Contents

[**Clean ABAP** 1](#_Toc15139901)

[**Content** 1](#_Toc15139902)

[**How to** 7](#_Toc15139903)

[**How to Get Started with Clean Code** 7](#_Toc15139904)

[**How to Refactor Legacy Code** 8](#_Toc15139905)

[**How to Check Automatically** 8](#_Toc15139906)

[**How to Relate to Other Guides** 9](#_Toc15139907)

[**How to Disagree** 9](#_Toc15139908)

[**Names** 9](#_Toc15139909)

[**Use descriptive names** 10](#_Toc15139910)

[**Prefer solution domain and problem domain terms** 10](#_Toc15139911)

[**Use plural** 11](#_Toc15139912)

[**Use pronounceable names** 11](#_Toc15139913)

[**Avoid abbreviations** 11](#_Toc15139914)

[**Use same abbreviations everywhere** 11](#_Toc15139915)

[**Use nouns for classes and verbs for methods** 12](#_Toc15139916)

[**Avoid noise words such as "data", "info", "object"** 12](#_Toc15139917)

[**Pick one word per concept** 12](#_Toc15139918)

[**Use pattern names only if you mean them** 13](#_Toc15139919)

[**Avoid encodings, esp. Hungarian notation and prefixes** 13](#_Toc15139920)

[**Language** 13](#_Toc15139921)

[**Mind the legacy** 13](#_Toc15139922)

[**Mind the performance** 14](#_Toc15139923)

[**Prefer object orientation to procedural programming** 14](#_Toc15139924)

[**Prefer functional to procedural language constructs** 15](#_Toc15139925)

[**Avoid obsolete language elements** 15](#_Toc15139926)

[**Use design patterns wisely** 16](#_Toc15139927)

[**Constants** 16](#_Toc15139928)

[**Use constants instead of magic numbers** 17](#_Toc15139929)

[**Prefer enumeration classes to constants interfaces** 17](#_Toc15139930)

[**If you don't use enumeration classes, group your constants** 18](#_Toc15139931)

[**Variables** 18](#_Toc15139932)

[**Prefer inline to up-front declarations** 18](#_Toc15139933)

[**Don't declare inline in optional branches** 19](#_Toc15139934)

[**Do not chain up-front declarations** 19](#_Toc15139935)

[**Prefer REF TO to FIELD-SYMBOL** 20](#_Toc15139936)

[**Tables** 21](#_Toc15139937)

[**Use the right table type** 21](#_Toc15139938)

[**Avoid DEFAULT KEY** 21](#_Toc15139939)

[**Prefer INSERT INTO TABLE to APPEND TO** 22](#_Toc15139940)

[**Prefer LINE\_EXISTS to READ TABLE or LOOP AT** 22](#_Toc15139941)

[**Prefer READ TABLE to LOOP AT** 22](#_Toc15139942)

[**Prefer LOOP AT WHERE to nested IF** 23](#_Toc15139943)

[**Avoid unnecessary table reads** 23](#_Toc15139944)

[**Strings** 24](#_Toc15139945)

[**Use ` to define literals** 24](#_Toc15139946)

[**Use | to assemble text** 24](#_Toc15139947)

[**Booleans** 24](#_Toc15139948)

[**Use Booleans wisely** 25](#_Toc15139949)

[**Use ABAP\_BOOL for Booleans** 25](#_Toc15139950)

[**Use ABAP\_TRUE and ABAP\_FALSE for comparisons** 25](#_Toc15139951)

[**Use XSDBOOL to set Boolean variables** 26](#_Toc15139952)

[**Conditions** 26](#_Toc15139953)

[**Try to make conditions positive** 27](#_Toc15139954)

[**Prefer IS NOT to NOT IS** 27](#_Toc15139955)

[**Consider decomposing complex conditions** 27](#_Toc15139956)

[**Consider extracting complex conditions** 28](#_Toc15139957)

[**Ifs** 28](#_Toc15139958)

[**No empty IF branches** 28](#_Toc15139959)

[**Prefer CASE to ELSE IF for multiple alternative conditions** 29](#_Toc15139960)

[**Keep the nesting depth low** 29](#_Toc15139961)

[**Regular expressions** 30](#_Toc15139962)

[**Prefer simpler methods to regular expressions** 30](#_Toc15139963)

[**Prefer basis checks to regular expressions** 30](#_Toc15139964)

[**Consider assembling complex regular expressions** 31](#_Toc15139965)

[**Classes** 31](#_Toc15139966)

[**Classes: Object orientation** 31](#_Toc15139967)

[**Scope** 33](#_Toc15139968)

[**Constructors** 36](#_Toc15139969)

[**Methods** 38](#_Toc15139970)

[**Calls** 38](#_Toc15139971)

[**Methods: Object orientation** 40](#_Toc15139972)

[**Parameter Number** 41](#_Toc15139973)

[**Parameter Types** 44](#_Toc15139974)

[**Parameter Names** 47](#_Toc15139975)

[**Parameter Initialization** 48](#_Toc15139976)

[**Method Body** 49](#_Toc15139977)

[**Control flow** 53](#_Toc15139978)

[**Error Handling** 55](#_Toc15139979)

[**Return Codes** 55](#_Toc15139980)

[**Exceptions** 56](#_Toc15139981)

[**Throwing** 57](#_Toc15139982)

[**Catching** 61](#_Toc15139983)

[**Comments** 61](#_Toc15139984)

[**Express yourself in code, not in comments** 62](#_Toc15139985)

[**Comments are no excuse for bad names** 63](#_Toc15139986)

[**Use methods instead of comments to segment your code** 63](#_Toc15139987)

[**Write comments to explain the why, not the what** 63](#_Toc15139988)

[**Design goes into the design documents, not the code** 64](#_Toc15139989)

[**Comment with ", not with \*** 64](#_Toc15139990)

[**Put comments before the statement they relate to** 64](#_Toc15139991)

[**Delete code instead of commenting it** 65](#_Toc15139992)

[**Use FIXME, TODO, and XXX and add your ID** 65](#_Toc15139993)

[**Don't add method signature and end-of comments** 65](#_Toc15139994)

[**Don't duplicate message texts as comments** 66](#_Toc15139995)

[**ABAP Doc only for public APIs** 67](#_Toc15139996)

[**Formatting** 67](#_Toc15139997)

[**Be consistent** 67](#_Toc15139998)

[**Optimize for reading, not for writing** 68](#_Toc15139999)

[**Use the Pretty Printer before activating** 68](#_Toc15140000)

[**Use your Pretty Printer team settings** 68](#_Toc15140001)

[**No more than one statement per line** 69](#_Toc15140002)

[**Stick to a reasonable line length** 69](#_Toc15140003)

[**Condense your code** 69](#_Toc15140004)

[**Add a single blank line to separate things, but not more** 70](#_Toc15140005)

[**Don't obsess with separating blank lines** 70](#_Toc15140006)

[**Align assignments to the same object, but not to different ones** 71](#_Toc15140007)

[**Close brackets at line end** 71](#_Toc15140008)

[**Keep single parameter calls on one line** 71](#_Toc15140009)

[**Keep parameters behind the call** 72](#_Toc15140010)

[**If you break, indent parameters under the call** 72](#_Toc15140011)

[**Line-break multiple parameters** 72](#_Toc15140012)

[**Align parameters** 73](#_Toc15140013)

[**Break the call to a new line if the line gets too long** 73](#_Toc15140014)

[**Indent and snap to tab** 73](#_Toc15140015)

[**Indent in-line declarations like method calls** 74](#_Toc15140016)

[**Don't align type clauses** 74](#_Toc15140017)

[**Testing** 74](#_Toc15140018)

[**Principles** 74](#_Toc15140019)

[**Test Classes** 77](#_Toc15140020)

[**Code Under Test** 78](#_Toc15140021)

[**Injection** 79](#_Toc15140022)

[**Test Methods** 84](#_Toc15140023)

[**Test Data** 85](#_Toc15140024)

[**Assertions** 86](#_Toc15140025)

**Clean ABAP**

This guide is an adoption of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/) for [ABAP](https://en.wikipedia.org/wiki/ABAP).

The [Cheat Sheet](https://github.com/SAP/styleguides/blob/master/clean-abap/cheat-sheet/CheatSheet.md) is a print-optimized version.

**Content**

* [How to](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to)
  + [How to Get Started with Clean Code](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-get-started-with-clean-code)
  + [How to Refactor Legacy Code](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-refactor-legacy-code)
  + [How to Check Automatically](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-check-automatically)
  + [How to Relate to Other Guides](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-relate-to-other-guides)
  + [How to Disagree](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-disagree)
* [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names)
  + [Use descriptive names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-descriptive-names)
  + [Prefer solution domain and problem domain terms](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-solution-domain-and-problem-domain-terms)
  + [Use plural](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-plural)
  + [Use pronounceable names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-pronounceable-names)
  + [Avoid abbreviations](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-abbreviations)
  + [Use same abbreviations everywhere](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-same-abbreviations-everywhere)
  + [Use nouns for classes and verbs for methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-nouns-for-classes-and-verbs-for-methods)
  + [Avoid noise words such as "data", "info", "object"](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-noise-words-such-as-data-info-object)
  + [Pick one word per concept](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#pick-one-word-per-concept)
  + [Use pattern names only if you mean them](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-pattern-names-only-if-you-mean-them)
  + [Avoid encodings, esp. Hungarian notation and prefixes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-encodings-esp-hungarian-notation-and-prefixes)
* [Language](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#language)
  + [Mind the legacy](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#mind-the-legacy)
  + [Mind the performance](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#mind-the-performance)
  + [Prefer object orientation to procedural programming](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-object-orientation-to-procedural-programming)
  + [Prefer functional to procedural language constructs](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-functional-to-procedural-language-constructs)
  + [Avoid obsolete language elements](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-obsolete-language-elements)
  + [Use design patterns wisely](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-design-patterns-wisely)
* [Constants](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constants)
  + [Use constants instead of magic numbers](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-constants-instead-of-magic-numbers)
  + [Prefer enumeration classes to constants interfaces](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-enumeration-classes-to-constants-interfaces)
  + [If you don't use enumeration classes, group your constants](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#if-you-dont-use-enumeration-classes-group-your-constants)
* [Variables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#variables)
  + [Prefer inline to up-front declarations](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-inline-to-up-front-declarations)
  + [Don't declare inline in optional branches](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-declare-inline-in-optional-branches)
  + [Do not chain up-front declarations](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#do-not-chain-up-front-declarations)
  + [Prefer REF TO to FIELD-SYMBOL](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-ref-to-to-field-symbol)
* [Tables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#tables)
  + [Use the right table type](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-the-right-table-type)
  + [Avoid DEFAULT KEY](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-default-key)
  + [Prefer INSERT INTO TABLE to APPEND TO](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-insert-into-table-to-append-to)
  + [Prefer LINE\_EXISTS to READ TABLE or LOOP AT](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-line_exists-to-read-table-or-loop-at)
  + [Prefer READ TABLE to LOOP AT](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-read-table-to-loop-at)
  + [Prefer LOOP AT WHERE to nested IF](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-loop-at-where-to-nested-if)
  + [Avoid unnecessary table reads](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-unnecessary-table-reads)
* [Strings](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#strings)
  + [Use ` to define literals](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use--to-define-literals)
  + [Use | to assemble text](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use--to-assemble-text)
* [Booleans](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#booleans)
  + [Use Booleans wisely](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-booleans-wisely)
  + [Use ABAP\_BOOL for Booleans](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-abap_bool-for-booleans)
  + [Use ABAP\_TRUE and ABAP\_FALSE for comparisons](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-abap_true-and-abap_false-for-comparisons)
  + [Use XSDBOOL to set Boolean variables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-xsdbool-to-set-boolean-variables)
* [Conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#conditions)
  + [Try to make conditions positive](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#try-to-make-conditions-positive)
  + [Prefer IS NOT to NOT IS](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-is-not-to-not-is)
  + [Consider decomposing complex conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-decomposing-complex-conditions)
  + [Consider extracting complex conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-extracting-complex-conditions)
* [Ifs](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#ifs)
  + [No empty IF branches](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#no-empty-if-branches)
  + [Prefer CASE to ELSE IF for multiple alternative conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-case-to-else-if-for-multiple-alternative-conditions)
  + [Keep the nesting depth low](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-the-nesting-depth-low)
* [Regular expressions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#regular-expressions)
  + [Prefer simpler methods to regular expressions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-simpler-methods-to-regular-expressions)
  + [Prefer basis checks to regular expressions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-basis-checks-to-regular-expressions)
  + [Consider assembling complex regular expressions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-assembling-complex-regular-expressions)
* [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes)
  + [Classes: Object orientation](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes-object-orientation)
    - [Prefer objects to static classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-objects-to-static-classes)
    - [Prefer composition to inheritance](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-composition-to-inheritance)
    - [Don't mix stateful and stateless in the same class](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-mix-stateful-and-stateless-in-the-same-class)
  + [Scope](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#scope)
    - [Global by default, local only in exceptional cases](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#global-by-default-local-only-in-exceptional-cases)
    - [FINAL if not designed for inheritance](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#final-if-not-designed-for-inheritance)
    - [Members PRIVATE by default, PROTECTED only if needed](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#members-private-by-default-protected-only-if-needed)
    - [Consider using immutable instead of getter](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-using-immutable-instead-of-getter)
    - [Use READ-ONLY sparingly](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-read-only-sparingly)
  + [Constructors](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constructors)
    - [Prefer NEW to CREATE OBJECT](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-new-to-create-object)
    - [If your global class is CREATE PRIVATE, leave the CONSTRUCTOR public](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#if-your-global-class-is-create-private-leave-the-constructor-public)
    - [Prefer multiple static factory methods to optional parameters](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-multiple-static-factory-methods-to-optional-parameters)
    - [Use descriptive names for multiple constructor methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-descriptive-names-for-multiple-constructor-methods)
    - [Make singletons only where multiple instances don't make sense](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#make-singletons-only-where-multiple-instances-dont-make-sense)
* [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods)
  + [Calls](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#calls)
    - [Prefer functional to procedural calls](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-functional-to-procedural-calls)
    - [Omit RECEIVING](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#omit-receiving)
    - [Omit the optional keyword EXPORTING](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#omit-the-optional-keyword-exporting)
    - [Omit the parameter name in single parameter calls](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#omit-the-parameter-name-in-single-parameter-calls)
    - [Omit the self-reference me when calling an instance method](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#omit-the-self-reference-me-when-calling-an-instance-method)
  + [Methods: Object orientation](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods-object-orientation)
    - [Prefer instance to static methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-instance-to-static-methods)
    - [Public instance methods should be part of an interface](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#public-instance-methods-should-be-part-of-an-interface)
  + [Parameter Number](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-number)
    - [Aim for few IMPORTING parameters, at best less than three](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#aim-for-few-importing-parameters-at-best-less-than-three)
    - [Split methods instead of adding OPTIONAL parameters](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#split-methods-instead-of-adding-optional-parameters)
    - [Use PREFERRED PARAMETER sparingly](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-preferred-parameter-sparingly)
    - [RETURN, EXPORT, or CHANGE exactly one parameter](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#return-export-or-change-exactly-one-parameter)
  + [Parameter Types](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-types)
    - [Prefer RETURNING to EXPORTING](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-returning-to-exporting)
    - [RETURNING large tables is usually okay](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#returning-large-tables-is-usually-okay)
    - [Use either RETURNING or EXPORTING or CHANGING, but not a combination](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-either-returning-or-exporting-or-changing-but-not-a-combination)
    - [Use CHANGING sparingly, where suited](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-changing-sparingly-where-suited)
    - [Split method instead of Boolean input parameter](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#split-method-instead-of-boolean-input-parameter)
  + [Parameter Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-names)
    - [Consider calling the RETURNING parameter RESULT](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-calling-the-returning-parameter-result)
  + [Parameter Initialization](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-initialization)
    - [Clear or overwrite EXPORTING reference parameters](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clear-or-overwrite-exporting-reference-parameters)
      * [Take care if input and output could be the same](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#take-care-if-input-and-output-could-be-the-same)
    - [Don't clear VALUE parameters](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-clear-value-parameters)
  + [Method Body](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#method-body)
    - [Do one thing, do it well, do it only](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#do-one-thing-do-it-well-do-it-only)
    - [Focus on the happy path or error handling, but not both](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#focus-on-the-happy-path-or-error-handling-but-not-both)
    - [Descend one level of abstraction](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#descend-one-level-of-abstraction)
    - [Keep methods small](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-methods-small)
  + [Control flow](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#control-flow)
    - [Fail fast](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#fail-fast)
    - [CHECK vs. RETURN](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#check-vs-return)
    - [Avoid CHECK in other positions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-check-in-other-positions)
* [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling)
  + [Return Codes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#return-codes)
    - [Prefer exceptions to return codes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-exceptions-to-return-codes)
    - [Don't let failures slip through](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-let-failures-slip-through)
  + [Exceptions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#exceptions)
    - [Exceptions are for errors, not for regular cases](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#exceptions-are-for-errors-not-for-regular-cases)
    - [Use class-based exceptions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-class-based-exceptions)
  + [Throwing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing)
    - [Use own super classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-own-super-classes)
    - [Throw one type of exception](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throw-one-type-of-exception)
    - [Use sub-classes to enable callers to distinguish error situations](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-sub-classes-to-enable-callers-to-distinguish-error-situations)
    - [Throw CX\_STATIC\_CHECK for manageable exceptions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throw-cx_static_check-for-manageable-exceptions)
    - [Throw CX\_NO\_CHECK for usually unrecoverable situations](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throw-cx_no_check-for-usually-unrecoverable-situations)
    - [Consider CX\_DYNAMIC\_CHECK for avoidable exceptions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-cx_dynamic_check-for-avoidable-exceptions)
    - [Dump for totally unrecoverable situations](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dump-for-totally-unrecoverable-situations)
    - [Prefer RAISE EXCEPTION NEW to RAISE EXCEPTION TYPE](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-raise-exception-new-to-raise-exception-type)
  + [Catching](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#catching)
    - [Wrap foreign exceptions instead of letting them invade your code](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#wrap-foreign-exceptions-instead-of-letting-them-invade-your-code)
* [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments)
  + [Express yourself in code, not in comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#express-yourself-in-code-not-in-comments)
  + [Comments are no excuse for bad names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments-are-no-excuse-for-bad-names)
  + [Use methods instead of comments to segment your code](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-methods-instead-of-comments-to-segment-your-code)
  + [Write comments to explain the why, not the what](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#write-comments-to-explain-the-why-not-the-what)
  + [Design goes into the design documents, not the code](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#design-goes-into-the-design-documents-not-the-code)
  + [Comment with ", not with \*](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comment-with--not-with-)
  + [Put comments before the statement they relate to](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#put-comments-before-the-statement-they-relate-to)
  + [Delete code instead of commenting it](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#delete-code-instead-of-commenting-it)
  + [Use FIXME, TODO, and XXX and add your ID](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-fixme-todo-and-xxx-and-add-your-id)
  + [Don't add method signature and end-of comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-add-method-signature-and-end-of-comments)
  + [Don't duplicate message texts as comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-duplicate-message-texts-as-comments)
  + [ABAP Doc only for public APIs](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#abap-doc-only-for-public-apis)
* [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting)
  + [Be consistent](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#be-consistent)
  + [Optimize for reading, not for writing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#optimize-for-reading-not-for-writing)
  + [Use the Pretty Printer before activating](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-the-pretty-printer-before-activating)
  + [Use your Pretty Printer team settings](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-your-pretty-printer-team-settings)
  + [No more than one statement per line](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#no-more-than-one-statement-per-line)
  + [Stick to a reasonable line length](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#stick-to-a-reasonable-line-length)
  + [Condense your code](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#condense-your-code)
  + [Add a single blank line to separate things, but not more](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#add-a-single-blank-line-to-separate-things-but-not-more)
  + [Don't obsess with separating blank lines](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-obsess-with-separating-blank-lines)
  + [Align assignments to the same object, but not to different ones](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#align-assignments-to-the-same-object-but-not-to-different-ones)
  + [Close brackets at line end](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#close-brackets-at-line-end)
  + [Keep single parameter calls on one line](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-single-parameter-calls-on-one-line)
  + [Keep parameters behind the call](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-parameters-behind-the-call)
  + [If you break, indent parameters under the call](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#if-you-break-indent-parameters-under-the-call)
  + [Line-break multiple parameters](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#line-break-multiple-parameters)
  + [Align parameters](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#align-parameters)
  + [Break the call to a new line if the line gets too long](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#break-the-call-to-a-new-line-if-the-line-gets-too-long)
  + [Indent and snap to tab](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#indent-and-snap-to-tab)
  + [Indent in-line declarations like method calls](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#indent-in-line-declarations-like-method-calls)
  + [Don't align type clauses](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-align-type-clauses)
* [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing)
  + [Principles](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#principles)
    - [Write testable code](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#write-testable-code)
    - [Enable others to mock you](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#enable-others-to-mock-you)
    - [Readability rules](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#readability-rules)
    - [Don't make copies or write test reports](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-make-copies-or-write-test-reports)
    - [Test publics, not private internals](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-publics-not-private-internals)
    - [Don't obsess about coverage](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-obsess-about-coverage)
  + [Test Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-classes)
    - [Call local test classes by their purpose](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#call-local-test-classes-by-their-purpose)
    - [Put tests in local classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#put-tests-in-local-classes)
    - [How to execute test classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-execute-test-classes)
  + [Code Under Test](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#code-under-test)
    - [Name the code under test meaningfully, or default to CUT](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#name-the-code-under-test-meaningfully-or-default-to-cut)
    - [Test interfaces, not classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-interfaces-not-classes)
    - [Extract the call to the code under test to its own method](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#extract-the-call-to-the-code-under-test-to-its-own-method)
  + [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection)
    - [Use dependency inversion to inject test doubles](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-dependency-inversion-to-inject-test-doubles)
    - [Use CL\_ABAP\_TESTDOUBLE](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-cl_abap_testdouble)
    - [Exploit the test tools](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#exploit-the-test-tools)
    - [Use test seams as temporary workaround](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-test-seams-as-temporary-workaround)
    - [Use LOCAL FRIENDS to access the dependency-inverting constructor](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-local-friends-to-access-the-dependency-inverting-constructor)
    - [Don't misuse LOCAL FRIENDS to invade the tested code](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-misuse-local-friends-to-invade-the-tested-code)
    - [Don't change the productive code to make the code testable](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-change-the-productive-code-to-make-the-code-testable)
    - [Don't sub-class to mock methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-sub-class-to-mock-methods)
    - [Don't mock stuff that's not needed](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-mock-stuff-thats-not-needed)
    - [Don't build test frameworks](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-build-test-frameworks)
  + [Test Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-methods)
    - [Test method names: reflect what's given and expected](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-method-names-reflect-whats-given-and-expected)
    - [Use given-when-then](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-given-when-then)
    - ["When" is exactly one call](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#when-is-exactly-one-call)
    - [Don't add a TEARDOWN unless you really need it](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-add-a-teardown-unless-you-really-need-it)
  + [Test Data](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-data)
    - [Make it easy to spot meaning](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#make-it-easy-to-spot-meaning)
    - [Make it easy to spot differences](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#make-it-easy-to-spot-differences)
    - [Use constants to describe purpose and importance of test data](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-constants-to-describe-purpose-and-importance-of-test-data)
  + [Assertions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assertions)
    - [Few, focused assertions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#few-focused-assertions)
    - [Use the right assert type](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-the-right-assert-type)
    - [Assert content, not quantity](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assert-content-not-quantity)
    - [Assert quality, not content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assert-quality-not-content)
    - [Use FAIL to check for expected exceptions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-fail-to-check-for-expected-exceptions)
    - [Forward unexpected exceptions instead of catching and failing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#forward-unexpected-exceptions-instead-of-catching-and-failing)
    - [Write custom asserts to shorten code and avoid duplication](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#write-custom-asserts-to-shorten-code-and-avoid-duplication)

