Clean ABAP

General + Comments + Formatting

Names

Use descriptive names

`max_wait_time_in_seconds`, ~~`iso3166tab`~~.

Prefer solution domain and problem domain terms

Business layers: `account`, `ledger`
Technical layers: `queue`, `tree`

Use plural

E.g. `countries` instead of `country`

Use pronounceable names

`detection_object`, ~~`dobj`~~

Avoid abbreviations

`customizing`, ~~`cust`~~

Use same abbreviations everywhere

`dobjt`, ~~`dot, dotype`~~

Use nouns for classes and verbs for methods

`account`, `withdraw`, `is_empty`

Avoid noise words

~~`data, controller, object`~~

Pick one word per concept

`read`, ~~`retrieve, query`~~

Use pattern names only if you mean them

E.g. `factory`, `façade`, `composite`

Avoid encodings, esp. Hungarian notation and prefixes

```
result = a + b
~~rv_result = iv_a + iv_b~~
```

Language

Mind the legacy

Try new syntax before applying

Mind the performance

Measure potentially slower patterns

Prefer object orientation over procedural programming

I.e. classes over functions and reports

Prefer functional over procedural language constructs

E.g. `index += 1` or `index = index + 1`
Instead of `ADD 1 to index`

Avoid obsolete statements

~~`MOVE 42 to b.`~~

Use design patterns wisely

I.e. where appropriate

Comments

Express yourself in code, not in comments

```
assert_is_valid( input )
~~"checks whether user input is valid~~
~~check( x )~~
```

Comments are no excuse for bad names

```
DATA total_sum
~~" the total sum~~
~~DATA s~~
```

Use methods instead of comments to segment your code

```
do_a( ).
do_b( ).
~~" do a~~
~~a = b + 1.~~
~~" do b~~
~~x = a / 10.~~
```

Write comments to explain the why, not the what

```
" can be missing if …
~~" reads the itab~~
READ TABLE itab
```

Design goes into the design documents, not the code

```
~~" some general observations on this~~
```

Comment with ", not with *

```
" inlines nicely
* aligns to weird places
```

Put comments before the statement they relate to

```
" right here
do_it( ). ~~" not there~~
~~" nor there~~
```

Delete code instead of commenting it

```
~~" READ TABLE~~
```

Use FIXME, TODO, and XXX and add your ID

```
" FIXME FH check sy-subrc!
```

Don't add method signature and end-of comments

```
ENDIF. ~~" IF a = 0.~~
```

Don't duplicate message texts as comments

```
" Business document not found
MESSAGE e100.
```

ABAP Doc only for public APIs

```
PRIVATE SECTION.
  ~~"! Reads something~~
  METHODS read_something
```

Formatting

Optimize for reading, not for writing

```
DATA: a
      ~~,~~ b.
```

Use the Pretty Printer before activating

Always!

Use your Pretty Printer team settings

```
Always!
```

No more than one statement per line

```
don't( ). ~~do_this( ).~~
```

Stick to a reasonable line length

<= 120 characters

Condense your code

No whitespace in weird places

Add a single blank line to separate things, but not more

No whitespace in weird places

Don't obsess with separating blank lines

No whitespace in weird places

Align assignments to the same object, but not to different ones

```
structure-type = 'A'.
structure-id   = '4711'.
```

Close brackets at line end

```
~~updater->update( this~~
~~).~~
```

Keep single parameter calls on one line

```
just_like( that )
```

Keep parameters behind the call

```
break_only(
    if_the_line_gets_too_long ).
```

If you break, indent parameters under the call

```
~~DATA(sum) = add_two_numbers(~~
~~    value_1 = 5~~
~~    value_2 = 6 ).~~
```

Line-break multiple parameters

```
~~add_two_numbers( a = 5 b = 6 ).~~
```

Align parameters

Break the call to a new line if the line gets too long

```
DATA(result) =
  some_object->some_interface~a_method(
    a = 1
    b = 2 ).
```

Indent and snap to tab

Don't force people to add single spaces

Indent in-line declarations like method calls

```
merge( a = VALUE #( b = 'X'
                    c = 'A' ) ).
```

Don't align type clauses

```
DATA name TYPE seoclsname.
DATA reader TYPE REF TO /clean/reader.
```

Clean ABAP

Variables + Statements + Classes

Constants

Use constants instead of magic numbers

E.g. `typekind_date` instead `'D'`

Prefer enumeration classes over constants interfaces

E.g. class `message_severity` over interface `common_constants`

If you don't use enumeration classes, group your constants

Don't mix unrelated constants in same structure

Variables

Prefer inline over up-front declarations

```
DATA(name) = 'something'
~~DATA: name TYPE char30~~
```

Don't declare inline in optional branches

```
~~IF has_entries = abap_true.~~
~~  DATA(value) = 1.~~
```

