Clean ABAP

The Golden Rules

Names
Use descriptive names
`max_wait_time_in_seconds`, ~~`iso3166tab`~~.

Language
Prefer object orientation over imperative programming
I.e. classes over functions and reports

Prefer functional over procedural language constructs
E.g. `index += 1` instead `ADD 1 to index`

Comments
Express yourself in code, not in comments

Delete code instead of commenting it

Formatting
Optimize for reading, not for writing

Constants
Use constants instead of magic numbers
E.g. `typekind_date` instead `'D'`

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

Booleans
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.
```

Ifs
Keep the nesting depth low
```
ELSE.
  IF <other>.
  ELSE.
    IF <something>.
```

Regular expressions
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
Members PRIVATE by default, PROTECTED only if needed

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( ).
```

Test classes
Call local test classes by their purpose
```
CLASS unit_tests
CLASS tests_for_the_class_under_test
```

Code under test
Test interfaces, not classes
```
DATA cut TYPE REF TO some_interface
DATA cut TYPE REF TO some_class
```

Injection
Use test seams as temporary workaround
They are *not* a permanent solution!

Don't misuse LOCAL FRIENDS to invade the tested code
```
CLASS unit_tests LOCAL FRIENDS cut.
cut->db_reader  = stub_db_reader
```

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( ).
```

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( … )
```

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

Descend one level of abstraction
```
do_something_high_level ( ).
DATA(low_level_op) = |a { b }|.
```

Keep methods small
3-5 statements, ~~one page, 1000 lines~~

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
```

RETURN, EXPORT, or CHANGE exactly one parameter
```
METHODS do_it
  EXPORTING a
  CHANGING b
```

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

Use class-based exceptions
```
METHODS do_it RAISING EXCEPTION
METHODS do_it EXCEPTIONS
```

Error handling: Throwing
Throw one type of exception
```
METHODS a RAISING EXCEPTION b c d
```

Throw CX_STATIC_CHECK for manageable situations
```
RAISE EXCEPTION no_customizing
```

Throw CX_NO_CHECK for usually unrecoverable situations
```
RAISE EXCEPTION db_unavailable
```

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.
```