**How to**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to)

**How to Get Started with Clean Code**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [How to](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-get-started-with-clean-code)

If you are new to Clean Code, you should first read [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/). The [Clean Code Developer initiative](https://clean-code-developer.com/) may help you getting started with a didactically smooth stepwise introduction to the topic in general.

We recommend you to start with things that are easily understood and broadly accepted, such as [Booleans](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#booleans), [Conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#conditions), and [Ifs](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#ifs).

You will probably benefit most from the section [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods), especially [Do one thing, do it well, do it only](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#do-one-thing-do-it-well-do-it-only) and [Small](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-methods-small), because these tremendously improve the overall structure of your code.

Some topics in here can spark difficult discussions in teams that are experienced in what they do but new to Clean Code; these topics are perfectly "healthy", but people may have problems making themselves comfortable with them in the beginning.

Continue to these more controversial topics later; especially [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments), [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names), and [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) can lead to near-religious disputes and should only be addressed by teams that already saw proof of Clean Code's positive effects.

**How to Refactor Legacy Code**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [How to](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-refactor-legacy-code)

The topics [Booleans](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#booleans), [Conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#conditions), [Ifs](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#ifs), and [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) are most rewarding if you are working on a legacy project with tons of code that you cannot or do not want to change because they can be applied to new code without conflicts.

The topic [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) is very demanding for legacy projects, as it may introduce a breach between old and new code, up to a degree where sections like [Avoid encodings, esp. Hungarian notation and prefixes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-encodings-esp-hungarian-notation-and-prefixes) are better ignored.

We observed good results with a four-step plan for refactoring:

1. Get the team aboard. Communicate and explain the new style, and get everybody on the project team to agree to it. You don't need to commit all guidelines at once, just start with an undisputed small subset and evolve from there.
2. Follow the *boy scout rule* to your daily work routine: *always leave the code you edit a little cleaner than you found it*. Don't obsess with this by sinking hours into "cleaning the campsite", just spend a couple of minutes extra and observe how the improvements accumulate over time.
3. Build *clean islands*: from time to time, pick a small object or component and try to make it clean in all aspects. These islands demonstrate the benefit of what you're doing and form solidly tested home bases for further refactoring.
4. Talk about it. No matter whether you set up old-school [Fagan code reviews](https://en.wikipedia.org/wiki/Fagan_inspection), hold info sessions, or form discussion boards in your favorite chat tool: you will need to talk about your experiences and learnings, to enable the team to grow a common understanding.

**How to Check Automatically**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [How to](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-check-automatically)

There is no comprehensive suite of static code checks that automatically detect the anti-patterns we describe here.

ABAP Test Cockpit, Code Inspector, Extended Check, and Checkman provide some checks that may help you find certain issues.

[abapOpenChecks](https://github.com/larshp/abapOpenChecks), an Open Source collection of Code Inspector checks, also covers some of the described anti-patterns.

**How to Relate to Other Guides**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [How to](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-relate-to-other-guides)

Our guide follows the *spirit* of Clean Code, meaning we adjusted some things to the ABAP programming language e.g. [Throw CX\_STATIC\_CHECK for manageable exceptions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throw-cx_static_check-for-manageable-exceptions).

Some facts are from the [ABAP Programming Guidelines](https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/index.htm?file=abenabap_pgl.htm), which this guide is mostly compatible to; deviations are indicated and always in the spirit of cleaner code.

This guide also respects the [DSAG's Recommendations for ABAP Development](https://www.dsag.de/sites/default/files/dsag_recommendation_abap_development.pdf), although we are more precise in most details.

**How to Disagree**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [How to](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-disagree)

We wrote this style guide for readers who are already acquainted with Clean Code or who are right now working on that, with a strong focus on how to apply Clean Code *specifically to ABAP*.

Please mind that we therefore did not introduce all concepts in the same length and depth as the original book and related resources: these are still worth a read, especially if you disagree with things in here just because we didn't explain them very well. Use the links in the sections to read up on the background of our guidance.

You are free to discuss and disagree with anything we say here. One of the pillars of Clean Code is that *the team rules*. Just be sure to give things a fair chance before you discard them.

[CONTRIBUTING.md](https://github.com/SAP/styleguides/blob/master/CONTRIBUTING.md) suggests ways how you can change this guide or deviate from it in minor details.

**Names**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names)

**Use descriptive names**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-descriptive-names)

Use names that convey the content and meaning of things.

CONSTANTS max\_wait\_time\_in\_seconds TYPE i ...

DATA customizing\_entries TYPE STANDARD TABLE ...

METHODS read\_user\_preferences ...

CLASS /clean/user\_preference\_reader ...

Do not focus on the data type or technical encoding. They hardly contribute to understanding the code.

" anti-pattern

CONSTANTS sysubrc\_04 TYPE sysubrc ...

DATA iso3166tab TYPE STANDARD TABLE ...

METHODS read\_t005 ...

CLASS /dirty/t005\_reader ...

[Do not attempt to fix bad names by comments.](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments-are-no-excuse-for-bad-names)

Read more in *Chapter 2: Meaningful Names: Use Intention-Revealing Names* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Prefer solution domain and problem domain terms**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-solution-domain-and-problem-domain-terms)

Search for good names in the solution domain, i.e. computer science terms such as "queue" or "tree", and in the problem domain, i.e. business field terms such as "account" or "ledger".

Layers that are business-like will sound best when named according to the problem domain. This is especially true for components that are designed with Domain-Driven Design, such as APIs and business objects.

Layers that provide mostly technical functionality, such as factory classes and abstract algorithms, will sound best when named according to the solution domain.

In any case, do not attempt to make up your own language. We need to be able to exchange information between developers, product owners, partners and customers, so choose names that all of these can relate to without a customized dictionary.

Read more in *Chapter 2: Meaningful Names: Use Solution Domain Names* and *[...]: Use Problem Domain Names* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Use plural**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-plural)

There is a legacy practice at SAP to name tables of things in singular, for example country for a "table of countries". Common tendency in the outside world is to use the plural for lists of things. We therefore recommend to prefer countriesinstead.

Read more in *Chapter 2: Meaningful Names: Use Intention-Revealing Names* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Use pronounceable names**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-pronounceable-names)

We think and talk a lot about objects, so use names that you can pronounce, for example prefer detection\_object\_types to something cryptic like dobjt.

Read more in *Chapter 2: Meaningful Names: Use Pronounceable Names* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/)

**Avoid abbreviations**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-abbreviations)

If you have enough space, write out names in full. Start abbreviating only if you exceed length limitations.

If you do have to abbreviate, start with the *unimportant* words.

Abbreviating things may appear efficient at first glance, but becomes ambiguous very fast. For example, does the "cust" in cust mean "customizing", "customer", or "custom"? All three are common in SAP applications.

Read more in *Chapter 2: Meaningful Names: Make Meaningful Distinctions* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Use same abbreviations everywhere**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-same-abbreviations-everywhere)

People will search for keywords to find relevant code. Support this by using the same abbreviation for the same thing. For example, always abbreviate "detection object type" to "dobjt" instead of mixing "dot", "dotype", "detobjtype" and so on.

Read more in *Chapter 2: Meaningful Names: Use Searchable Names* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Use nouns for classes and verbs for methods**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-nouns-for-classes-and-verbs-for-methods)

Use nouns or noun phrases to name classes, interfaces, and objects:

CLASS /clean/account

CLASS /clean/user\_preferences

INTERFACE /clean/customizing\_reader

Use verbs or verb phrases to name methods:

METHODS withdraw

METHODS add\_message

METHODS read\_entries

Starting Boolean methods with verbs like is\_ and has\_ yields nice reading flow:

IF is\_empty( table ).

We recommend naming functions like methods:

FUNCTION /clean/read\_alerts

**Avoid noise words such as "data", "info", "object"**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-noise-words-such-as-data-info-object)

Omit noise words

account " instead of account\_data

alert " instead of alert\_object

or replace them with something specific that really adds value

user\_preferences " instead of user\_info

response\_time\_in\_seconds " instead of response\_time\_variable

Read more in *Chapter 2: Meaningful Names: Make Meaningful Distinctions* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/)

**Pick one word per concept**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#pick-one-word-per-concept)

METHODS read\_this.

METHODS read\_that.

METHODS read\_those.

Choose a term for a concept and stick to it; don't mix in other synonyms. Synonyms will make the reader waste time on finding a difference that's not there.

" anti-pattern

METHODS read\_this.

METHODS retrieve\_that.

METHODS query\_those.

Read more in *Chapter 2: Meaningful Names: Pick One Word per Concept* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/)

**Use pattern names only if you mean them**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-pattern-names-only-if-you-mean-them)

Don't use the names of software design patterns for classes and interfaces unless you really mean them. For example, don't call your class file\_factory unless it really implements the factory design pattern. The most common patterns include:[singleton](https://en.wikipedia.org/wiki/Singleton_pattern), [factory](https://en.wikipedia.org/wiki/Factory_method_pattern), [facade](https://en.wikipedia.org/wiki/Facade_pattern), [composite](https://en.wikipedia.org/wiki/Composite_pattern), [decorator](https://en.wikipedia.org/wiki/Decorator_pattern), [iterator](https://en.wikipedia.org/wiki/Iterator_pattern), [observer](https://en.wikipedia.org/wiki/Observer_pattern), and [strategy](https://en.wikipedia.org/wiki/Strategy_pattern).

Read more in *Chapter 2: Meaningful Names: Avoid Disinformation* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/)

**Avoid encodings, esp. Hungarian notation and prefixes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-encodings-esp-hungarian-notation-and-prefixes)

We encourage you to get rid of *all* encoding prefixes.

METHOD add\_two\_numbers.

result = a + b.

ENDMETHOD.

instead of the needlessly longer

METHOD add\_two\_numbers.

rv\_result = iv\_a + iv\_b.

ENDMETHOD.

[Avoid Encodings](https://github.com/SAP/styleguides/blob/master/clean-abap/sub-sections/AvoidEncodings.md) describes the reasoning in depth.

**Language**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#language)

**Mind the legacy**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Language](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#language) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#mind-the-legacy)

If you code for older ABAP releases, take the advice in this guide with care: Many recommendations below make use of relatively new syntax and constructs that may not be supported in older ABAP releases. Validate the guidelines you want to follow on the oldest release you must support. Do not simply discard Clean Code as a whole - the vast majority of rules (e.g. naming, commenting) will work in *any* ABAP version.

**Mind the performance**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Language](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#language) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#mind-the-performance)

If you code high performance components, take the advice in this guide with care: Some aspects of Clean Code may make things slower (more method calls) or consume more memory (more objects). ABAP has some specialties that may intensify this, for example it compares data types when calling a method, such that splitting a single large method into many sub-methods may make the code slower.

However, we strongly recommend to not optimize prematurely, based on obscure fears. The vast majority of rules (e.g. naming, commenting) has no negative impact at all. Try to build things in a clean, object-oriented way. If something is too slow, make a performance measurement. Only then should you take a fact-based decision to discard selected rules.

Some further thoughts, taken in part from Chapter 2 of [Martin Fowler's *Refactoring*](https://martinfowler.com/books/refactoring.html):

In a typical application the majority of the runtime is spent in a very small proportion of the code. As little as 10% of the code can account for 90% of the runtime, and especially in ABAP a large proportion of runtime is likely to be database time.

Thus it is not the best use of resources to spend significant effort on trying to make *all* code super-efficient all the time. We're not suggesting ignoring performance, but rather focus more on clean and well structured code during initial development, and use the profiler to identify critical areas to optimize.

In fact, we would argue that such an approach will have a net positive effect on performance because it is a more targeted optimization effort, and it should be easier to identify performance bottlenecks and easier to refactor and tune well structured code.

**Prefer object orientation to procedural programming**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Language](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#language) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-object-orientation-to-procedural-programming)

Object-oriented programs (classes, interfaces) are segmented better and can be refactored and tested more easily than procedural code (functions, programs). Although there are situations where you must provide procedural objects (a function for an RFC, a program for a transaction), these objects should do little more than call a corresponding class that provides the actual feature:

FUNCTION check\_business\_partner [...].

DATA(validator) = NEW /clean/biz\_partner\_validator( ).

result = validator->validate( business\_partners ).

ENDFUNCTION.

[Function Groups vs. Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/sub-sections/FunctionGroupsVsClasses.md) describes the differences in detail.

**Prefer functional to procedural language constructs**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Language](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#language) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-functional-to-procedural-language-constructs)

They are usually shorter and come more natural to modern programmers.