Do not chain up-front declarations

```
DATA name TYPE seoclsname.
DATA reader TYPE REF TO something.
```

Prefer REF TO over FIELD-SYMBOL

```
LOOP AT itab REFERENCE INTO …
```

Tables

Use the right table type

`HASHED:` large, filled at once, never modified, read often

`SORTED:` large, always sorted, filled over time or modified, read often

`STANDARD:` small, array-like

Avoid DEFAULT_KEY

```
DATA itab TYPE … WITH EMPTY KEY
~~DATA itab TYPE … WITH DEFAULT KEY~~
```

Prefer INSERT INTO TABLE over APPEND TO

Except to express that row *must* be last

Prefer LINE_EXISTS over READ TABLE

```
IF line_exists( itab[ key = 'A' ] )
```

Prefer READ TABLE over LOOP AT

```
~~LOOP AT my_table … WHERE key = 'A'.~~
~~  EXIT.~~
```

PREFER LOOP AT WHERE over nested IF

```
LOOP AT my_table … WHERE key = 'A'.
```

Strings

Use ` to define literals

```
CONSTANTS a TYPE string VALUE `abc`
```

Use | to assemble text

```
text = |Received { http_code }|
```

Booleans

Use Booleans wisely

Enumerations often make more sense

Use ABAP_BOOL for Booleans

`DATA has_entries TYPE abap_bool` or `BOOLE_D` where DDIC type needed

Use ABAP_TRUE and ABAP_FALSE for comparisons

Instead `'X'`, `space`, and `IS INITIAL`

Use XSDBOOL to set Boolean variables

```
empty = xsdbool( itab IS INITIAL )
```

Conditions

Try to make conditions positive

```
IF has_entries = abap_true.
```

Consider decomposing complex conditions

```
DATA(example_provided) = xsdbool(…)
IF example_provided = abap_true AND
   one_example_fits = abap_true.
```

Consider extracting complex conditions

```
IF is_provided( example ).
```

Ifs

No empty IF branches

```
~~IF has_entries = abap_true.~~
~~ELSE.~~
```

Prefer CASE to ELSE IF for multiple alternative conditions

```
CASE type.
  WHEN this.
  WHEN OTHERS.
ENDCASE.
```

Keep the nesting depth low

```
~~ELSE.~~
~~  IF <other>.~~
~~  ELSE.~~
~~    IF <something>.~~
```

Regular expressions

Prefer simpler methods to regular expressions

```
IF input IS NOT INITIAL.
~~IF matches( … regex = '.+' ).~~
```

Prefer basis checks to regular expressions

```
CALL FUNCTION 'SEO_CLIF_CHECK_NAME'
~~pattern = '[A-Z][A-Z0-9_]{0,29}'~~
```

Consider assembling complex regular expressions

```
CONSTANTS classes …
CONSTANTS interfaces …
… = |{ classes }|{ interfaces }|.
```

Classes: Object orientation

Prefer objects to static classes

Prefer composition over inheritance

```
DATA delegate TYPE REF TO
~~CLASS a DEFINITION INHERITING FROM~~
```

Don't mix stateful and stateless in the same class

Classes: Scope

Global by default, local only in exceptional cases

```
~~CLASS lcl_some_helper~~
```

FINAL if not designed for inheritance

```
CLASS a DEFINITION FINAL
```

Members PRIVATE by default, PROTECTED only if needed

```
PRIVATE SECTION.
  DATA attribute
```

Consider using immutable instead of getter

```
CLASS data_container
  DATA a TYPE i READ-ONLY
```

Use READ-ONLY sparingly

```
READ-ONLY
```

Classes: Constructors

Prefer NEW over CREATE OBJECT

```
DATA(a) = NEW b( ).
~~CREATE OBJECT a TYPE b~~
```

If your global class is CREATE PRIVATE, leave the CONSTRUCTOR public

```
CLASS a DEFINITION CREATE PRIVATE.
  PUBLIC SECTION.
    METHODS constructor
```

Prefer multiple static factory methods over optional parameters

```
~~METHODS constructor~~
~~  IMPORTING~~
~~    a OPTIONAL~~
~~    b OPTIONAL~~
```

Use descriptive names for multiple constructor methods

```
METHODS create_from_sample
METHODS create_from_definition
```

Make singletons only where multiple instances don't make sense

```
~~DATA singleton~~
```

Clean ABAP

Methods + Exceptions

Methods: Calls

Prefer functional over procedural calls

```
do_it( ).
~~CALL METHOD do_it.~~
```

Omit RECEIVING

```
DATA(a) = do_it( ).
~~do_it( RECEIVING result = a ).~~
```

Omit the optional keyword EXPORTING

```
do_it( a = b ).
~~do_it( EXPORTING a = b ).~~
```

