 176
sorted ............................................... 172
Target currency ...................................... 53
Teams .................................................. 321
learning ............................................. 323
members ............................................ 334
Teardown methods .................................. 262, 265

Technical debt ......................................... 316, 324
Technical packages ................................ 305
Template method .................................. 61
design pattern ..................................... 70
termination ......................................... 251
test classes .......................................... 254
executing ........................................... 260
given/when ......................................... 267
global ............................................... 257
higher level tests .................................. 257
isolate ............................................... 293
local .................................................. 254, 257-258
resetting ............................................. 265
scope ............................................... 257
test code ........................................... 292
test coverage ....................................... 294
test doubles ........................................ 279-280, 282, 293
ABAP Test Double Framework ............. 287
mock object ........................................ 287
test tools ........................................... 288
test execution ...................................... 255
test fixture methods .............................. 262
test helper classes ................................ 259, 276
test injections ...................................... 290
test methods ....................................... 255, 259, 262
executing ........................................... 262
isolation ............................................ 263, 293
naming ............................................... 268
scenarios ........................................... 265
static exceptions ................................ 275
test relations ....................................... 258, 288
test seams .......................................... 289
test superclass ..................................... 259
test-driven development (TDD) ............ 271, 291
testing subclasses .................................. 291
TEST-SEAM block ................................ 290
test IDs .............................................. 239
The Stepdown Rule ............................... 118
Then sections ........................................ 265, 267, 269, 284
Thermal protector .................................. 99
thermal switch ..................................... 71, 99, 280
timely tests ......................................... 291
todo prefix ......................................... 212
Transaction CHECKMAN ...................... 29
SAUENT_CLIENT_SETUP ..................... 255
SEII .................................................... 141
SE24 ................................................... 217
SE80 .................................................. 138, 217, 312
SE91 .................................................. 142
Translation relevance ............................. 360
options .............................................. 307

Transparent tables ..................................... 136
T-shaped profile ..................................... 334-335
Type keyword ......................................... 150
Type safety ............................................ 155, 157, 162

U

Ubiquitous language .................................. 135
Unavailable constructs .......................... 34
Unchecked exceptions .................... 243, 246, 251
Understandable code ........................... 24, 188, 195, 207
Uniformity ........................................... 217
Unit testing ......................................... 147, 188, 253, 257, 289
assertions ........................................... 269
class under test ..................................... 266
classes ............................................... 254
coverage ............................................. 154, 156
executing ............................................. 260
messages ........................................... 231
methods ............................................. 262
naming ............................................... 267
principles .......................................... 291
test doubles ......................................... 279
test seams .......................................... 289
Units ................................................... 253
Unnamed arguments ............................ 128
Unstructured programs ....................... 37
Unsuitable names .................................. 134
Upfront declaration ............................... 149
Use access ........................................... 309, 313-314
Utility classes ....................................... 807
rules .................................................. 63
Utility functions ................................... 63
Utility methods .................................... 97

V

Validation messages .................................. 124
Value copies ....................................... 113
Value objects ...................................... 63, 78, 99
attributes .......................................... 65
naming ............................................... 65

Value operator ....................................... 147
Value semantics .................................... 68
Variables ............................................. 145
alignment ........................................... 222
assignment ......................................... 45
Boolean ............................................... 163
branches and scope .............................. 148
comments .......................................... 206
declaring ............................................ 193
doc ................................................... 146
helper .................................................. 167
looping ............................................... 151
messages ........................................... 231
strings ............................................... 161
Verbs .................................................. 137
Vertical alignment ................................... 226
Vertically dense code ............................. 220
Visibility ............................................. 81
constructors ......................................... 83

W

Warnings .............................................. 213
messages ........................................... 230
WHEN block ......................................... 197
When sections ...................................... 265, 267, 284
Where-used list ..................................... 232
White grade ......................................... 322
Whitespaces ......................................... 220
Wrapping ............................................. 236
classes with functions ......................... 43

X

XSD/BOOL method ................................. 163
XXX prefix ......................................... 212

Y

Yellow grade .......................................... 322
Klaus Haeuptle is a developer, trainer, and product owner for engineering topics. During his career at SAP, he has worked as a developer on several products based on various technologies.

Florian Hoffmann is a software architect for governance, risk, and compliance applications at SAP. As an agile driver, he is constantly trying to make writing code more efficient.

Rodrigo Jordão is a development architect at SAP currently working on integrated business planning (IBP) with a focus on SAP Supply Chain Control Tower.

Michel Martin is a development architect at SAP. He has played various roles during his SAP career, including leading trainings, managing projects, and coaching teams on ASE, lean principles, and scrum.

Anagha Ravinarayan is a developer at SAP working on SAP S/4HANA Cloud and on-premise applications in capacity planning and demand-driven replenishment product areas using ABAP, SAP HANA, and SAPUI5.

Kai Westerholz is a senior developer working in the SAP S/4HANA quote-to-cash area. In this role, he focuses on creating APIs like the Sales Order Simulation API or the Sales Order Bulk Processing API.

Clean ABAP: A Style Guide for Developers

Klaus Haeuptle, Florian Hoffmann, Rodrigo Jordão, Michel Martin, Anagha Ravinarayan, Kai Westerholz

351 Pages, 2021, $79.95
ISBN 978-1-4932-2026-7

www.sap-press.com/5190