DATA(variable) = 'A'.

" MOVE 'A' TO variable.

DATA(uppercase) = to\_upper( lowercase ).

" TRANSLATE lowercase TO UPPER CASE.

index += 1. " >= NW 7.54

index = index + 1. " < NW 7.54

" ADD 1 TO index.

DATA(object) = NEW /clean/my\_class( ).

" CREATE OBJECT object TYPE /dirty/my\_class.

result = VALUE #( FOR row IN input ( row-text ) ).

" LOOP AT input INTO DATA(row).

" INSERT row-text INTO TABLE result.

" ENDLOOP.

DATA(line) = value\_pairs[ name = 'A' ].

" READ TABLE value\_pairs INTO DATA(line) WITH KEY name = 'A'.

DATA(exists) = xsdbool( line\_exists( value\_pairs[ name = 'A' ] ) ).

IF line\_exists( value\_pairs[ name = 'A' ] ).

" READ TABLE value\_pairs TRANSPORTING NO FIELDS WITH KEY name = 'A'.

" DATA(exists) = xsdbool( sy-subrc = 0 ).

Many of the detailed rules below are just specific reiterations of this general advice.

**Avoid obsolete language elements**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Language](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#language) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-obsolete-language-elements)

When upgrading your ABAP version, make sure to check for obsolete language elements and refrain from using them.

For example, the @-escaped "host" variables in the following statement make a little clearer what's a program variable and what's a column in the database,

SELECT \*

FROM spfli

WHERE carrid = @carrid AND

connid = @connid

INTO TABLE @itab.

as compared to the [obsolete unescaped form](https://help.sap.com/doc/abapdocu_750_index_htm/7.50/en-US/abenopen_sql_hostvar_obsolete.htm)

SELECT \*

FROM spfli

WHERE carrid = carrid AND

connid = connid

INTO TABLE itab.

Newer alternatives tend to improve readability of the code, and reduce design conflicts with modern programming paradigms, such that switching to them can automatically clean your code.

While continuing to work, obsolete elements may stop benefitting from optimizations in terms of processing speed and memory consumption.

With modern language elements, you can onboard young ABAPers easier, who may no longer be familiar with the outdated constructs because they are no longer taught in SAP's trainings.

The SAP NetWeaver documentation contains a stable section that lists obsolete language elements, for example [NW 7.50](https://help.sap.com/doc/abapdocu_750_index_htm/7.50/en-US/index.htm?file=abenabap_obsolete.htm),[NW 7.51](https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/index.htm?file=abenabap_obsolete.htm), [NW 7.52](https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-US/index.htm?file=abenabap_obsolete.htm), [NW 7.53](https://help.sap.com/doc/abapdocu_753_index_htm/7.53/en-US/index.htm?file=abenabap_obsolete.htm).

**Use design patterns wisely**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Language](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#language) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-design-patterns-wisely)

Where they are appropriate and provide noticeable benefit. Don't apply design patterns everywhere just for the sake of it.

**Constants**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constants)

**Use constants instead of magic numbers**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Constants](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constants) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-constants-instead-of-magic-numbers)

IF abap\_type = cl\_abap\_typedescr=>typekind\_date.

is clearer than

" anti-pattern

IF abap\_type = 'D'.

Read more in *Chapter 17: Smells and Heuristics: G25: Replace Magic Numbers with Named Constants* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Prefer enumeration classes to constants interfaces**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Constants](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constants) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-enumeration-classes-to-constants-interfaces)

CLASS /clean/message\_severity DEFINITION PUBLIC ABSTRACT FINAL.

PUBLIC SECTION.

CONSTANTS:

warning TYPE symsgty VALUE 'W',

error TYPE symsgty VALUE 'E'.

ENDCLASS.

or

CLASS /clean/message\_severity DEFINITION PUBLIC CREATE PRIVATE FINAL.

PUBLIC SECTION.

CLASS-DATA:

warning TYPE REF TO /clean/message\_severity READ-ONLY,

error TYPE REF TO /clean/message\_severity READ-ONLY.

" ...

ENDCLASS.

instead of mixing unrelated things

" anti-pattern

INTERFACE /dirty/common\_constants.

CONSTANTS:

warning TYPE symsgty VALUE 'W',

transitional TYPE i VALUE 1,

error TYPE symsgty VALUE 'E',

persisted TYPE i VALUE 2.

ENDINTERFACE.

[Enumerations](https://github.com/SAP/styleguides/blob/master/clean-abap/sub-sections/Enumerations.md) describes common enumeration patterns and discusses their advantages and disadvantages.

Read more in *Chapter 17: Smells and Heuristics: J3: Constants versus Enums* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

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

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Constants](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constants) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#if-you-dont-use-enumeration-classes-group-your-constants)

If you collect constants in a loose way, for example in an interface, group them:

CONSTANTS:

BEGIN OF message\_severity,

warning TYPE symsgty VALUE 'W',

error TYPE symsgty VALUE 'E',

END OF message\_severity,

BEGIN OF message\_lifespan,

transitional TYPE i VALUE 1,

persisted TYPE i VALUE 2,

END OF message\_lifespan.

Makes the relation clearer than:

CONSTANTS:

warning TYPE symsgty VALUE 'W',

transitional TYPE i VALUE 1,

error TYPE symsgty VALUE 'E',

persisted TYPE i VALUE 2,

The group also allows you group-wise access, for example for input validation:

DO number\_of\_constants TIMES.

ASSIGN COMPONENT sy-index OF STRUCTURE message\_severity TO FIELD-SYMBOL(<constant>).

IF <constant> = input.

is\_valid = abap\_true.

RETURN.

ENDIF.

ENDWHILE.

Read more in *Chapter 17: Smells and Heuristics: G27: Structure over Convention* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Variables**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#variables)

**Prefer inline to up-front declarations**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Variables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#variables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-inline-to-up-front-declarations)

If you follow these guidelines, your methods will become so short (3-5 statements) that declaring variables inline at first occurrence will look more natural

METHOD do\_something.

DATA(name) = 'something'.

DATA(reader) = /clean/reader=>get\_instance\_for( name ).

result = reader->read\_it( ).

ENDMETHOD.

than declaring variables with a separate DATA section at the beginning of the method

" anti-pattern

METHOD do\_something.

DATA:

name TYPE seoclsname,

reader TYPE REF TO /dirty/reader.

name = 'something'.

reader = /dirty/reader=>get\_instance\_for( name ).

result = reader->read\_it( ).

ENDMETHOD.

Read more in *Chapter 5: Formatting: Vertical Distance: Variable Declarations* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Don't declare inline in optional branches**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Variables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#variables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-declare-inline-in-optional-branches)

" anti-pattern

IF has\_entries = abap\_true.

DATA(value) = 1.

ELSE.

value = 2.

ENDIF.

This works fine because ABAP handles inline declarations as if they were at the beginning of the method. However, it is extremely confusing for readers, especially if the method is longer and you don't spot the declaration right away. In this case, break with inlining and put the declaration up-front:

DATA value TYPE i.

IF has\_entries = abap\_true.

value = 1.

ELSE.

value = 2.

ENDIF.

Read more in *Chapter 5: Formatting: Vertical Distance: Variable Declarations* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Do not chain up-front declarations**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Variables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#variables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#do-not-chain-up-front-declarations)

DATA name TYPE seoclsname

DATA reader TYPE REF TO /dirty/reader.

Chaining suggests the defined variables are related on a logical level. To consistently use it, you would have to ensure that all chained variables belong together, and introduce additional chain groups to add variables. While this is possible, it is usually not worth the effort.

Chaining also needlessly complicates reformatting and refactoring because each line looks different and changing them requires meddling with colons, dots, and commas, that are not worth the effort.

" anti-pattern

DATA:

name TYPE seoclsname,

reader TYPE REF TO /dirty/reader.

Also refer to [Don't align type clauses](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-align-type-clauses)

**Prefer REF TO to FIELD-SYMBOL**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Variables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#variables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-ref-to-to-field-symbol)

LOOP AT components REFERENCE INTO DATA(component).

instead of the equivalent

" anti-pattern

LOOP AT components ASSIGNING FIELD-SYMBOL(<component>).

except where you need field symbols

ASSIGN generic->\* TO FIELD-SYMBOL(<generic>).

ASSIGN COMPONENT name OF STRUCTURE structure TO FIELD-SYMBOL(<component>).

ASSIGN (class\_name)=>(static\_member) TO FIELD-SYMBOL(<member>).

Code reviews demonstrate that people tend to choose between the two arbitrarily, "just because", "because we are always LOOPing that way", or "for no special reason". Arbitrary choices make the reader waste time on the pointless question why one is used over the other and thus should be replaced with well-founded, precise decisions. Our recommendation is based on this reasoning:

* Field symbols can do some things that references cannot, such as dynamically accessing the components of a structure. Likewise, references can do things that field symbols can't, such as constructing a dynamically typed data structure. In summary, settling for one alone is not possible.
* In object-oriented ABAP, references are all over the place and cannot be avoided, as any object is a REF TO <class-name>. In contrast, field symbols are only strictly required in few, special cases concerned with dynamic typing. References thus form a natural preference in any object-oriented program.
* Field symbols are shorter than references, but the resulting memory saving is so tiny that it can be safely neglected. Similarly, speed is not an issue. As a consequence, there is no performance-related reason to prefer one to the other.

Read more in the article [*Accessing Data Objects Dynamically* in the ABAP Programming Guidelines](https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/index.htm?file=abendyn_access_data_obj_guidl.htm).

**Tables**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#tables)

**Use the right table type**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Tables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#tables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-the-right-table-type)

* You typically use HASHED tables for **large tables** that are **filled in a single step**, **never modified**, and **read often by their key**. Their inherent memory and processing overhead makes hash tables only valuable for large amounts of data and lots of read accesses. Each change to the table's content requires expensive recalculation of the hash, so don't use this for tables that are modified too often.
* You typically use SORTED tables for **large tables** that need to be **sorted at all times**, that are **filled bit by bit** or **need to be modified**, and **read often by one or more full or partial keys** or processed **in a certain order**. Adding, changing, or removing content requires finding the right insertion spot, but doesn't require adjusting the rest of the table's index. Sorted tables demonstrate their value only for large numbers of read accesses.
* Use STANDARD tables for **small tables**, where indexing produces more overhead than benefit, and **"arrays"**, where you either don't care at all for the order of the rows, or you want to process them in exactly the order they were appended.

These are only rough guidelines. Find more details in the article [*Selection of Table Category* in the ABAP Language Help](https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/abenitab_kind.htm).

**Avoid DEFAULT KEY**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Tables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#tables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-default-key)

" anti-pattern

DATA itab TYPE STANDARD TABLE OF row\_type WITH DEFAULT KEY.

Default keys are often only added to get the newer functional statements working. The keys themselves in fact are usually superfluous and waste resources for nothing. They can even lead to obscure mistakes because they ignore numeric data types.

Either specify the key components explicitly

DATA itab2 TYPE STANDARD TABLE OF row\_type WITH NON-UNIQUE KEY comp1 comp2.

or resort to EMPTY KEY if you don't need a key at all.

DATA itab1 TYPE STANDARD TABLE OF row\_type WITH EMPTY KEY.

Following [Horst Keller's blog on *Internal Tables with Empty Key*](https://blogs.sap.com/2013/06/27/abap-news-for-release-740-internal-tables-with-empty-key/)

**Prefer INSERT INTO TABLE to APPEND TO**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Tables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#tables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-insert-into-table-to-append-to)

INSERT VALUE #( ... ) INTO TABLE itab.

INSERT INTO TABLE works with all table and key types, thus making it easier for you to refactor the table's type and key definitions if your performance requirements change.

Use APPEND TO only if you use a STANDARD table in an array-like fashion, if you want to stress that the added entry shall be the last row.

**Prefer LINE\_EXISTS to READ TABLE or LOOP AT**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Tables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#tables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-line_exists-to-read-table-or-loop-at)

IF line\_exists( my\_table[ key = 'A' ] ).

expresses the intent clearer and shorter than

" anti-pattern

READ TABLE my\_table TRANSPORTING NO FIELDS WITH KEY key = 'A'.

IF sy-subrc = 0.

or even

" anti-pattern

LOOP AT my\_table REFERENCE INTO DATA(line) WHERE key = 'A'.

line\_exists = abap\_true.

EXIT.

ENDLOOP.

**Prefer READ TABLE to LOOP AT**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Tables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#tables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-read-table-to-loop-at)

READ TABLE my\_table REFERENCE INTO DATA(line) WITH KEY key = 'A'.

expresses the intent clearer and shorter than

" anti-pattern

LOOP AT my\_table REFERENCE INTO DATA(line) WHERE key = 'A'.

EXIT.

ENDLOOP.

or even

" anti-pattern

LOOP AT my\_table REFERENCE INTO DATA(line).

IF line->key = 'A'.

EXIT.

ENDIF.

ENDLOOP.

**Prefer LOOP AT WHERE to nested IF**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Tables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#tables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-loop-at-where-to-nested-if)

LOOP AT my\_table REFERENCE INTO DATA(line) WHERE key = 'A'.

expresses the intent clearer and shorter than

LOOP AT my\_table REFERENCE INTO DATA(line).

IF line->key = 'A'.

EXIT.

ENDIF.

ENDLOOP.

**Avoid unnecessary table reads**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Tables](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#tables) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-unnecessary-table-reads)

In case you *expect* a row to be there, read once and react to the exception,

TRY.

DATA(row) = my\_table[ key = input ].

CATCH cx\_sy\_itab\_line\_not\_found.

RAISE EXCEPTION NEW /clean/my\_data\_not\_found( ).

ENDTRY.

instead of littering and slowing down the main control flow with a double read

" anti-pattern

IF NOT line\_exists( my\_table[ key = input ] ).

RAISE EXCEPTION NEW /clean/my\_data\_not\_found( ).

ENDTRY.

DATA(row) = my\_table[ key = input ].

Besides being a performance improvement, this is a special variant of the more general [Focus on the happy path or error handling, but not both](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#focus-on-the-happy-path-or-error-handling-but-not-both).

**Strings**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#strings)

**Use ` to define literals**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Strings](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#strings) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use--to-define-literals)

CONSTANTS some\_constant TYPE string VALUE `ABC`.

DATA(some\_string) = `ABC`. " --> TYPE string

Refrain from using ', as it adds a superfluous type conversion and confuses the reader whether he's dealing with a CHARor STRING:

" anti-pattern

DATA some\_string TYPE string.

some\_string = 'ABC'.

| is generally okay, but cannot be used for CONSTANTS and adds needless overhead when specifying a fixed value:

" anti-pattern

DATA(some\_string) = |ABC|.

**Use | to assemble text**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Strings](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#strings) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use--to-assemble-text)

DATA(message) = |Received HTTP code { status\_code } with message { text }|.

String templates highlight better what's literal and what's variable, especially if you embed multiple variables in a text.

" anti-pattern

DATA(message) = `Received an unexpected HTTP ` && status\_code && ` with message ` && text.

**Booleans**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#booleans)

**Use Booleans wisely**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Booleans](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#booleans) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-booleans-wisely)

We often encounter cases where Booleans seem to be a natural choice

" anti-pattern

is\_archived = abap\_true.

until a change of viewpoint suggests we should have chosen an enumeration

archiving\_status = /clean/archivation\_status=>archiving\_in\_process.

Generally, Booleans are a bad choice to distinguish types of things because you will nearly always encounter cases that are not exclusively one or the other

assert\_true( xsdbool( document->is\_archived( ) = abap\_true AND

document->is\_partially\_archived( ) = abap\_true ) ).

[Split method instead of Boolean input parameter](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#split-method-instead-of-boolean-input-parameter) moreover explains why you should always challenge Boolean parameters.

Read more in [1](http://www.beyondcode.org/articles/booleanVariables.html)

**Use ABAP\_BOOL for Booleans**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Booleans](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#booleans) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-abap_bool-for-booleans)

DATA has\_entries TYPE abap\_bool.

Don't use the generic type char1. Although it is technically compatible it obscures the fact that we're dealing with a Boolean variable.

Also avoid other Boolean types as they often have strange side effects, for example boolean supports a third value "undefined" that results in subtle programming errors.

In some cases you may need a data dictionary element, for example for DynPro fields. abap\_bool cannot be used here because it is defined in the type pool abap, not in the data dictionary. In this case, resort to boole\_d or xfeld. Create your own data element if you need a custom description.

ABAP may be the one single programming language that does not come with a universal Boolean data type. However, having one is imperative. This recommendation is based on the ABAP Programming Guidelines.

**Use ABAP\_TRUE and ABAP\_FALSE for comparisons**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Booleans](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#booleans) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-abap_true-and-abap_false-for-comparisons)

has\_entries = abap\_true.

IF has\_entries = abap\_false.

Don't use the character equivalents 'X' and ' ' or space; they make it hard to see that this is a Boolean expression:

" anti-pattern

has\_entries = 'X'.

IF has\_entries = space.

Avoid comparisons with INITIAL - it forces readers to recollect that abap\_bool's default is abap\_false:

" anti-pattern

IF has\_entries IS NOT INITIAL.

ABAP may be the one single programming language that does not come with built-in "constants" for true and false. However, having them is imperative. This recommendation is based on the ABAP Programming Guidelines.

**Use XSDBOOL to set Boolean variables**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Booleans](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#booleans) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-xsdbool-to-set-boolean-variables)

DATA(has\_entries) = xsdbool( line IS NOT INITIAL ).

The equivalent IF-THEN-ELSE is much longer for nothing:

" anti-pattern

IF line IS INITIAL.

has\_entries = abap\_false.

ELSE.

has\_entries = abap\_true.

ENDIF.

xsdbool is the best method for our purpose, as it directly produces a char1, which fits our boolean type abap\_bool best. The equivalent functions boolc and boolx produce different types and add an unnecessary implicit type conversion.

We agree that the name xsdbool is unlucky and misleading; after all, we're not at all interested in the "XML Schema Definition" parts that the "xsd" prefix suggests.

A possible alternative to xsdbool is the COND ternary form. Its syntax is intuitive, but a little longer because it needlessly repeats the THEN abap\_true segment, and requires knowledge of the implicit default value abap\_false - which is why we suggest it only as secondary solution.