Omit the parameter name in single parameter calls

```
do_it( b ).
~~do_it( a = b ).~~
```

Methods: Object orientation

Prefer instance to static methods

```
METHODS a
~~CLASS-METHODS a~~
```

Public instance methods should be part of an interface

```
INTERFACES the_interface.
~~METHODS a~~
```

Methods: Method body

Do one thing, do it well, do it only

Focus on the happy path or error handling, but not both

```
TRY.
  " focus here
CATCH.
  " do somewhere else
ENDTRY.
```

Descend one level of abstraction

```
~~do_something_high_level ( ).~~
~~DATA(low_level_op) = |a { b }|.~~
```

Keep methods small

3-5 statements, o~~ne page, 1000 lines~~

Methods: Control flow

Fail fast

```
~~METHOD do_it.~~
~~  " some more actions~~
~~  CHECK input IS NOT INITIAL.~~
```

CHECK or RETURN

```
METHOD do_it.
  CHECK input IS NOT INITIAL.
```

Avoid CHECK in other positions

```
~~LOOP AT itab INTO DATA(row).~~
~~  CHECK row IS NOT INITIAL.~~
```

Methods: Parameter number

Aim for few IMPORTING parameters, at best less than three

```
~~METHODS a IMPORTING b c d e~~
```

Split methods instead of adding OPTIONAL parameters

```
METHODS a IMPORTING b
METHODS c IMPORTING d
~~METHODS x~~
~~  IMPORTING b~~
~~            d~~
```

Use PREFERRED parameter sparingly

```
~~METHODS do_it~~
~~  IMPORTING a PREFERRED~~
~~            B TYPE i~~
```

RETURN, EXPORT, or CHANGE exactly one parameter

```
~~METHODS do_it~~
~~  EXPORTING a~~
~~  CHANGING b~~
```

Methods: Parameter types

Prefer RETURNING over EXPORTING

```
METHODS a RETURNING b
~~METHODS a EXPORTING b~~
```

RETURNING large tables is usually okay

```
METHODS a RETURNING b TYPE TABLE
~~METHODS a EXPORTING b TYPE TABLE~~
```

Use either RETURNING or EXPORTING or CHANGING, but not a combination

```
~~METHODS do_it~~
~~  EXPORTING a~~
~~  CHANGING b~~
```

Use CHANGING sparingly, where suited

```
METHODS IMPORTING … RETURNING …
~~METHODS CHANGING~~
```

Split method instead of Boolean input parameter

```
METHODS do_it_without_saving
METHODS do_it_and_save
~~METHODS do_it IMPORTING and_save~~
```

Methods: Parameter names

Consider calling the RETURNING parameter RESULT

```
METHODS sum RETURNING result
~~METHODS sum RETURNING sum~~
```

Methods: Parameter initialization

Clear or overwrite EXPORTING reference parameters

```
CLEAR et_result.
```

Don't clear VALUE parameters

```
~~CLEAR rv_result.~~
```

Error handling: Return codes

Prefer exceptions to return codes

```
METHODS check RAISING EXCEPTION
~~METHODS check RETURNING result~~
```

Don't let failures slip through

```
DATA(result) = check( input )
IF result = abap_false.
```

Error handling: Exceptions

Exceptions are for errors, not for regular cases

```
RAISE EXCEPTION db_read_failure
~~RAISE EXCEPTION not_enough_money~~
```

Use class-based exceptions

```
METHODS do_it RAISING EXCEPTION
~~METHODS do_it EXCEPTIONS~~
```

Error handling: Throwing

Use own super classes

```
CLASS our_products_static_check
INHERITING FROM cx_static_check
```

Throw one type of exception

```
~~METHODS a RAISING EXCEPTION b c d~~
```

Use sub-classes to enable callers to distinguish error situations

```
METHODS do_it RAISING EXCEPTION r
CLASS a INHERITING FROM r
CLASS b INHERITING FROM r
```

Throw CX_STATIC_CHECK for manageable situations

```
RAISE EXCEPTION no_customizing
```

Throw CX_NO_CHECK for usually unrecoverable situations

```
RAISE EXCEPTION db_unavailable
```

Consider CX_DYNAMIC_CHECK for avoidable exceptions

```
RAISE EXCEPTION division_by_zero
```

Dump for totally unrecoverable situations

```
~~RAISE EXCEPTION out_of_memory~~
```

Prefer RAISE EXCEPTION NEW to RAISE EXCEPTION TYPE

```
RAISE EXCEPTION NEW a( ).
~~RAISE EXCEPTION TYPE a.~~
```

Error handling: Catching

Wrap foreign exceptions instead of letting them invade your code

```
CATCH foreign INTO DATA(error).
  RAISE EXCEPTION NEW my( error ).
~~  RAISE EXCEPTION error.~~
```