DATA(has\_entries) = COND abap\_bool( WHEN line IS NOT INITIAL THEN abap\_true ).

**Conditions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#conditions)

**Try to make conditions positive**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#conditions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#try-to-make-conditions-positive)

IF has\_entries = abap\_true.

For comparison, see how hard to understand the same statement gets by reversing it:

" anti-pattern

IF has\_no\_entries = abap\_false.

The "try" in the section title means you shouldn't force this up to the point where you end up with something like [empty IF branches](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#no-empty-if-branches):

" anti-pattern

IF has\_entries = abap\_true.

ELSE.

" only do something in the ELSE block, IF remains empty

ENDIF.

Read more in *Chapter 17: Smells and Heuristics: G29: Avoid Negative Conditionals* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Prefer IS NOT to NOT IS**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#conditions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-is-not-to-not-is)

IF variable IS NOT INITIAL.

IF variable NP 'TODO\*'.

IF variable <> 42.

Negation is logically equivalent but requires a "mental turnaround" that makes it harder to understand.

" anti-pattern

IF NOT variable IS INITIAL.

IF NOT variable CP 'TODO\*'.

IF NOT variable = 42.

A more specific variant of [Try to make conditions positive](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#try-to-make-conditions-positive). Also as described in the section [Alternative Language Constructs](https://help.sap.com/doc/abapdocu_753_index_htm/7.53/en-US/index.htm?file=abenalternative_langu_guidl.htm) in the ABAP programming guidelines.

**Consider decomposing complex conditions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#conditions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-decomposing-complex-conditions)

Conditions can become easier when decomposing them into the elementary parts that make them up:

DATA(example\_provided) = xsdbool( example\_a IS NOT INITIAL OR

example\_b IS NOT INITIAL ).

DATA(one\_example\_fits) = xsdbool( applies( example\_a ) = abap\_true OR

applies( example\_b ) = abap\_true OR

fits( example\_b ) = abap\_true ).

IF example\_provided = abap\_true AND

one\_example\_fits = abap\_true.

instead of leaving everything in-place:

" anti-pattern

IF ( example\_a IS NOT INITIAL OR

example\_b IS NOT INITIAL ) AND

( applies( example\_a ) = abap\_true OR

applies( example\_b ) = abap\_true OR

fits( example\_b ) = abap\_true ).

Use the ABAP Development Tools quick fixes to quickly extract conditions and create variables as shown above.

**Consider extracting complex conditions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Conditions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#conditions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-extracting-complex-conditions)

It's nearly always a good idea to extract complex conditions to methods of their own:

IF is\_provided( example ).

METHOD is\_provided.

DATA(is\_filled) = xsdbool( example IS NOT INITIAL ).

DATA(is\_working) = xsdbool( applies( example ) = abap\_true OR

fits( example ) = abap\_true ).

result = xsdbool( is\_filled = abap\_true AND

is\_working = abap\_true ).

ENDMETHOD.

**Ifs**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#ifs)

**No empty IF branches**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Ifs](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#ifs) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#no-empty-if-branches)

IF has\_entries = abap\_false.

" do some magic

ENDIF.

is shorter and clearer than

" anti-pattern

IF has\_entries = abap\_true.

ELSE.

" do some magic

ENDIF.

**Prefer CASE to ELSE IF for multiple alternative conditions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Ifs](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#ifs) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-case-to-else-if-for-multiple-alternative-conditions)

CASE type.

WHEN type-some\_type.

" ...

WHEN type-some\_other\_type.

" ...

WHEN OTHERS.

RAISE EXCEPTION NEW /clean/unknown\_type\_failure( ).

ENDCASE.

CASE makes it easy to see a set of alternatives that exclude each other. It can be faster than a series of IFs because it can translate to a different microprocessor command instead of a series of subsequently evaluated conditions. You can introduce new cases quickly, without having to repeat the discerning variable over and over again. The statement even prevents some errors that can occur when accidentally nesting the IF-ELSEIFs.

" anti-pattern

IF type = type-some\_type.

" ...

ELSEIF type = type-some\_other\_type.

" ...

ELSE.

RAISE EXCEPTION NEW /dirty/unknown\_type\_failure( ).

ENDIF.

**Keep the nesting depth low**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Ifs](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#ifs) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-the-nesting-depth-low)

" ani-pattern

IF <this>.

IF <that>.

ENDIF.

ELSE.

IF <other>.

ELSE.

IF <something>.

ENDIF.

ENDIF.

ENDIF.

Nested IFs get hard to understand very quickly and require an exponential number of test cases for complete coverage.

Decision trees can usually be taken apart by forming sub-methods and introducing boolean helper variables.

Other cases can be simplified by merging IFs, such as

IF <this> AND <that>.

instead of the needlessly nested

" anti-pattern

IF <this>.

IF <that>.

**Regular expressions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#regular-expressions)

**Prefer simpler methods to regular expressions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Regular expressions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#regular-expressions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-simpler-methods-to-regular-expressions)

IF input IS NOT INITIAL.

" IF matches( val = input regex = '.+' ).

WHILE contains( val = input sub = 'abc' ).

" WHILE contains( val = input regex = 'abc' ).

Regular expressions become hard to understand very quickly. Simple cases are usually easier without them.

Regular expressions also usually consume more memory and processing time because they need to be parsed into an expression tree and compiled at runtime into an executable matcher. Simple solutions may do with a straight-forward loop and a temporary variable.

**Prefer basis checks to regular expressions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Regular expressions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#regular-expressions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-basis-checks-to-regular-expressions)

CALL FUNCTION 'SEO\_CLIF\_CHECK\_NAME'

EXPORTING

cls\_name = class\_name

EXCEPTIONS

...

instead of reinventing things

" anti-pattern

DATA(is\_valid) = matches( val = class\_name

pattern = '[A-Z][A-Z0-9\_]{0,29}' ).

There seems to be a natural tendency to turn blind to the Don't-Repeat-Yourself (DRY) principle when there are regular expressions around, compare section *Chapter 17: Smells and Heuristics: General: G5: Duplication* in [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Consider assembling complex regular expressions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Regular expressions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#regular-expressions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-assembling-complex-regular-expressions)

CONSTANTS class\_name TYPE string VALUE `CL\\_.\*`.

CONSTANTS interface\_name TYPE string VALUE `IF\\_.\*`.

DATA(object\_name) = |{ class\_name }\|{ interface\_name }|.

Some complex regular expressions become easier when you demonstrate to the reader how they are built up from more elementary pieces.

**Classes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes)

**Classes: Object orientation**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes-object-orientation)

**Prefer objects to static classes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Classes: Object orientation](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes-object-orientation) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-objects-to-static-classes)

Static classes give up all advantages gained by object orientation in the first place. They especially make it nearly impossible to replace productive dependencies with test doubles in unit tests.

If you think about whether to make a class or method static, the answer will nearly always be: no.

One accepted exception to this rule are plain type utils classes. Their methods make it easier to interact with certain ABAP types. They are not only completely stateless, but so basic that they look like ABAP statements or built-in functions. The discriminating factor is that their consumers tie them into their code so tightly that they actually don't want to mock them in unit tests.

CLASS /clean/string\_utils DEFINITION [...].

CLASS-METHODS trim

IMPORTING

string TYPE string

RETURNING

VALUE(result) TYPE string.

ENDCLASS.

METHOD retrieve.

DATA(trimmed\_name) = /clean/string\_utils=>trim( name ).

result = read( trimmed\_name ).

ENDMETHOD.

**Prefer composition to inheritance**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Classes: Object orientation](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes-object-orientation) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-composition-to-inheritance)

Avoid building hierarchies of classes with inheritance. Instead, favor composition.

Clean inheritance is hard to design because you need to respect rules like the [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle). It is also hard to understand because people need to realize and digest the guiding principles behind the hierarchy. Inheritance reduces reuse because methods tend to be made available only to sub-classes. It also complicates refactoring because moving or changing members tend to require changes to the whole hierarchy tree.

Composition means that you design small, independent objects, each of which serves one specific purpose. These objects can be recombined into more complex objects by simple delegation and facade patterns. Composition may produce more classes, but has otherwise no further disadvantages.

Don't let this rule discourage you from using inheritance in the right places. There are good applications for inheritance, for example the [Composite design pattern](https://en.wikipedia.org/wiki/Composite_pattern). Just ask yourself critically whether inheritance in your case will really provide more benefits than disadvantages. If in doubt, composition generally is the safer choice.

**Don't mix stateful and stateless in the same class**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Classes: Object orientation](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes-object-orientation)

Don't mix the stateless and the stateful programming paradigms in the same class.

In stateless programming, methods get input and produce output, *without any side effects*, resulting in methods that produce the same result no matter when and in what order they are called.

CLASS /clean/xml\_converter DEFINITION PUBLIC FINAL CREATE PUBLIC.

PUBLIC SECTION.

METHODS convert

IMPORTING

file\_content TYPE xstring

RETURNING

VALUE(result) TYPE /clean/some\_inbound\_message.

ENDCLASS.

CLASS /clean/xml\_converter IMPLEMENTATION.

METHOD convert.

cl\_proxy\_xml\_transform=>xml\_xstring\_to\_abap(

EXPORTING

xml = file\_content

ext\_xml = abap\_true

svar\_name = 'ROOT\_NODE'

IMPORTING

abap\_data = result ).

ENDMETHOD.

ENDCLASS.

In stateful programming, we manipulate the internal state of objects through their methods, meaning it is *full of side effects*.

CLASS /clean/log DEFINITION PUBLIC CREATE PUBLIC.

PUBLIC SECTION.

METHODS add\_message IMPORTING message TYPE /clean/message.

PRIVATE SECTION.

DATA messages TYPE /clean/message\_table.

ENDCLASS.

CLASS /clean/log IMPLEMENTATION.

METHOD add\_message.

INSERT message INTO TABLE messages.

ENDMETHOD.

ENDCLASS.

Both paradigms are okay and have their applications. However, *mixing* them in the same object produces code that is hard to understand and sure to fail with obscure carry-over errors and synchronicity problems. Don't do that.

**Scope**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#scope)

**Global by default, local only in exceptional cases**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Scope](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#scope) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#global-by-default-local-only-in-exceptional-cases)

Work with global classes as default (meaning the ones that are visible in the dictionary).

Use local classes only in exceptional cases, for example for very specific data structures or to facilitate writing unit tests.

Local classes hinder reuse because they cannot be used elsewhere. Although they are easy to extract, people will usually fail to even find them.

A clear indication that a local class should be made global is if you have the urge to write tests for it. A local class is a natural private cogwheel in its greater global class that you will usually not test. The need for tests indicates that the class is independent from its surrounding and so complex that it should become an object of its own.

**FINAL if not designed for inheritance**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Scope](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#scope) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#final-if-not-designed-for-inheritance)

Make classes that are not explicitly designed for inheritance FINAL.

When designing class cooperation, your first choice should be [composition, not inheritance](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-composition-to-inheritance). Enabling inheritance is not something that should be done lightly, as it requires you to think about things like PROTECTED vs. PRIVATE and the [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle), and freezes a lot of design internals. If you didn't consider these things in your class design, you should thus prevent accidental inheritance by making your class FINAL.

There *are* some good applications for inheritance, of course, for example the design pattern [composite](https://en.wikipedia.org/wiki/Composite_pattern). Business Add-Ins can also become more useful by allowing sub-classes, enabling the customer to reuse most of the original code. However, note that all of these cases have inheritance built in by design from the start.

Unclean classes that don't [implement interfaces](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#public-instance-methods-should-be-part-of-an-interface) should be left non-FINAL to allow consumers mocking them in their unit tests.

**Members PRIVATE by default, PROTECTED only if needed**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Scope](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#scope) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#members-private-by-default-protected-only-if-needed)

Make attributes, methods, and other class members PRIVATE by default.

Make them only PROTECTED if you want to enable sub-classes that override them.

Internals of classes should be made available to others only on a need-to-know basis. This includes not only outside callers but also sub-classes. Making information over-available can cause subtle errors by unexpected redefinitions and hinder refactoring because outsiders freeze members in place that should still be liquid.

**Consider using immutable instead of getter**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Scope](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#scope) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-using-immutable-instead-of-getter)

An immutable is an object that never changes after its construction. For this kind of object consider using public read-only attributes instead of getter methods.

CLASS /clean/some\_data\_container DEFINITION.

PUBLIC SECTION.

METHODS constructor

IMPORTING

a TYPE i

b TYPE c

c TYPE d.

DATA a TYPE i READ-ONLY.

DATA b TYPE c READ-ONLY.

DATA c TYPE d READ-ONLY.

ENDCLASS.

instead of

CLASS /dirty/some\_data\_container DEFINITION.

PUBLIC SECTION.

METHODS get\_a ...

METHODS get\_b ...

METHODS get\_c ...

PRIVATE SECTION.

DATA a TYPE i.

DATA b TYPE c.

DATA c TYPE d.

ENDCLASS.

**Caution**: For objects which **do** have changing values, do not use public read-only attributes. Otherwise this attributes always have to be kept up to date, regardless if their value is needed by any other code or not.

**Use READ-ONLY sparingly**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Scope](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#scope) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-read-only-sparingly)

Many modern programming languages, especially Java, recommend making class members read-only wherever appropriate to prevent accidental side effects.

While ABAP *does* offer the READ-ONLY addition for data declarations, we recommend to use it sparingly.

First, the addition is only available in the PUBLIC SECTION, reducing its applicability drastically. You can neither add it to protected or private members nor to local variables in a method.

Second, the addition works subtly different from what people might expect from other programming languages: READ-ONLY data can still be modified freely from any method within the class itself, its friends, and its sub-classes. This contradicts the more widespread write-once-modify-never behavior found in other languages. The difference may lead to bad surprises.

To avoid misunderstandings: Protecting variables against accidental modification is a good practice. We would recommend applying it to ABAP as well if there was an appropriate statement.

**Constructors**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constructors)

**Prefer NEW to CREATE OBJECT**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Constructors](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constructors) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-new-to-create-object)

DATA object TYPE REF TO /clean/some\_number\_range.

object = NEW #( '/CLEAN/CXTGEN' )

...

DATA(object) = NEW /clean/some\_number\_range( '/CLEAN/CXTGEN' ).

...

DATA(object) = CAST /clean/number\_range( NEW /clean/some\_number\_range( '/CLEAN/CXTGEN' ) ).

instead of the needlessly longer

" anti-pattern

DATA object TYPE REF TO /dirty/some\_number\_range.

CREATE OBJECT object

EXPORTING

number\_range = '/DIRTY/CXTGEN'.

except where you need dynamic types, of course

CREATE OBJECT number\_range TYPE (dynamic\_type)

EXPORTING

number\_range = '/CLEAN/CXTGEN'.

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

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Constructors](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constructors) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#if-your-global-class-is-create-private-leave-the-constructor-public)

CLASS /clean/some\_api DEFINITION PUBLIC FINAL CREATE PRIVATE.

PUBLIC SECTION.

METHODS constructor.

We agree that this contradicts itself. However, according to the article [*Instance Constructor* of the ABAP Help](https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/abeninstance_constructor_guidl.htm), specifying the CONSTRUCTOR in the PUBLIC SECTION is required to guarantee correct compilation and syntax validation.

This applies only to global classes. In local classes, make the constructor private, as it should be.

**Prefer multiple static factory methods to optional parameters**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Constructors](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constructors) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-multiple-static-factory-methods-to-optional-parameters)

CLASS-METHODS describe\_by\_data IMPORTING data TYPE any [...]

CLASS-METHODS describe\_by\_name IMPORTING name TYPE any [...]

CLASS-METHODS describe\_by\_object\_ref IMPORTING object\_ref TYPE REF TO object [...]

CLASS-METHODS describe\_by\_data\_ref IMPORTING data\_ref TYPE REF TO data [...]

Don't try to "remedy" ABAP's missing support for [overloading](https://en.wikipedia.org/wiki/Function_overloading) by adding optional parameters.

" anti-pattern

METHODS constructor

IMPORTING

data TYPE any OPTIONAL

name TYPE any OPTIONAL

object\_ref TYPE REF TO object OPTIONAL

data\_ref TYPE REF TO data OPTIONAL

[...]

The general guideline [*Split methods instead of adding OPTIONAL parameters*](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#split-methods-instead-of-adding-optional-parameters) explains the reasoning behind this.

Consider resolving complex constructions to a multi-step construction with the [Builder design pattern](https://en.wikipedia.org/wiki/Builder_pattern).

**Use descriptive names for multiple constructor methods**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Constructors](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constructors) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-descriptive-names-for-multiple-constructor-methods)

CLASS-METHODS describe\_by\_data IMPORTING p\_data TYPE any [...]

CLASS-METHODS describe\_by\_name IMPORTING p\_name TYPE any [...]

CLASS-METHODS describe\_by\_object\_ref IMPORTING p\_object\_ref TYPE REF TO object [...]

CLASS-METHODS describe\_by\_data\_ref IMPORTING p\_data\_ref TYPE REF TO data [...]

instead of something meaningless like

" anti-pattern

CLASS-METHODS create\_1 IMPORTING p\_data TYPE any [...]

CLASS-METHODS create\_2 IMPORTING p\_name TYPE any [...]

CLASS-METHODS create\_3 IMPORTING p\_object\_ref TYPE REF TO object [...]

CLASS-METHODS create\_4 IMPORTING p\_data\_ref TYPE REF TO data [...]

Good words to start constructors are new\_, create\_, and construct\_. People intuitively connect them to the construction of objects. They also add up nicely to verb phrases like new\_from\_template, create\_as\_copy, or create\_by\_name.

**Make singletons only where multiple instances don't make sense**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#classes) > [Constructors](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#constructors) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#make-singletons-only-where-multiple-instances-dont-make-sense)

METHOD new.

IF singleton IS NOT BOUND.

singleton = NEW /clean/my\_class( ).

ENDIF.

result = singleton.

ENDMETHOD.

Apply the singleton pattern where your object-oriented design says that having a second instance of something doesn't make sense. Use it to ensure that every consumer is working with the same thing in the same state and with the same data.

Do not use the singleton pattern out of habit or because some performance rule tells you so. It is the most overused and wrongly applied pattern and produces unexpected cross-effects and needlessly complicates testing. If there are no design-driven reasons for a unitary object, leave that decision to the consumer - he can still reach the same by means outside the constructor, for example with a factory.

**Methods**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods)

These rules apply to methods in classes and function modules.

**Calls**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#calls)

**Prefer functional to procedural calls**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Calls](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#calls) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-functional-to-procedural-calls)

modify->update( node = /clean/my\_bo\_c=>node-item

key = item->key

data = item

changed\_fields = changed\_fields ).

instead of the needlessly longer

" anti-pattern

CALL METHOD modify->update

EXPORTING

node = /dirty/my\_bo\_c=>node-item

key = item->key

data = item

changed\_fields = changed\_fields.

If dynamic typing forbids functional calls, resort to the procedural style

CALL METHOD modify->(method\_name)

EXPORTING

node = /clean/my\_bo\_c=>node-item

key = item->key

data = item

changed\_fields = changed\_fields.

Many of the detailed rules below are just more specific variations of this advice.

**Omit RECEIVING**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Calls](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#calls) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#omit-receiving)

DATA(sum) = aggregate\_values( values ).

instead of the needlessly longer

" anti-pattern

aggregate\_values(

EXPORTING

values = values

RECEIVING

result = DATA(sum) ).

**Omit the optional keyword EXPORTING**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Calls](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#calls) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#omit-the-optional-keyword-exporting)

modify->update( node = /clean/my\_bo\_c=>node-item

key = item->key

data = item

changed\_fields = changed\_fields ).

instead of the needlessly longer

" anti-pattern

modify->update(

EXPORTING

node = /dirty/my\_bo\_c=>node-item

key = item->key

data = item

changed\_fields = changed\_fields ).

**Omit the parameter name in single parameter calls**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Calls](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#calls) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#omit-the-parameter-name-in-single-parameter-calls)

DATA(unique\_list) = remove\_duplicates( list ).

instead of the needlessly longer

" anti-pattern

DATA(unique\_list) = remove\_duplicates( list = list ).

There are cases, however, where the method name alone is not clear enough and repeating the parameter name may further understandability:

car->drive( speed = 50 ).

update( asynchronous = abap\_true ).

**Omit the self-reference me when calling an instance method**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Calls](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#calls) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#omit-the-self-reference-me-when-calling-an-instance-method)

Since the self-reference me-> is implicitly set by the system, omit it when calling an instance method

DATA(sum) = aggregate\_values( values ).

instead of the needlessly longer

" anti-pattern

DATA(sum) = me->aggregate\_values( values ).

**Methods: Object orientation**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods-object-orientation)

**Prefer instance to static methods**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Methods: Object orientation](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods-object-orientation) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-instance-to-static-methods)

Methods should be instance members by default. Instance methods better reflect the "object-hood" of the class. They can be mocked easier in unit tests.

METHODS publish.

Methods should be static only in exceptional cases, such as static constructor methods.

CLASS-METHODS create\_instance

RETURNING

VALUE(result) TYPE REF TO /clean/blog\_post.

**Public instance methods should be part of an interface**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Methods: Object orientation](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods-object-orientation) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#public-instance-methods-should-be-part-of-an-interface)

Public instance methods should always be part of an interface. This decouples dependencies and simplifies mocking them in unit tests.

METHOD /clean/blog\_post~publish.

In clean object orientation, having a method public without an interface does not make much sense - with few exceptions such as enumeration classes which will never have an alternative implementation and will never be mocked in test cases.

**Parameter Number**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-number)

**Aim for few IMPORTING parameters, at best less than three**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Number](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-number) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#aim-for-few-importing-parameters-at-best-less-than-three)

FUNCTION seo\_class\_copy

IMPORTING

clskey TYPE seoclskey

new\_clskey TYPE seoclskey

config TYPE class\_copy\_config

EXPORTING

...

would be much clearer than

" anti-pattern

FUNCTION seo\_class\_copy

IMPORTING

clskey TYPE seoclskey

new\_clskey TYPE seoclskey

access\_permission TYPE seox\_boolean DEFAULT seox\_true

VALUE(save) TYPE seox\_boolean DEFAULT seox\_true

VALUE(suppress\_corr) TYPE seox\_boolean DEFAULT seox\_false

VALUE(suppress\_dialog) TYPE seox\_boolean DEFAULT seox\_false

VALUE(authority\_check) TYPE seox\_boolean DEFAULT seox\_true

lifecycle\_manager TYPE REF TO if\_adt\_lifecycle\_manager OPTIONAL

lock\_handle TYPE REF TO if\_adt\_lock\_handle OPTIONAL

VALUE(suppress\_commit) TYPE seox\_boolean DEFAULT seox\_false

EXPORTING

...

Too many input parameters let the complexity of a method explode because it needs to handle an exponential number of combinations. Many parameters are an indicator that the method may do more than one thing.

You can reduce the number of parameters by combining them into meaningful sets with structures and objects.

**Split methods instead of adding OPTIONAL parameters**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Number](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-number) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#split-methods-instead-of-adding-optional-parameters)

METHODS do\_one\_thing IMPORTING what\_i\_need TYPE string.

METHODS do\_another\_thing IMPORTING something\_else TYPE i.

instead of trying to compensate ABAP's missing support for [overloading](https://en.wikipedia.org/wiki/Function_overloading) by adding optional parameters

" anti-pattern

METHODS do\_one\_or\_the\_other

IMPORTING

what\_i\_need TYPE string OPTIONAL

something\_else TYPE i OPTIONAL.

Optional parameters confuse callers: Which ones are really required? Which combinations are valid? Which ones exclude each other?

Multiple methods avoid this confusion by giving clear guidance which parameter combinations are valid and expected.

**Use PREFERRED PARAMETER sparingly**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Number](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-number) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-preferred-parameter-sparingly)

The addition PREFERRED PARAMETER makes it hard to see which parameter is actually supplied, making it harder to understand the code. Minimizing the number of parameters, especially optional ones, automatically reduces the need for PREFERRED PARAMETER.

**RETURN, EXPORT, or CHANGE exactly one parameter**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Number](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-number) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#return-export-or-change-exactly-one-parameter)

A good method does *one thing*, and that should be reflected by the method also returning exactly one thing. If the output parameters of your method do *not* form a logical entity, your method does more than one thing and you should split it.

There are cases where the output is a logical entity that consists of multiple things. These are easiest represented by returning a structure or object:

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

RETURNING

VALUE(result) TYPE check\_result.

instead of

" anti-pattern

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.

Especially in comparison to multiple EXPORTING parameters, this allows people to use the functional call style, spares you having to think about IS SUPPLIED and saves people from accidentally forgetting to retrieve a vital ERROR\_OCCURREDinformation.

Instead of multiple optional output parameters, consider splitting the method according to meaningful call patterns:

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

IMPORTING

business\_partners TYPE business\_partners

RETURNING

VALUE(result) TYPE result\_type.

METHODS check\_and\_report

IMPORTING

business\_partners TYPE business\_partners

RETURNING

VALUE(result) TYPE check\_result.

**Parameter Types**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-types)

**Prefer RETURNING to EXPORTING**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Types](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-types) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-returning-to-exporting)

METHODS square

IMPORTING

number TYPE i

RETURNING

VALUE(result) TYPE i.

DATA(result) = square( 42 ).

Instead of the needlessly longer

" anti-pattern

METHODS square

IMPORTING

number TYPE i

EXPORTING

result TYPE i.

square(

EXPORTING

number = 42

IMPORTING

result = DATA(result) ).

RETURNING not only makes the call shorter, it also allows method chaining and prevents [same-input-and-output errors](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#take-care-if-input-and-output-could-be-the-same).

**RETURNING large tables is usually okay**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Types](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-types) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#returning-large-tables-is-usually-okay)

Although the ABAP language documentation and performance guides say otherwise, we rarely encounter cases where handing over a large or deeply-nested table in a VALUE parameter *really* causes performance problems. We therefore recommend to generally use

METHODS get\_large\_table

RETURNING

VALUE(result) TYPE /clean/some\_table\_type.

METHOD get\_large\_table.

result = me->large\_table.

ENDMETHOD.

DATA(my\_table) = get\_large\_table( ).

Only if there is actual proof (= a bad performance measurement) for your individual case should you resort to the more cumbersome procedural style

" anti-pattern

METHODS get\_large\_table

EXPORTING

result TYPE /dirty/some\_table\_type.

METHOD get\_large\_table.

result = me->large\_table.

ENDMETHOD.

get\_large\_table( IMPORTING result = DATA(my\_table) ).

This section contradicts the ABAP Programming Guidelines and Code Inspector checks, both of whom suggest that large tables should be EXPORTED by reference to avoid performance deficits. We consistently failed to reproduce any performance and memory deficits and received notice about kernel optimization that generally improves RETURNING performance.

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

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Types](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-types) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-either-returning-or-exporting-or-changing-but-not-a-combination)

METHODS copy\_class

IMPORTING

old\_name TYPE seoclsname

new name TYPE secolsname

RETURNING

VALUE(result) TYPE copy\_result

RAISING

/clean/class\_copy\_failure.

instead of confusing mixtures like

" anti-pattern

METHODS copy\_class

...

RETURNING

VALUE(result) TYPE vseoclass

EXPORTING

error\_occurred TYPE abap\_bool

CHANGING

correction\_request TYPE trkorr

package TYPE devclass.

Different sorts of output parameters is an indicator that the method does more than one thing. It confuses the reader and makes calling the method needlessly complicated.

An acceptable exception to this rule may be builders that consume their input while building their output:

METHODS build\_tree

CHANGING

tokens TYPE tokens

RETURNING

VALUE(result) TYPE REF TO tree.

However, even those can be made clearer by objectifying the input:

METHODS build\_tree

IMPORTING

tokens TYPE REF TO token\_stack

RETURNING

VALUE(result) TYPE REF TO tree.

**Use CHANGING sparingly, where suited**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Types](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-types) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-changing-sparingly-where-suited)

CHANGING should be reserved for cases where an existing local variable that is already filled is updated in only some places:

METHODS update\_references

IMPORTING

new\_reference TYPE /bobf/conf\_key

CHANGING

bo\_nodes TYPE root\_nodes.

METHOD update\_references.

LOOP AT bo\_nodes REFERENCE INTO DATA(bo\_node).

bo\_node->reference = new\_reference.

ENDLOOP.

ENDMETHOD.

Do not force your callers to introduce unnecessary local variables only to supply your CHANGING parameter. Do not use CHANGING parameters to initially fill a previously empty variable.

**Split method instead of Boolean input parameter**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Types](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-types) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#split-method-instead-of-boolean-input-parameter)

Boolean input parameters are often an indicator that a method does *two* things instead of one.

" anti-pattern

METHODS update

IMPORTING

do\_save TYPE abap\_bool.

Also, method calls with a single - and thus unnamed - Boolean parameter tend to obscure the parameter's meaning.

" anti-pattern

update( abap\_true ). " what does 'true' mean? synchronous? simulate? commit?

Splitting the method may simplify the methods' code and describe the different intentions better

update\_without\_saving( ).

update\_and\_save( ).

Common perception suggests that setters for Boolean variables are okay:

METHODS set\_is\_deleted

IMPORTING

new\_value TYPE abap\_bool.

Read more in [1](http://www.beyondcode.org/articles/booleanVariables.html) [2](https://silkandspinach.net/2004/07/15/avoid-boolean-parameters/) [3](http://jlebar.com/2011/12/16/Boolean_parameters_to_API_functions_considered_harmful..html)

**Parameter Names**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-names)

**Consider calling the RETURNING parameter RESULT**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Names](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-names) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-calling-the-returning-parameter-result)

Good method names are usually so good that the RETURNING parameter does not need a name of its own. The name would do little more than parrot the method name or repeat something obvious.

Repeating a member name can even produce conflicts that need to be resolved by adding a superfluous me->.

" anti-pattern

METHODS get\_name

RETURNING

VALUE(name) TYPE string.

METHOD get\_name.

name = me->name.

ENDMETHOD.

In these cases, simply call the parameter RESULT, or something like RV\_RESULT if you prefer Hungarian notation.

Name the RETURNING parameter if it is *not* obvious what it stands for, for example in methods that return me for method chaining, or in methods that create something but don't return the created entity but only its key or so.

**Parameter Initialization**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-initialization)

**Clear or overwrite EXPORTING reference parameters**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Initialization](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-initialization) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clear-or-overwrite-exporting-reference-parameters)

Reference parameters refer to existing memory areas that may be filled beforehand. Clear or overwrite them to provide reliable data:

METHODS square

EXPORTING

result TYPE i.

" clear

METHOD square.

CLEAR result.

" ...

ENDMETHOD.

" overwrite

METHOD square.

result = cl\_abap\_math=>square( 2 ).

ENDMETHOD.

Code inspector and Checkman point out EXPORTING variables that are never written. Use these static checks to avoid this otherwise rather obscure error source.

**Take care if input and output could be the same**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Initialization](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-initialization) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#take-care-if-input-and-output-could-be-the-same)

Generally, it is a good idea to clear the parameter as a first thing in the method after type and data declarations. This makes the statement easy to spot and avoids that the still-contained value is accidentally used by later statements.

However, some parameter configurations could use the same variable as input and output. In this case, an early CLEARwould delete the input value before it can be used, producing wrong results.

" anti-pattern

DATA value TYPE i.

square\_dirty(

EXPORTING

number = value

IMPORTING

result = value ).

METHOD square\_dirty.

CLEAR result.

result = number \* number.

ENDMETHOD.

Consider redesigning such methods by replacing EXPORTING with RETURNING. Also consider overwriting the EXPORTINGparameter in a single result calculation statement. If neither fits, resort to a late CLEAR.

**Don't clear VALUE parameters**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Parameter Initialization](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#parameter-initialization) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-clear-value-parameters)

Parameters that work by VALUE are handed over as new, separate memory areas that are empty by definition. Don't clear them again:

METHODS square

EXPORTING

VALUE(result) TYPE i.

METHOD square.

" no need to CLEAR result

ENDMETHOD.

RETURNING parameters are always VALUE parameters, so you never have to clear them:

METHODS square

RETURNING

VALUE(result) TYPE i.

METHOD square.

" no need to CLEAR result

ENDMETHOD.

**Method Body**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#method-body)

**Do one thing, do it well, do it only**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Method Body](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#method-body) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#do-one-thing-do-it-well-do-it-only)

A method should do one thing, and only one thing. It should do it in the best way possible.

A method likely does one thing if

* it has [few input parameters](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#aim-for-few-importing-parameters-at-best-less-than-three)
* that [don't include Boolean parameters](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#split-method-instead-of-boolean-input-parameter)
* it has [exactly one output parameter](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#return-export-or-change-exactly-one-parameter)
* it is [small](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-methods-small)
* it [descends one level of abstraction](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#descend-one-level-of-abstraction)
* you cannot extract meaningful other methods
* you cannot meaningfully group its statements into sections

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

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Method Body](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#method-body) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#focus-on-the-happy-path-or-error-handling-but-not-both)

As a specialization of the rule [*Do one thing, do it well, do it only*](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#do-one-thing-do-it-well-do-it-only), a method should either follow the happy-path it's built for, or the error-handling-detour in case it can't, but probably not both.

" anti-pattern

METHOD append\_xs.

IF input > 0.

DATA(remainder) = input.

WHILE remainder > 0.

result = result && `X`.

remainder = remainder - 1.

ENDWHILE.

ELSEIF input = 0.

RAISE EXCEPTION /dirty/sorry\_cant\_do( ).

ELSE.

RAISE EXCEPTION cx\_sy\_illegal\_argument( ).

ENDIF.

ENDMETHOD.

Can be decomposed into

METHOD append\_xs.

validate( input ).

DATA(remainder) = input.

WHILE remainder > 0.

result = result && `X`.

remainder = remainder - 1.

ENDWHILE.

ENDMETHOD.

METHOD validate.

IF input = 0.

RAISE EXCEPTION /dirty/sorry\_cant\_do( ).

ELSEIF input < 0.

RAISE EXCEPTION cx\_sy\_illegal\_argument( ).

ENDIF.

ENDMETHOD.

or, to stress the validation part

METHOD append\_xs.

IF input > 0.

result = append\_xs\_without\_check( input ).

ELSEIF input = 0.

RAISE EXCEPTION /dirty/sorry\_cant\_do( ).

ELSE.

RAISE EXCEPTION cx\_sy\_illegal\_argument( ).

ENDIF.

ENDMETHOD.

METHOD append\_xs\_without\_check.

DATA(remainder) = input.

WHILE remainder > 0.

result = result && `X`.

remainder = remainder - 1.

ENDWHILE.

ENDMETHOD.

**Descend one level of abstraction**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Method Body](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#method-body) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#descend-one-level-of-abstraction)

Statements in a method should be one level of abstraction below the method itself. Correspondingly, they should all be on the same level of abstraction.

METHOD create\_and\_publish.

post = create\_post( user\_input ).

post->publish( ).

ENDMETHOD.

instead of confusing mixtures of low level (trim, to\_upper, ...) and high level (publish, ...) concepts like

" anti-pattern

METHOD create\_and\_publish.

post = NEW blog\_post( ).

DATA(user\_name) = trim( to\_upper( sy-uname ) ).

post->set\_author( user\_name ).

post->publish( ).

ENDMETHOD.

A reliable way to find out what the right level of abstraction is is this: Let the method's author explain what the method does in few, short words, without looking at the code. The bullets (s)he numbers are the sub-methods the method should call or the statements it should execute.

**Keep methods small**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Method Body](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#method-body) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-methods-small)

Methods should be very small, optimally around 3 to 5 statements.

METHOD read\_and\_parse\_version\_filters.

DATA(active\_model\_version) = read\_random\_version\_under( model\_guid ).

DATA(filter\_json) = read\_model\_version\_filters( active\_model\_version-guid ).

result = parse\_model\_version\_filters( filter\_json ).

ENDMETHOD.

The following DATA declaration alone is sufficient to see that the surrounding method does way more than one thing:

" anti-pattern

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,

typepusages TYPE seot\_typepusages\_r,

clsdeferrds TYPE seot\_clsdeferrds\_r,

intdeferrds TYPE seot\_intdeferrds\_r,

attribute TYPE vseoattrib,

method TYPE vseomethod,

event TYPE vseoevent,

type TYPE vseotype,

alias TYPE seoaliases,

implementing TYPE vseoimplem,

friendship TYPE seofriends,

typepusage TYPE vseotypep,

clsdeferrd TYPE vseocdefer,

intdeferrd TYPE vseoidefer,

new\_clskey\_save TYPE seoclskey.