Clean ABAP

Testing

Principles

Write testable code

There are no tricks to writing tests, there are only tricks to writing testable code. (Google)

Enable others to mock you

```
CLASS my_super_object DEFINITION.
  INTERFACES you_can_mock_this.
```

Readability rules

```
given_some_data( ).
do_the_good_thing( ).
and_assert_that_it_worked( ).
```

Don't make copies or write test reports

```
~~REPORT zmy_copy.~~
~~" for playing around~~
```

Test publics, not private internals

```
CLASS unit_tests DEFINITION ~~LOCAL FRIENDS~~
```

Don't obsess about coverage

60% ~~-> all done!~~

Test classes

Call local test classes by their purpose

```
CLASS unit_tests
~~CLASS tests_for_the_class_under_test~~
```

Put tests in local classes

```
~~REPORT some_tests_for_this~~
```

Code under test

Name the code under test meaningfully, or default to CUT

```
DATA switch
DATA cut
```

Test interfaces, not classes

```
DATA cut TYPE REF TO some_interface
~~DATA cut TYPE REF TO some_class~~
```

Extract the call to the code under test to its own method

```
METHODS map_xml_to_itab
  IMPORTING
    xml_string TYPE string
    config     TYPE … DEFAULT …
    format     TYPE … DEFAULT ….

METHOD map_xml_to_itab.
  result = cut->map_xml_to_itab( … ).
ENDMETHOD.
```

Allows tests to focus on the parameters that are really needed:

```
METHOD some_test.
  map_xml_to_itab( `<xml></xml>` ).
ENDMETHOD.
```

Injection

Use dependency inversion to inject test doubles

```
cut = NEW( stub_db_reader )
~~cut->set_db_reader( stub_db_reader )~~
~~cut->db_reader  = stub_db_reader~~
```

Use CL_ABAP_TESTDOUBLE

before writing custom stubs and mocks

Exploit the test tools

CL_OSQL_REPLACE, CDS Test Framework, Avalon

Use test seams as temporary workaround

They are *not* a permanent solution!

Use LOCAL FRIENDS to access the dependency-inverting constructor

if it's hidden away

Don't misuse LOCAL FRIENDS to invade the tested code

```
~~CLASS unit_tests LOCAL FRIENDS cut.~~
~~cut->db_reader  = stub_db_reader~~
```

Don't change the productive code to make the code testable

```
~~IF in_test_mode = abap_true.~~
```

Don't sub-class to mock methods

Use test seams or OSQL_REPLACE or extract the methods to own class

Don't mock stuff that's not needed

```
~~DATA unused_dependency~~
```

Don't build test frameworks

```
~~setup( test_case_id = '4711' )~~
```

Test Methods

Test methods names: reflect what's given and expected

```
METHODS accepts_emtpy_user_input
~~METHODS test_1~~
```

Use given-when-then

```
given_some_data( ).
do_the_good_thing( ).
assert_that_it_worked( ).
```

"When" is exactly one call

```
given_some_data( ).
do_the_good_thing( ).
~~and_another_good_thing( ).~~
assert_that_it_worked( ).
```

Don't add a TEARDOWN unless you really need it

```
" recreated in setup anyway
~~METHOD teardown.~~
~~  CLEAR stub_db_reader~~
~~ENDMETHOD.~~
```

Test Data

Make it easy to spot meaning

```
METHODS accepts_emtpy_user_input
~~METHODS test_1~~
```

Make it easy to spot differences

```
given_some_data( ).
do_the_good_thing( ).
assert_that_it_worked( ).
```

Use constants to describe purpose and importance of test data

```
CONSTANTS some_nonsense_key …
```

Assertions

Few, focused assertions

```
~~assert_not_initial( itab ).~~
assert_equals( act = itab exp = exp ).
```

Use the right assert type

```
assert_equals( act = itab exp = exp ).
~~assert_true( itab = exp ).~~
```

Assert content, not quantity

```
assert_contains_message( key )
~~assert_equals( act = lines( messages )~~
~~                exp = 3 ).~~
```

Assert quality, not content

```
assert_all_lines_shorter_than( … )
```

Use FAIL to check for expected exceptions

```
METHOD throws_on_empty_input.
  TRY.
      " when
      cut->do_something( '' ).
      cl_abap_unit_assert=>fail( ).
    CATCH /clean/some_exception.
      " then
  ENDTRY.
ENDMETHOD.
```

Forward unexpected exceptions instead of catching and failing

```
METHODS throws RAISING EXCEPTION bad
```

Write custom asserts to shorten code and avoid duplication

```
assert_table_contains( row )
~~READ TABLE itab~~
~~assert_subrc( )~~
```