Of course there are occasions where it does not make sense to reduce a larger method further. This is perfectly okay as long as the method remains [focused on one thing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#do-one-thing-do-it-well-do-it-only):

METHOD decide\_what\_to\_do.

CASE temperature.

WHEN burning.

result = air\_conditioning.

WHEN hot.

result = ice\_cream.

WHEN moderate.

result = chill.

WHEN cold.

result = skiing.

WHEN freezing.

result = hot\_cocoa.

ENDCASE.

ENDMETHOD.

However, it still makes sense to validate whether the verbose code hides a more suitable pattern:

METHOD decide\_what\_to\_do.

result = VALUE #( spare\_time\_activities[ temperature = temperature ] OPTIONAL ).

ENDMETHOD.

Cutting methods very small can have bad impact on performance because it increases the number of method calls. The [section *Mind the performance*](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#mind-the-performance) gives guidance on how to balance Clean Code and performance.

**Control flow**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#control-flow)

**Fail fast**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Control flow](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#control-flow) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#fail-fast)

Validate and fail as early as possible:

METHOD do\_something.

IF input IS INITIAL.

RAISE EXCEPTION cx\_sy\_illegal\_argument( ).

ENDIF.

DATA(massive\_object) = build\_expensive\_object\_from( input ).

result = massive\_object->do\_some\_fancy\_calculation( ).

ENDMETHOD.

Later validations are harder to spot and understand and may have already wasted resources to get there.

" anti-pattern

METHOD do\_something.

DATA(massive\_object) = build\_expensive\_object\_from( input ).

IF massive\_object IS NOT BOUND. " happens if input is initial

RAISE EXCEPTION cx\_sy\_illegal\_argument( ).

ENDIF.

result = massive\_object->do\_some\_fancy\_calculation( ).

ENDMETHOD.

**CHECK vs. RETURN**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Control flow](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#control-flow) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#check-vs-return)

There is no consensus on whether you should use CHECK or RETURN to exit a method if the input doesn't meet expectations.

While CHECK definitely provides the shorter syntax

METHOD read\_customizing.

CHECK keys IS NOT INITIAL.

" do whatever needs doing

ENDMETHOD.

the statement's name is so obscure that people will probably understand the long form better:

METHOD read\_customizing.

IF keys IS INITIAL.

RETURN.

ENDIF.

" do whatever needs doing

ENDMETHOD:

You can also avoid the question completely by reversing the validation and adopting Dijkstra's single-exit pattern for structured programming

METHOD read\_customizing.

IF keys IS NOT INITIAL.

" do whatever needs doing

ENDIF.

ENDMETHOD:

In any case, consider whether returning nothing is really the appropriate behavior. Methods should provide a meaningful result, meaning either a filled return parameter, or an exception. Returning nothing is in many cases similar to returning null, which should be avoided.

The [section *Exiting Procedures* in the ABAP Programming Guidelines](https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/index.htm?file=abenexit_procedure_guidl.htm) recommends using CHECK in this instance. Community discussion suggests that the statement is so unclear that many people will not understand the program's behavior.

**Avoid CHECK in other positions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#methods) > [Control flow](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#control-flow) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#avoid-check-in-other-positions)

Do 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, [CHECK in a LOOP ends the current iteration and proceeds with the next one](https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-US/abapcheck_loop.htm); people might accidentally expect it to end the method or exit the loop.

Based on the [section *Exiting Procedures* in the ABAP Programming Guidelines](https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/index.htm?file=abenexit_procedure_guidl.htm). Note that this contradicts the [keyword reference for CHECK in loops](https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-US/abapcheck_loop.htm).

**Error Handling**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling)

**Return Codes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#return-codes)

**Prefer exceptions to return codes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Return Codes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#return-codes) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-exceptions-to-return-codes)

METHOD try\_this\_and\_that.

RAISE EXCEPTION NEW cx\_failed( ).

ENDMETHOD.

instead of

" anti-pattern

METHOD try\_this\_and\_that.

error\_occurred = abap\_true.

ENDMETHOD.

Exceptions have multiple advantages over return codes:

* Exceptions keep your method signatures clean: you can return the result of the method as a RETURNING parameter and still throw exceptions alongside. Return codes pollute your signatures with additional parameters for error handling.
* The caller doesn't have to react to them immediately. He can simply write down the happy path of his code. The exception-handling CATCH can be at the very end of his method, or completely outside.
* Exceptions can provide details on the error in their attributes and through methods. Return codes require you to devise a different solution on your own, such as also returning a log.
* The environment reminds the caller with syntax errors to handle exceptions. Return codes can be accidentally ignored without anybody noticing.

**Don't let failures slip through**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Return Codes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#return-codes) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-let-failures-slip-through)

If you do have to use return codes, for example because you call Functions and older code not under your control, make sure you don't let failures slip through.

DATA:

current\_date TYPE string,

response TYPE bapiret2.

CALL FUNCTION 'BAPI\_GET\_CURRENT\_DATE'

IMPORTING

current\_date = current\_date

CHANGING

response = response.

IF response-type = 'E'.

RAISE EXCEPTION NEW /clean/some\_error( );

ENDIF.

**Exceptions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#exceptions)

**Exceptions are for errors, not for regular cases**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Exceptions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#exceptions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#exceptions-are-for-errors-not-for-regular-cases)

" anti-pattern

METHODS entry\_exists\_in\_db

IMPORTING

key TYPE char10

RAISING

cx\_not\_found\_exception.

If something is a regular, valid case, it should be handled with regular result parameters.

METHODS entry\_exists\_in\_db

IMPORTING

key TYPE char10

RETURNING

VALUE(result) TYPE abap\_bool.

Exceptions should be reserved for cases that you don't expect and that reflect error situations.

METHODS assert\_user\_input\_is\_valid

IMPORTING

user\_input TYPE string

RAISING

cx\_bad\_user\_input.

Misusing exceptions misguides the reader into thinking something went wrong, when really everything is just fine. Exceptions are also much slower than regular code because they need to be constructed and often gather lots of context information.

**Use class-based exceptions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Exceptions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#exceptions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-class-based-exceptions)

TRY.

get\_component\_types( ).

CATCH cx\_has\_deep\_components\_error.

ENDTRY.

The outdated non-class-based exceptions have the same features as return codes and shouldn't be used anymore.

" anti-pattern

get\_component\_types(

EXCEPTIONS

has\_deep\_components = 1

OTHERS = 2 ).

**Throwing**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing)

**Use own super classes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Throwing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-own-super-classes)

CLASS cx\_fra\_static\_check DEFINITION ABSTRACT INHERITING FROM cx\_static\_check.

CLASS cx\_fra\_no\_check DEFINITION ABSTRACT INHERITING FROM cx\_no\_check.

Consider creating abstract super classes for each exception type for your application, instead of sub-classing the foundation classes directly. Allows you to CATCH all *your* exceptions. Enables you to add common functionality to all exceptions, such as special text handling. ABSTRACT prevents people from accidentally using these non-descriptive errors directly.

**Throw one type of exception**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Throwing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throw-one-type-of-exception)

METHODS generate

RAISING

cx\_generation\_error.

In the vast majority of cases, throwing multiple types of exceptions has no use. The caller usually is neither interested nor able to distinguish the error situations. He will therefore typically handle them all in the same way - and if this is the case, why distinguish them in the first place?

" anti-pattern

METHODS generate

RAISING

cx\_abap\_generation

cx\_hdbr\_access\_error

cx\_model\_read\_error.

A better solution to recognize different error situations is using one exception type but adding sub-classes that allow - but don't require - reacting to individual error situations, as described in [Use sub-classes to enable callers to distinguish error situations](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-sub-classes-to-enable-callers-to-distinguish-error-situations).

**Use sub-classes to enable callers to distinguish error situations**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Throwing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-sub-classes-to-enable-callers-to-distinguish-error-situations)

CLASS cx\_bad\_generation\_variable DEFINITION INHERITING FROM cx\_generation\_error.

CLASS cx\_bad\_code\_composer\_template DEFINITION INHERITING FROM cx\_generation\_error.

TRY.

generator->generate( ).

CATCH cx\_bad\_generation\_variable.

log\_failure( ).

CATCH cx\_bad\_code\_composer\_template INTO DATA(bad\_template\_exception).

show\_error\_to\_user( bad\_template\_exception ).

CATCH cx\_generation\_error INTO DATA(other\_exception).

RAISE EXCEPTION NEW cx\_application\_error( previous = other\_exception ).

ENDTRY.

If there are many different error situations, use error codes instead:

CLASS cx\_generation\_error DEFINITION ...

PUBLIC SECTION.

TYPES error\_code\_type TYPE i.

CONSTANTS:

BEGIN OF error\_code\_enum,

bad\_generation\_variable TYPE error\_code\_type VALUE 1,

bad\_code\_composer\_template TYPE error\_code\_type VALUE 2,

...

END OF error\_code\_enum.

DATA error\_code TYPE error\_code\_type.

TRY.

generator->generate( ).

CATCH cx\_generation\_error INTO DATA(exception).

CASE exception->error\_code.

WHEN cx\_generation\_error=>error\_code\_enum-bad\_generation\_variable.

WHEN cx\_generation\_error=>error\_code\_enum-bad\_code\_composer\_variable.

...

ENDCASE.

ENDTRY.

**Throw CX\_STATIC\_CHECK for manageable exceptions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Throwing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throw-cx_static_check-for-manageable-exceptions)

If an exception can be expected to occur and be reasonably handled by the receiver, throw a checked exception inheriting from CX\_STATIC\_CHECK: failing user input validation, missing resource for which there are fallbacks, etc.

CLASS cx\_file\_not\_found DEFINITION INHERITING FROM cx\_static\_check.

METHODS read\_file

IMPORTING

file\_name\_enterd\_by\_user TYPE string

RAISING

cx\_file\_not\_found.

This exception type *must* be given in method signatures and *must* be caught or forwarded to avoid syntax errors. It is therefore plain to see for the consumer and ensures that (s)he won't be surprised by an unexpected exception and will take care of reacting to the error situation.

This is in sync with the [ABAP Programming Guidelines](https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/abenexception_category_guidl.htm) but contradicts [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/), which recommends to prefer unchecked exceptions; [Exceptions](https://github.com/SAP/styleguides/blob/master/clean-abap/sub-sections/Exceptions.md) explains why.

**Throw CX\_NO\_CHECK for usually unrecoverable situations**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Throwing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throw-cx_no_check-for-usually-unrecoverable-situations)

If an exception is so severe that the receiver is unlikely to recover from it, use CX\_NO\_CHECK: failure to read a must-have resource, failure to resolve the requested dependency, etc.

CLASS cx\_out\_of\_memory DEFINITION INHERITING FROM cx\_no\_check.

METHODS create\_guid

RETURNING

VALUE(result) TYPE /bobf/conf\_key.

CX\_NO\_CHECK *cannot* be declared in method signatures, such that its occurrence will come as a bad surprise to the consumer. In the case of unrecoverable situations, this is okay because the consumer will not be able to do anything useful about it anyway.

However, there *may* be cases where the consumer actually wants to recognize and react to this kind of failure. For example, a dependency manager could throw a CX\_NO\_CHECK if it's unable to provide an implementation for a requested interface because regular application code will not be able to continue. However, there may be a test report that tries to instantiate all kinds of things just to see if it's working, and that will report failure simply as a red entry in a list - this service should be able to catch and ignore the exception instead of being forced to dump.

**Consider CX\_DYNAMIC\_CHECK for avoidable exceptions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Throwing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#consider-cx_dynamic_check-for-avoidable-exceptions)

Use cases for CX\_DYNAMIC\_CHECK are rare, and in general we recommend to resort to the other exception types. However, you may want to consider this kind of exception as a replacement for CX\_STATIC\_CHECK if the caller has full, conscious control over whether an exception can occur.

DATA value TYPE decfloat.

value = '7.13'.

cl\_abap\_math=>get\_db\_length\_decs(

EXPORTING

in = value

IMPORTING

length = DATA(length) ).

For example, consider the method get\_db\_length\_decs of class cl\_abap\_math, that tells you the number of digits and decimal places of a decimal floating point number. This method raises the dynamic exception cx\_parameter\_invalid\_typeif the input parameter does not reflect a decimal floating point number. Usually, this method will be called for a fully and statically typed variable, such that the developer knows whether that exception can ever occur or not. In this case, the dynamic exception would enable the caller to omit the unnecessary CATCH clause.

**Dump for totally unrecoverable situations**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Throwing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dump-for-totally-unrecoverable-situations)

If a situation is so severe that you are totally sure the receiver is unlikely to recover from it, or that clearly indicates a programming error, dump instead of throwing an exception: failure to acquire memory, failed index reads on a table that must be filled, etc.

RAISE SHORTDUMP TYPE cx\_sy\_create\_object\_error. " >= NW 7.53

MESSAGE x666(general). " < NW 7.53

This behavior will prevent any kind of consumer from doing anything useful afterwards. Use this only if you are sure about that.

**Prefer RAISE EXCEPTION NEW to RAISE EXCEPTION TYPE**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Throwing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#throwing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#prefer-raise-exception-new-to-raise-exception-type)

RAISE EXCEPTION NEW cx\_generation\_error( previous = exception ).

in general is shorter than the needlessly longer

RAISE EXCEPTION TYPE cx\_generation\_error

EXPORTING

previous = exception.

However, if you make massive use of the addition MESSAGE, you may want to stick with the TYPE variant:

RAISE EXCEPTION TYPE cx\_generation\_error

EXPORTING

previous = exception

MESSAGE e136(messages).

**Catching**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#catching)

**Wrap foreign exceptions instead of letting them invade your code**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Error Handling](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#error-handling) > [Catching](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#catching) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#wrap-foreign-exceptions-instead-of-letting-them-invade-your-code)

METHODS generate RAISING cx\_generation\_failure.

METHOD generate.

TRY.

generator->generate( ).

CATCH cx\_amdp\_generation\_failure INTO DATA(exception).

RAISE EXCEPTION NEW cx\_generation\_failure( previous = exception ).

ENDTRY.

ENDMETHOD.

The [Law of Demeter](https://en.wikipedia.org/wiki/Law_of_Demeter) recommends de-coupling things. Forwarding exceptions from other components violates this principle. Make yourself independent from the foreign code by catching those exceptions and wrapping them in an exception type of your own.

" anti-pattern

METHODS generate RAISING cx\_sy\_gateway\_failure.

METHOD generate.

generator->generate( ).

ENDMETHOD.

**Comments**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments)

**Express yourself in code, not in comments**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#express-yourself-in-code-not-in-comments)

METHOD correct\_day\_to\_last\_in\_month.

WHILE is\_invalid( date ).

reduce\_day\_by\_one( CHANGING date = date ).

ENDWHILE.

ENDMETHOD.

METHOD is\_invalid.

DATA zero\_if\_invalid TYPE i.

zero\_if\_invalid = date.

result = xsdbool( zero\_if\_invalid = 0 ).

ENDMETHOD.

METHOD reduce\_day\_by\_one.

date+6(2) = date+6(2) - 1.

ENDMETHOD.

instead of

" anti-pattern

" correct e.g. 29.02. in non-leap years as well as result of a date calculation would be

" something like e.g. the 31.06. that example has to be corrected to 30.06.

METHOD fix\_day\_overflow.

DO 3 TIMES.

" 31 - 28 = 3 => this correction is required not more than 3 times

lv\_dummy = cv\_date.

" lv\_dummy is 0 if the date value is a not existing date - ABAP specific implementation

IF ( lv\_dummy EQ 0 ).

cv\_date+6(2) = cv\_date+6(2) - 1. " subtract 1 day from the given date

ELSE.

" date exists => no correction required

EXIT.

ENDIF.

ENDDO.

ENDMETHOD.

Clean Code does *not* forbid you to comment your code - it encourages you to exploit *better* means, and resort to comments only if that fails.

This example has been challenged from a performance point of view, claiming that cutting the methods so small worsens performance too much. Sample measurements show that the refactored code is 2.13 times slower than the original dirty variant. The clean variant takes 9.6 microseconds to fix the input 31-02-2018, the dirty variant only 4.5 microseconds. This may be a problem when the method is run very often in a high-performance application; for regular user input validation, it should be acceptable. Resort to the section [Mind the performance](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#mind-the-performance) to deal with Clean Code and performance issues.

**Comments are no excuse for bad names**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments-are-no-excuse-for-bad-names)

DATA(input\_has\_entries) = has\_entries( input ).

Improve your names instead of explaining what they really mean or why you chose bad ones.

" anti-pattern

" checks whether the table input contains entries

DATA(result) = check\_table( input ).

**Use methods instead of comments to segment your code**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-methods-instead-of-comments-to-segment-your-code)

DATA(statement) = build\_statement( ).

DATA(data) = execute\_statement( statement ).

This not only makes the intent, structure, and dependencies of the code much clearer, it also avoids carry-over errors when temporary variables aren't properly cleared between the sections.

" anti-pattern

" -----------------

" Build statement

" -----------------

DATA statement TYPE string.

statement = |SELECT \* FROM d\_document\_roots|.

" -----------------

" Execute statement

" -----------------

DATA(result\_set) = adbc->execute\_sql\_query( statement ).

result\_set->next\_package( IMPORTING data = data ).

**Write comments to explain the why, not the what**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#write-comments-to-explain-the-why-not-the-what)

" can't fail, existence of >= 1 row asserted above

DATA(first\_line) = table[ 1 ].

Nobody needs repeating the code in natural language

" anti-pattern

" select alert root from database by key

SELECT \* FROM d\_alert\_root WHERE key = key.

**Design goes into the design documents, not the code**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#design-goes-into-the-design-documents-not-the-code)

" anti-pattern

" This class serves a double purpose. First, it does one thing. Then, it does another thing.

" It does so by executing a lot of code that is distributed over the local helper classes.

" To understand what's going on, let us at first ponder the nature of the universe as such.

" Have a look at this and that to get the details.

Nobody reads that - seriously. If people need to read a textbook to be able to use your code, this may be an indicator that your code has severe design issues that you should solve otherwise. Some code *does* need some explanation beyond a single line of comment; consider linking the design document in these cases.

**Comment with ", not with \***

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comment-with--not-with-)

Quote comments indent along with the statements they comment

METHOD do\_it.

IF input IS NOT INITIAL.

" delegate pattern

output = calculate\_result( input ).

ENDIF.

ENDMETHOD.

Asterisked comments tend to indent to weird places

" anti-pattern

METHOD do\_it.

IF input IS NOT INITIAL.

\* delegate pattern

output = calculate\_result( input ).

ENDIF.

ENDMETHOD.

**Put comments before the statement they relate to**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#put-comments-before-the-statement-they-relate-to)

" delegate pattern

output = calculate\_result( input ).

Clearer than

" anti-pattern

output = calculate\_result( input ).

" delegate pattern

And less invasive than

output = calculate\_result( input ). " delegate pattern

**Delete code instead of commenting it**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#delete-code-instead-of-commenting-it)

" anti-pattern

\* output = calculate\_result( input ).

When you find something like this, delete it. The code is obviously not needed because your application works and all tests are green. Deleted code can be reproduced from the version history later on. If you need to preserve a piece of code permanently, copy it to a file or a $TMP or HOME object.

**Use FIXME, TODO, and XXX and add your ID**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-fixme-todo-and-xxx-and-add-your-id)

METHOD do\_something.

" XXX FH delete this method - it does nothing

ENDMETHOD.

* FIXME points to errors that are too small or too much in-the-making for internal incidents.
* TODOs are places where you want to complete something in the near(!) future.
* XXX marks code that works but could be better.

When you enter such a comment, add your nick, initials, or user to enable your co-developers to contact you and ask questions if the comment is unclear.

**Don't add method signature and end-of comments**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-add-method-signature-and-end-of-comments)

Method signature comments don't help anybody.

" anti-pattern

\* <SIGNATURE>---------------------------------------------------------------------------------------+

\* | Static Public Method CALIBRATION\_KPIS=>CALCULATE\_KPI

\* +-------------------------------------------------------------------------------------------------+

\* | [--->] STRATEGY\_ID TYPE STRATEGY\_ID

\* | [--->] THRESHOLD TYPE STRATEGY\_THRESHOLD

\* | [--->] DETECTION\_OBJECT\_SCORE TYPE T\_HIT\_RESULT

\* | [<---] KPI TYPE T\_SIMULATED\_KPI

\* +--------------------------------------------------------------------------------------</SIGNATURE>

Decades ago, when you couldn't see the method signature when inspecting its code, or working with printouts that had dozens of pages, these comments may have made sense. But all modern ABAP IDEs (SE24, SE80, ADT) show the method signature easily such that these comments have become nothing but noise.

In the form-based editor of SE24/SE80, press button *Signature*. In the ABAP Development Tools, mark the method name and press F2 or add the view *ABAP Element Info* to your perspective.

Similarly, end-of comments are superfluous. These comments may have been helpful decades ago, when programs and functions and the nested IFs inside were hundreds of lines of code long. But our modern coding style produces methods short enough to readily see what opening statement an ENDIF or ENDMETHOD belongs to:

" anti-pattern

METHOD get\_kpi\_calc.

IF has\_entries = abap\_false.

result = 42.

ENDIF. " IF has\_entries = abap\_false

ENDMETHOD. " get\_kpi\_calc

**Don't duplicate message texts as comments**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-duplicate-message-texts-as-comments)

" anti-pattern

" alert category not filled

MESSAGE e003 INTO dummy.

Messages change independently from your code, and nobody will remember adjusting the comment, such that it will outdate and even become misleading quickly and without anybody noticing.

The modern IDEs give you easy ways to see the text behind a message, for example in the ABAP Development Tools, mark the message ID and press Shift+F2.

If you want it more explicit, consider extracting the message to a method of its own.

METHOD create\_alert\_not\_found\_message.

MESSAGE e003 INTO dummy.

ENDMETHOD.

**ABAP Doc only for public APIs**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Comments](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#comments) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#abap-doc-only-for-public-apis)

Write ABAP Doc to document public APIs, meaning APIs that are intended for developers in other teams or applications. Don't write ABAP Doc for internal stuff.

ABAP Doc suffers from the same weaknesses as all comments, that is, it outdates quickly and then becomes misleading. As a consequence, you should employ it only where it makes sense, not enforce writing ABAP Doc for each and everything.

Read more in *Chapter 4: Good Comments: Javadocs in Public APIs* and *Chapter 4: Bad Comments: Javadocs in Nonpublic Code* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Formatting**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting)

The suggestions below are [optimized for reading, not for writing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#optimize-for-reading-not-for-writing). As ABAP's Pretty Printer doesn't cover them, some of them produce additional manual work to reformat statements when name lengths etc. change; if you want to avoid this, consider dropping rules like [Align assignments to the same object, but not to different ones](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#align-assignments-to-the-same-object-but-not-to-different-ones).

**Be consistent**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#be-consistent)

Format all code of a project in the same way. Let all team members use the same formatting style.

If you edit foreign code, adhere to that project's formatting style instead of insisting on your personal style.

If you change your formatting rules over time, use [refactoring best practices](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-refactor-legacy-code) to update your code over time.

**Optimize for reading, not for writing**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#optimize-for-reading-not-for-writing)

Developers spend most time *reading* code. Actually *writing* code takes up a way smaller portion of the day.

As a consequence, you should optimize your code formatting for reading, not for writing.

For example, you should prefer

DATA:

a TYPE b,

c TYPE d,

e TYPE f.

to hacks such as

" anti-pattern

DATA:

a TYPE b

,c TYPE d

,e TYPE f.

**Use the Pretty Printer before activating**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-the-pretty-printer-before-activating)

Apply the pretty printer - Shift+F1 in SE80, SE24, and ADT - before activating an object.

If you modify a larger unformatted legacy code base, you may want to apply the Pretty Printer only to selected lines to avoid huge change lists and transport dependencies. Consider pretty-printing the complete development object in a separate Transport Request or Note.

Read more in *Chapter 5: Formatting: Team Rules* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Use your Pretty Printer team settings**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-your-pretty-printer-team-settings)

Always use your team settings. Specify them under *Menu* > *Utilities* > *Settings ...* > *ABAP Editor* > *Pretty Printer*.

Set *Indent* and *Convert Uppercase/Lowercase* > *Uppercase Keyword* as agreed in your team.

[Upper vs. Lower Case](https://github.com/SAP/styleguides/blob/master/clean-abap/sub-sections/UpperVsLowerCase.md) explains why we do not give clear guidance for the type case of keywords.

Read more in *Chapter 5: Formatting: Team Rules* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**No more than one statement per line**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#no-more-than-one-statement-per-line)

DATA do\_this TYPE i.

do\_this = input + 3.

Even if some occurrences may trick you into the misconception that this was readable:

" anti-pattern

DATA do\_this TYPE i. do\_this = input + 3.

**Stick to a reasonable line length**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#stick-to-a-reasonable-line-length)

Adhere to a maximum line length of 120 characters.

The human eye reads text more comfortably if the lines are not too wide - ask a UI designer or eye movement researcher of your choice. You will also appreciate the narrower code when debugging or comparing two sources next to each other.

The 80 or even 72 characters limit originating in the old terminal devices is a little too restrictive. While 100 characters are often recommended and a viable choice, 120 characters seem to work a little better for ABAP, maybe because of the general verbosity of the language.

As a reminder you can configure in ADT the print margin to 120 characters, which then is visualized in the code view as a vertical line. Configure it under *Menu* > *Window* > *Preferences* > *General* > *Editors* > *Text Editors*.

**Condense your code**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#condense-your-code)

DATA(result) = calculate( items ).

instead of adding unneeded blanks

" anti-pattern

DATA(result) = calculate( items = items ) .

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

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#add-a-single-blank-line-to-separate-things-but-not-more)

DATA(result) = do\_something( ).

DATA(else) = calculate\_this( result ).

to highlight that the two statements do different things. But there is no reason for

" anti-pattern

DATA(result) = do\_something( ).

DATA(else) = calculate\_this( result ).

The urge to add separating blank lines may be an indicator that your method doesn't [do one thing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#do-one-thing-do-it-well-do-it-only).

**Don't obsess with separating blank lines**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-obsess-with-separating-blank-lines)

METHOD do\_something.

do\_this( ).

then\_that( ).

ENDMETHOD.

No reason for the bad habit to tear your code apart with blank lines

" anti-pattern

METHOD do\_something.

do\_this( ).

then\_that( ).

ENDMETHOD.

Blank lines actually only make sense if you have statements that span multiple lines

METHOD do\_something.

do\_this( ).

then\_that(

EXPORTING

variable = 'A'

IMPORTING

result = result ).

ENDMETHOD.

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

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#align-assignments-to-the-same-object-but-not-to-different-ones)

To highlight that these things somehow belong together

structure-type = 'A'.

structure-id = '4711'.

or even better

structure = VALUE #( type = 'A'

id = '4711' ).

But leave things ragged that have nothing to do with each other:

customizing\_reader = fra\_cust\_obj\_model\_reader=>s\_get\_instance( ).

hdb\_access = fra\_hdbr\_access=>s\_get\_instance( ).

Read more in *Chapter 5: Formatting: Horizontal Alignment* of [Robert C. Martin's *Clean Code*](https://www.oreilly.com/library/view/clean-code/9780136083238/).

**Close brackets at line end**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#close-brackets-at-line-end)

modify->update( node = if\_fra\_alert\_c=>node-item

key = item->key

data = item

changed\_fields = changed\_fields ).

instead of the needlessly longer

" anti-pattern

modify->update( node = if\_fra\_alert\_c=>node-item

key = item->key

data = item

changed\_fields = changed\_fields

).

**Keep single parameter calls on one line**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-single-parameter-calls-on-one-line)

DATA(unique\_list) = remove\_duplicates( list ).

remove\_duplicates( CHANGING list = list ).

instead of the needlessly longer

" anti-pattern

DATA(unique\_list) = remove\_duplicates(

list ).

DATA(unique\_list) = remove\_duplicates(

CHANGING

list = list ).

**Keep parameters behind the call**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#keep-parameters-behind-the-call)

DATA(sum) = add\_two\_numbers( value\_1 = 5

value\_2 = 6 ).

When this makes the lines very long, you can break the parameters into the next line:

DATA(sum) = add\_two\_numbers(

value\_1 = round\_up( input DIV 7 ) \* 42 + round\_down( 19 \* step\_size )

value\_2 = VALUE #( ( `Calculation failed with a very weird result` ) ) ).

**If you break, indent parameters under the call**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#if-you-break-indent-parameters-under-the-call)

DATA(sum) = add\_two\_numbers(

value\_1 = 5

value\_2 = 6 ).

Aligning the parameters elsewhere makes it hard to spot what they belong to:

" anti-pattern

DATA(sum) = add\_two\_numbers(

value\_1 = 5

value\_2 = 6 ).

This is on the other side the best pattern if you want to avoid the formatting to be broken by a name length change.

**Line-break multiple parameters**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#line-break-multiple-parameters)

DATA(sum) = add\_two\_numbers( value\_1 = 5

value\_2 = 6 ).

Yes, this wastes space. However, otherwise, it's hard to spot where one parameter ends and the next starts:

" anti-pattern

DATA(sum) = add\_two\_numbers( value\_1 = 5 value\_2 = 6 ).

**Align parameters**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#align-parameters)

modify->update( node = if\_fra\_alert\_c=>node-item

key = item->key

data = item

changed\_fields = changed\_fields ).

Ragged margins make it hard to see where the parameter ends and its value begins:

" anti-pattern

modify->update( node = if\_fra\_alert\_c=>node-item

key = item->key

data = item

changed\_fields = changed\_fields ).

This is on the other side the best pattern if you want to avoid the formatting to be broken by a name length change.

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

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#break-the-call-to-a-new-line-if-the-line-gets-too-long)

DATA(some\_super\_long\_param\_name) =

if\_some\_annoying\_interface~add\_two\_numbers\_in\_a\_long\_name(

value\_1 = 5

value\_2 = 6 ).

**Indent and snap to tab**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#indent-and-snap-to-tab)

Indent parameter keywords by 2 spaces and parameters by 4 spaces:

DATA(sum) = add\_two\_numbers(

EXPORTING

value\_1 = 5

value\_2 = 6

CHANGING

errors = errors ).

If you have no keywords, indent the parameters by 4 spaces.

DATA(sum) = add\_two\_numbers(

value\_1 = 5

value\_2 = 6 ).

Use the Tab key to indent. It's okay if this adds one more space than needed. (This happens if the DATA(sum) = part at the left has an uneven number of characters.)

**Indent in-line declarations like method calls**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#indent-in-line-declarations-like-method-calls)

Indent in-line declarations with VALUE or NEW as if they were method calls:

DATA(result) = merge\_structures( a = VALUE #( field\_1 = 'X'

field\_2 = 'A' )

b = NEW /clean/structure\_type( field\_3 = 'C'

field\_4 = 'D' ) ).

**Don't align type clauses**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Formatting](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#formatting) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-align-type-clauses)

DATA name TYPE seoclsname.

DATA reader TYPE REF TO /clean/reader.

A variable and its type belong together and should therefore be visually grouped in close proximity. Aligning the TYPEclauses draws attention away from that and suggests that the variables form one vertical group, and their types another one. Alignment also produces needless editing overhead, requiring you to adjust all indentations when the length of the longest variable name changes.

" anti-pattern

DATA name TYPE seoclsname.

DATA reader TYPE REF TO /clean/reader.

**Testing**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing)

**Principles**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#principles)

**Write testable code**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Principles](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#principles) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#write-testable-code)

Write all code in a way that allows you to test it in an automatic fashion.

If this requires refactoring your code, do it. Do that first, before you start adding other features.

If you add to legacy code that is too badly structured to be tested, refactor it at least to the extent that you can test your additions.

**Enable others to mock you**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Principles](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#principles) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#enable-others-to-mock-you)

If you write code to be consumed by others, enable them to write unit tests for their own code, for example by adding interfaces in all outward-facing places, providing helpful test doubles that facilitate integration tests, or applying dependency inversion to enable them to substitute the productive configuration with a test config.

**Readability rules**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Principles](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#principles) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#readability-rules)

Make your test code even more readable than your productive code. You can tackle bad productive code with good tests, but if you don't even get the tests, you're lost.

Keep your test code so simple and stupid that you will still understand it in a year from now.

Stick to standards and patterns, to enable your co-workers to quickly get into the code.

**Don't make copies or write test reports**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Principles](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#principles) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-make-copies-or-write-test-reports)

Don't start working on a backlog item by making a $TMP copy of a development object and playing around with it. Others won't notice these objects and therefore won't know the status of your work. You will probably waste a lot of time by making the working copy in the first place. You will also forget to delete the copy afterwards, spamming your system and dependencies. (Don't believe this? Go to your development system and check your $TMP right now.)

Also, don't start by writing a test report that calls something in a specific way, and repeat that to verify that things are still working when you're working on it. This is poor man's testing: repeating a test report by hand and verifying by eye whether everything is still fine. Take the next step and automate this report in a unit test, with an automatic assertion that tells you whether the code is still okay. First, you will spare yourself the effort of having to write the unit tests afterwards. Second, you will save a lot of time for the manual repetitions, plus avoid getting bored and tired over it.

**Test publics, not private internals**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Principles](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#principles) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-publics-not-private-internals)

Public parts of classes, especially the interfaces they implement, are rather stable and unlikely to change. Let your unit tests validate only the publics to make them robust and minimize the effort you have to spend when you refactor the class. Protected and private internals, in contrast, may change very quickly through refactoring, such that each refactoring would needlessly break your tests.

An urgent need to test private or protected methods may be an early warning sign for several kinds of design flaws. Ask yourself:

* Did you accidentally bury a concept in your class that wants to come out into its own class, with its own dedicated suite of tests?
* Did you forget to separate the domain logic from the glue code? For example, implementing the domain logic directly in the class that is plugged into BOPF as an action, determination, or validation, or that was generated by SAP Gateway as a \*\_DPC\_EXT data provider, may not the best idea.
* Are your interfaces too complicated and request too much data that is irrelevant or that cannot be mocked easily?

**Don't obsess about coverage**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Principles](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#principles) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-obsess-about-coverage)

Code coverage is there to help you find code you forgot to test, not to meet some random KPI:

Don't make up tests without or with dummy asserts just to reach the coverage. Better leave things untested to make transparent that you cannot safely refactor them. You can have < 100% coverage and still have perfect tests. There are cases - such as IFs in the constructor to insert test doubles - that may make it unpractical to reach 100%. Good tests tend to cover the same statement multiple times, for different branches and conditions. They will in fact have imaginary > 100% coverage.

**Test Classes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-classes)

**Call local test classes by their purpose**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-classes) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#call-local-test-classes-by-their-purpose)

CLASS ltc\_unit\_tests DEFINITION FOR TESTING ... .

CLASS ltc\_integration\_tests DEFINITION FOR TESTING ... .

CLASS ltc\_unit\_tests\_with\_mocks DEFINITION FOR TESTING ... .

Good names reveal the level of the tests and what's common to their setup.

" anti-patterns

CLASS ltc\_fra\_online\_detection\_api DEFINITION FOR TESTING ... . " We know that's the class under test - why repeat it?

CLASS ltc\_test DEFINITION FOR TESTING .... " Of course it's a test, what else should it be?

**Put tests in local classes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-classes) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#put-tests-in-local-classes)

Put unit tests into the local test include of the class under test. This ensures that people find these tests when refactoring the class and allows them to run all associated tests with a single key press, as described in [How to execute test classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-execute-test-classes).

Put component/integration/system tests, that do not directly relate to a single class under test, into the local test include of a separate global class. Mark this global "container" class as FOR TESTING and ABSTRACT to avoid that it is accidentally delivered or referenced in production code. Putting tests into other classes has the danger that people overlook them and forget to run them when refactoring the involved classes, such that this choice is only second-best.

**How to execute test classes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Classes](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-classes) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#how-to-execute-test-classes)

In the ABAP Development Tools, press Ctrl+Shift+F10 to run all tests in a class. Press Ctrl+Shift+F11 to include coverage measurements. Press Ctrl+Shift+F12 to also run tests in other classes that are maintained as test relations.

On macOS, use Cmd instead of Ctrl.

**Code Under Test**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#code-under-test)

**Name the code under test meaningfully, or default to CUT**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Code Under Test](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#code-under-test) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#name-the-code-under-test-meaningfully-or-default-to-cut)

Give the variable that represents the code under test a meaningful name:

DATA blog\_post TYPE REF TO ...

Don't just repeat the class name with all its non-valuable namespaces and prefixes:

" anti-pattern

DATA clean\_fra\_blog\_post TYPE REF TO ...

If you have different test setups, it can be helpful to describe the object's varying state:

DATA empty\_blog\_post TYPE REF TO ...

DATA simple\_blog\_post TYPE REF TO ...

DATA very\_long\_blog\_post TYPE REF TO ...

If you have problems finding a meaningful name, resort to cut as a default. The abbreviation stands for "code under test".

DATA cut TYPE REF TO ...

Especially in unclean and confusing tests, calling the variable cut can temporarily help the reader see what's actually tested. However, tidying up the tests is the actual way to go for the long run.

**Test interfaces, not classes**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Code Under Test](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#code-under-test) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-interfaces-not-classes)

A practical consequence of the [*Test publics, not private internals*](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-publics-not-private-internals), type your code under test with an *interface*

DATA code\_under\_test TYPE REF TO some\_interface.

rather than a *class*

" anti-pattern

DATA code\_under\_test TYPE REF TO some\_class.

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

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Code Under Test](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#code-under-test) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#extract-the-call-to-the-code-under-test-to-its-own-method)

If the method to be tested requires a lot of parameters or prepared data, it can help to extract the call to it to a helper method of its own that defaults the uninteresting parameters:

METHODS map\_xml\_to\_itab

IMPORTING

xml\_string TYPE string

config TYPE /clean/xml2itab\_config DEFAULT default\_config

format TYPE /clean/xml2itab\_format DEFAULT default\_format.

METHOD map\_xml\_to\_itab.

result = cut->map\_xml\_to\_itab( xml\_string = xml\_string

config = config

format = format ).

ENDMETHOD.

DATA(itab) = map\_xml\_to\_itab( '<xml></xml>' ).

Calling the original method directly can swamp your test with a lot of meaningless details:

" anti-pattern

DATA(itab) = cut->map\_xml\_to\_itab( xml\_string = '<xml></xml>'

config = VALUE #( 'some meaningless stuff' )

format = VALUE #( 'more meaningless stuff' ) ).

**Injection**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection)

**Use dependency inversion to inject test doubles**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-dependency-inversion-to-inject-test-doubles)

Dependency inversion means that you hand over all dependencies to the constructor:

METHODS constructor

IMPORTING

customizing\_reader TYPE REF TO if\_fra\_cust\_obj\_model\_reader.

METHOD constructor.

me->customizing\_reader = customizing\_reader.

ENDMETHOD.

Don't use setter injection. It enables using the productive code in ways that are not intended:

" anti-pattern

METHODS set\_customizing\_reader

IMPORTING

customizing\_reader TYPE REF TO if\_fra\_cust\_obj\_model\_reader.

METHOD do\_something.

object->set\_customizing\_reader( a ).

object->set\_customizing\_reader( b ). " would you expect that somebody does this?

ENDMETHOD.

Don't use FRIENDS injection. It will initialize productive dependencies before they are replaced, with probably unexpected consequences. It will break as soon as you rename the internals. It also circumvents initializations in the constructor.

" anti-pattern

METHOD setup.

cut = NEW fra\_my\_class( ). " <- builds a productive customizing\_reader first - what will it break with that?

cut->customizing\_reader ?= cl\_abap\_testdouble=>create( 'if\_fra\_cust\_obj\_model\_reader' ).

ENDMETHOD.

METHOD constructor.

customizing\_reader = fra\_cust\_obj\_model\_reader=>s\_get\_instance( ).

customizing\_reader->fill\_buffer( ). " <- won't be called on your test double, so no chance to test this

ENDMETHOD.

**Use CL\_ABAP\_TESTDOUBLE**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-cl_abap_testdouble)

DATA(customizing\_reader) = CAST /clean/customizing\_reader( cl\_abap\_testdouble=>create( '/clean/default\_custom\_reader' ) ).

cl\_abap\_testdouble=>configure\_call( customizing\_reader)->returning( sub\_claim\_customizing ).

customizing\_reader->read( 'SOME\_ID' ).

Shorter and easier to understand than custom test doubles:

" anti-pattern

CLASS /dirty/default\_custom\_reader DEFINITION FOR TESTING CREATE PUBLIC.

PUBLIC SECTION.

INTERFACES /dirty/customizing\_reader.

DATA customizing TYPE /dirty/customizing\_table.

ENDCLASS.

CLASS /dirty/default\_custom\_reader IMPLEMENTATION.

METHOD /dirty/customizing\_reader~read.

result = customizing.

ENDMETHOD.

ENDCLASS.

METHOD test\_something.

DATA(customizing\_reader) = NEW /dirty/customizing\_reader( ).

customizing\_reader->customizing = sub\_claim\_customizing.

ENDMETHOD.

**Exploit the test tools**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#exploit-the-test-tools)

In general, a clean programming style will let you do much of the work with standard ABAP unit tests and test doubles. However, there are tools that will allow you to tackle trickier cases in elegant ways:

* Use the CL\_OSQL\_REPLACE service to test complex OpenSQL statements by redirecting them to a test data bin that can be filled with test data without interfering with the rest of the system.
* Use the CDS test framework to test your CDS views.

**Use test seams as temporary workaround**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-test-seams-as-temporary-workaround)

If all other techniques fail, or when in dangerous shallow waters of legacy code, refrain to [test seams](https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/index.htm?file=abendyn_access_data_obj_guidl.htm) to make things testable.

Although they look comfortable at first sight, test seams are invasive and tend to get entangled in private dependencies, such that they are hard to keep alive and stable in the long run.

We therefore recommend to refrain to test seams only as a temporary workaround to allow you refactoring the code into a more testable form.

**Use LOCAL FRIENDS to access the dependency-inverting constructor**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-local-friends-to-access-the-dependency-inverting-constructor)

CLASS /clean/unit\_tests DEFINITION.

PRIVATE SECTION.

DATA cut TYPE REF TO /clean/interface\_under\_test.

METHODS setup.

ENDCLASS.

CLASS /clean/class\_under\_test DEFINITION LOCAL FRIENDS unit\_tests.

CLASS unit\_tests IMPLEMENTATION.

METHOD setup.

DATA(mock) = cl\_abap\_testdouble=>create( '/clean/some\_mock' ).

" /clean/class\_under\_test is CREATE PRIVATE

" so this only works because of the LOCAL FRIENDS

cut = NEW /clean/class\_under\_test( mock ).

ENDMETHOD.

ENDCLASS.

**Don't misuse LOCAL FRIENDS to invade the tested code**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-misuse-local-friends-to-invade-the-tested-code)

Unit tests that access private and protected members to insert mock data are fragile: they break when the internal structure of the tested code changes.

" anti-pattern

CLASS /dirty/class\_under\_test DEFINITION LOCAL FRIENDS unit\_tests.

CLASS unit\_tests IMPLEMENTATION.

METHOD returns\_right\_result.

cut->some\_private\_member = 'AUNIT\_DUMMY'.

ENDMETHOD.

ENDCLASS.

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

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-change-the-productive-code-to-make-the-code-testable)

" anti-pattern

IF me->in\_test\_mode = abap\_true.

**Don't sub-class to mock methods**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-sub-class-to-mock-methods)

Don't sub-class and overwrite methods to mock them in your unit tests. Although this works, it is fragile because the tests break easily when refactoring the code. It also enables real consumers to inherit your class, which [may hit you unprepared when not explicitly designing for it](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#final-if-not-designed-for-inheritance).

" anti-pattern

CLASS unit\_tests DEFINITION INHERITING FROM /dirty/real\_class FOR TESTING [...].

PROTECTED SECTION.

METHODS needs\_to\_be\_mocked REDEFINITION.

To get legacy code under test, [resort to test seams instead](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-test-seams-as-temporary-workaround). They are just as fragile but still the cleaner way because they at least don't change the class's productive behavior, as would happen when enabling inheritance by removing a previous FINAL flag or by changing method scope from PRIVATE to PROTECTED.

When writing new code, take this testability issue into account directly when designing the class, and find a different, better way. Common best practices include [resorting to other test tools](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#exploit-the-test-tools) and extracting the problem method to a separate class with its own interface.

A more specific variant of [Don't change the productive code to make the code testable](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-change-the-productive-code-to-make-the-code-testable).

**Don't mock stuff that's not needed**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-mock-stuff-thats-not-needed)

cut = NEW /clean/class\_under\_test( db\_reader = db\_reader

config = VALUE #( )

writer = VALUE #( ) ).

Define your givens as precisely as possible: don't set data that your test doesn't need, and don't mock objects that are never called. These things distract the reader from what's really going on.

" anti-pattern

cut = NEW /dirty/class\_under\_test( db\_reader = db\_reader

config = config

writer = writer ).

There are also cases where it's not necessary to mock something at all - this is usually the case with data structures and data containers. For example, your unit tests may well work with the productive version of a transient\_log because it only stores data without any side effects.

**Don't build test frameworks**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Injection](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#injection) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-build-test-frameworks)

Unit tests should be data-in-data-out, with all test data being defined on the fly as needed.

cl\_abap\_testdouble=>configure\_call( test\_double )->returning( data ).

Don't start building frameworks that distinguish "test case IDs" to decide what data to provide. The resulting code will be so long and tangled that you won't be able to keep these tests alive in the long term.

" anti-pattern

test\_double->set\_test\_case( 1 ).

CASE me->test\_case.

WHEN 1.

WHEN 2.

ENDCASE.

**Test Methods**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-methods)

**Test method names: reflect what's given and expected**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-method-names-reflect-whats-given-and-expected)

Good names reflect the given and then of the test:

METHOD reads\_existing\_entry.

METHOD throws\_on\_invalid\_key.

METHOD detects\_invalid\_input.

Bad names reflect the when, repeat meaningless facts, or are cryptic:

" anti-patterns

" What's expected, success or failure?

METHOD get\_conversion\_exits.

" It's a test method, what else should it do but "test"?

METHOD test\_loop.

" So it's parameterized, but what is its aim?

METHOD parameterized\_test.

" What's "\_wo\_w" supposed to mean and will you still remember that in a year from now?

METHOD get\_attributes\_wo\_w.

As ABAP allows only 30 characters in method names, it's fair to add an explanatory comment if the name is too short to convey enough meaning. ABAP Doc or the first line in the test method may be an appropriate choice for the comment.

Having lots of test methods whose names are too long may be an indicator that you should split your single test class into several ones and express the differences in the givens in the class's names.

**Use given-when-then**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-given-when-then)

Organize your test code along the given-when-then paradigm: First, initialize stuff in a given section ("given"), second call the actual tested thing ("when"), third validate the outcome ("then").

If the given or then sections get so long that you cannot visually separate the three sections anymore, extract sub-methods. Blank lines or comments as separators may look good at first glance but don't really reduce the visual clutter. Still they are helpful for the reader and the novice test writer to separate the sections.

**"When" is exactly one call**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#when-is-exactly-one-call)

Make sure that the "when" section of your test method contains exactly one call to the class under test:

METHOD rejects\_invalid\_input.

" when

DATA(is\_valid) = cut->is\_valid\_input( 'SOME\_RANDOM\_ENTRY' ).

" then

cl\_abap\_unit\_assert=>assert\_true( is\_valid ).

ENDMETHOD.

Calling multiple things indicates that the method has no clear focus and tests too much. This makes it harder to find the cause when the test fails: was it the first, second, or third call that caused the failure? It also confuses the reader because he is not sure what the exact feature under test is.

**Don't add a TEARDOWN unless you really need it**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Methods](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-methods) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#dont-add-a-teardown-unless-you-really-need-it)

teardown methods are usually only needed to clear up database entries or other external resources in integration tests.

Resetting members of the test class, esp. cut and the used test doubles, is superfluous; they are overwritten by the setupmethod before the next test method is started.

**Test Data**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-data)

**Make it easy to spot meaning**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Data](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-data) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#make-it-easy-to-spot-meaning)

In unit tests, you want to be able to quickly tell which data and doubles are important, and which ones are only there to keep the code from crashing. Support this by giving things that have no meaning obvious names and values, for example:

DATA(alert\_id) = '42'. " well-known meaningless numbers

DATA(detection\_object\_type) = '?=/"&'. " 'keyboard accidents'

CONSTANTS some\_random\_number TYPE i VALUE 782346. " revealing variable names

Don't trick people into believing something connects to real objects or real customizing if it doesn't:

" anti-pattern

DATA(alert\_id) = '00000001223678871'. " this alert really exists

DATA(detection\_object\_type) = 'FRA\_SCLAIM'. " this detection object type, too

CONSTANTS memory\_limit TYPE i VALUE 4096. " this number looks carefully chosen

**Make it easy to spot differences**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Data](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-data) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#make-it-easy-to-spot-differences)

exp\_parameter\_in = VALUE #( ( parameter\_name = '45678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789END1' )

( parameter\_name = '45678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789END2' ) ).

Don't force readers to compare long meaningless strings to spot tiny differences.

**Use constants to describe purpose and importance of test data**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Test Data](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#test-data) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-constants-to-describe-purpose-and-importance-of-test-data)

CONSTANTS some\_nonsense\_key TYPE char8 VALUE 'ABCDEFGH'.

METHOD throws\_on\_invalid\_entry.

TRY.

" when

cut->read\_entry( some\_nonsense\_key ).

cl\_abap\_unit\_assert=>fail( ).

CATCH /clean/customizing\_reader\_error.

" then

ENDTRY.

ENDMETHOD.

**Assertions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assertions)

**Few, focused assertions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Assertions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assertions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#few-focused-assertions)

Assert only exactly what the test method is about, and this with a small number of assertions.

METHOD rejects\_invalid\_input.

" when

DATA(is\_valid) = cut->is\_valid\_input( 'SOME\_RANDOM\_ENTRY' ).

" then

cl\_abap\_unit\_assert=>assert\_true( is\_valid ).

ENDMETHOD.

Asserting too much is an indicator that the method has no clear focus. This couples productive and test code in too many places: changing a feature will require rewriting a large number of tests although they are not really involved with the changed feature. It also confuses the reader with a large variety of assertions, obscuring the one important, distinguishing assertion among them.

" anti-pattern

METHOD rejects\_invalid\_input.

" when

DATA(is\_valid) = cut->is\_valid\_input( 'SOME\_RANDOM\_ENTRY' ).

" then

cl\_abap\_unit\_assert=>assert\_true( is\_valid ).

cl\_abap\_unit\_assert=>assert\_not\_initial( log->get\_messages( ) ).

cl\_abap\_unit\_assert=>assert\_equals( act = sy-langu

exp = 'E' ).

ENDMETHOD.

**Use the right assert type**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Assertions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assertions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#use-the-right-assert-type)

cl\_abap\_unit\_assert=>assert\_equals( act = table

exp = test\_data ).

Asserts often do more than meets the eye, for example assert\_equals includes type matching and providing precise descriptions if values differ. Using the wrong, too-common asserts will force you into the debugger immediately instead of allowing you to see what is wrong right from the error message.

" anti-pattern

cl\_abap\_unit\_assert=>assert\_true( xsdbool( act = exp ) ).

**Assert content, not quantity**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Assertions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assertions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assert-content-not-quantity)

assert\_contains\_exactly( actual = table

expected = VALUE string\_table( ( `ABC` ) ( `DEF` ) ( `GHI` ) ) ).

Don't write magic-number-quantity assertions if you can express the actual content you expect. Numbers may vary although the expectations are still met. In reverse, the numbers may match although the content is something completely unexpected.

" anti-pattern

assert\_equals( act = lines( log\_messages )

exp = 3 ).

**Assert quality, not content**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Assertions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assertions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assert-quality-not-content)

If you are interested in a meta quality of the result, but not in the actual content itself, express that with a suitable assert:

assert\_all\_lines\_shorter\_than( actual\_lines = table

expected\_max\_length = 80 ).

Asserting the precise content obscures what you actually want to test. It is also fragile because refactoring may produce a different but perfectly acceptable result although it breaks all your too-precise unit tests.

" anti-pattern

assert\_equals( act = table

exp = VALUE string\_table( ( `ABC` ) ( `DEF` ) ( `GHI` ) ) ).

**Use FAIL to check for expected exceptions**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Assertions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assertions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#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**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Assertions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assertions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#forward-unexpected-exceptions-instead-of-catching-and-failing)

METHODS reads\_entry FOR TESTING RAISING /clean/some\_exception.

METHOD reads\_entry.

"when

DATA(entry) = cut->read\_something( ).

"then

cl\_abap\_unit\_assert=>assert\_not\_initial( entry ).

ENDMETHOD.

Your test code remains focused on the happy path and is therefore much easier to read and understand, as compared to:

" anti-pattern

METHOD reads\_entry.

TRY.

DATA(entry) = cut->read\_something( ).

CATCH /clean/some\_exception INTO DATA(unexpected\_exception).

cl\_abap\_unit\_assert=>fail( unexpected\_exception->get\_text( ) ).

ENDTRY.

cl\_abap\_unit\_assert=>assert\_not\_initial( entry ).

ENDMETHOD.

**Write custom asserts to shorten code and avoid duplication**

[Clean ABAP](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#clean-abap) > [Content](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#content) > [Testing](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#testing) > [Assertions](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#assertions) > [This section](https://github.com/SAP/styleguides/blob/master/clean-abap/CleanABAP.md#write-custom-asserts-to-shorten-code-and-avoid-duplication)

METHODS assert\_contains

IMPORTING

actual\_entries TYPE STANDARD TABLE OF entries\_tab

expected\_key TYPE key\_structure.

METHOD assert\_contains.

TRY.

actual\_entries[ key = expected\_key ].

CATCH cx\_sy\_itab\_line\_not\_found.

cl\_abap\_unit\_assert=>fail( |Couldn't find the key { expected\_key }| ).

ENDTRY.

ENDMETHOD.

Instead of copy-pasting this over and over again.