How to write an ATC Check

A brief introduction.

EXTERNAL

The ATC is an important framework for carrying out static checks of ABAP source code and ABAP dictionary objects.

A major use case is to carry out these static checks remotely: The checks are supposed to run on a central test system that gathers all necessary information remotely from the systems on which the objects to be tested are located. The remote scenario allows a check system running the newest release to run its checks against a system running an older release.

In the following, you will learn how to write your own check to analyze source-code and how to ensure it is able to be carried out remotely.

In contrast to earlier manuals, this brief introduction focuses on describing checks inheriting from `CL_CI_TEST_ABAP_COMP_PROCS`. One prominent advantage of the novel infrastructure is the direct link between the source-code based scan output and corresponding compiler symbols. On page 4, the different check super classes are introduced and compared so that check authors can decide which class to use.

TABLE OF CONTENTS

PRELIMINARIES

Please note that this guide assumes you are working with a system with SAP release 7.52 or higher. Some of the features and specifics described work differently or are not present at all in lower releases.

CREATING A CHECK CLASS

The first step is to create a class that the framework will recognize as a new check. You need to create a class that inherits from any class that is a subclass of `CL_CI_TEST_ROOT`. Alternatively, create a class that implements the interface `IF_CI_TEST`. By convention, check classes start with `CL_CI_TEST` (or `ZCL_CI_TEST` in customer namespace). By convention, check classes start with `CL_CI_TEST` (or `ZCL_CI_TEST` in customer namespace). Note that result classes conventionally start with `CL_CI_RESULT`, so it is a good idea to use a suffix for your check class that does not exceed the maximum of 30 characters when appended to `CL_CI_RESULT`.

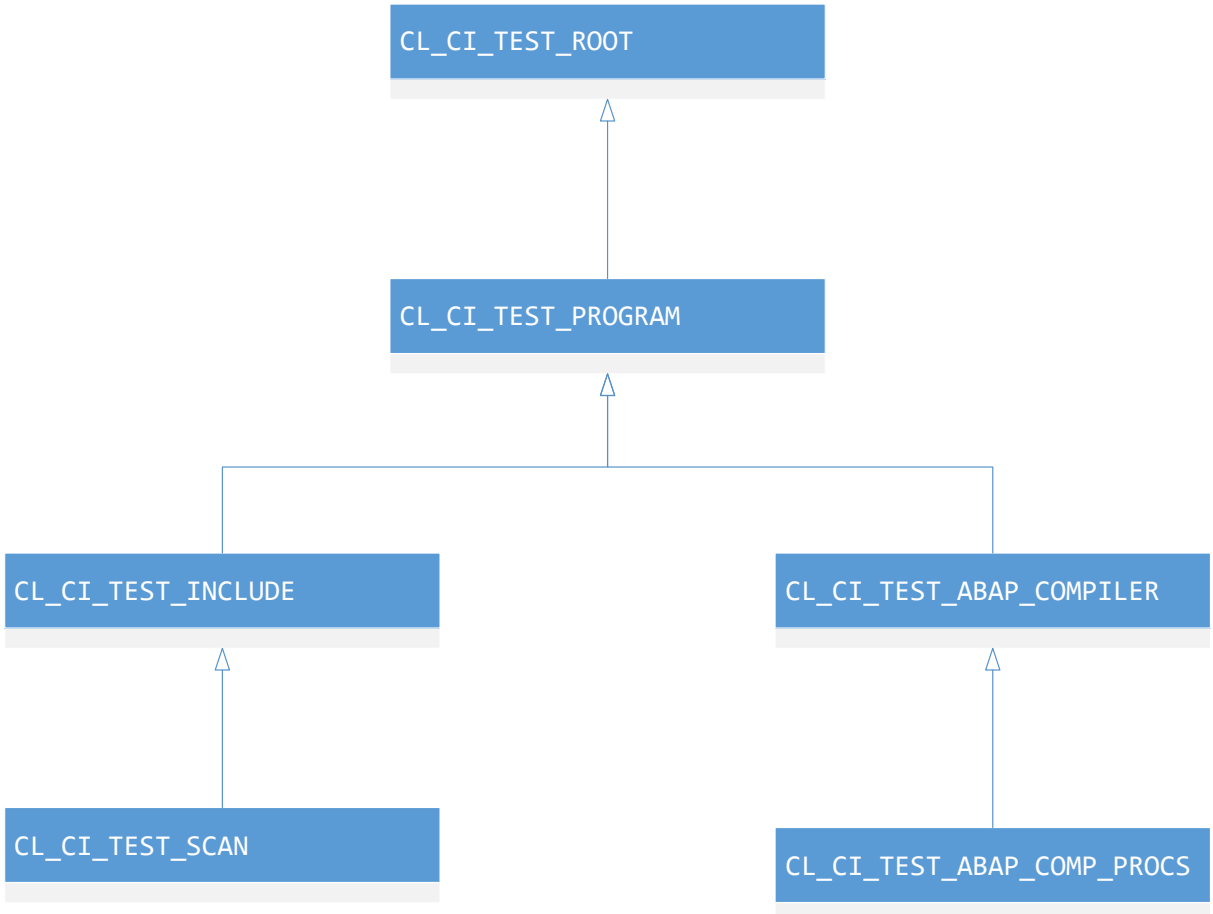

Figure 1: UML diagram of check classes. All test classes must inherit from `CL_CI_TEST_ROOT`. To add functionality to the infrastructure, tests may inherit from `CL_CI_TEST_SCAN` or `CL_CI_TEST_ABAP_COMP_PROCS` to analyze ABAP source code. The class `CL_CI_TEST_ABAP_COMP_PROCS` provides the latest infrastructure which includes scan results, refined scan results (called `PROC_DEFS` in the class) and compiler symbol information.

When asked to carry out your check, the framework will instantiate an object of your check class and then call its method `RUN`. Therefore, the main logic of your check needs to be implemented in this method. The way this implementation proceeds depends on what kind of check exactly you have. Your three main options are:

1. A subclass of `CL_CI_TEST_ABAP_COMP_PROCS`
2. A subclass of `CL_CI_TEST_ROOT`

3. A subclass of CL_CI_TEST_SCAN

While a lot of implementation details depend on which of these options you have chosen, there are a few things that always work in the same manner.

1. Checks based on CL_CI_TEST_ABAP_COMP_PROCS

Checks based on CL_CI_TEST_ABAP_COMP_PROCS are meant to analyze tokenized ABAP source code. The paradigm in this case is to partition the code into programming blocks – simply called “procs”, which is short for “procedure” – and analyze the code proc by proc. If the main goal of your check is to analyze ABAP statements, then this is the recommended choice.

A programming block is a logically coherent unit of source code. In the context of ABAP Objects, this mainly refers to the declaration and implementation blocks of classes, with each implementation block itself split up into one block for each method implementation. For reports, a programming block is basically equivalent to an event block, i.e. the set of statements following a START-OF-SELECTION, AT-SELECTION-SCREEN or any other event reporting keyword. If a report contains multiple START-OF-SELECTION words, this will lead to multiple programming blocks in the CL_ABAP_COMP_PROCS output, even though logically these form a single set of statements all executed when the event occurs.

2. Checks based on CL_CI_TEST_ROOT

Any ATC check must inherit from CL_CI_TEST_ROOT to be properly detected by the framework. For checks that examine ABAP source code, enhanced source information is provided when inheriting from CL_CI_TEST_ABAP_COMP_PROCS.

We generally do not recommend directly inheriting from CL_CI_TEST_ROOT. However, if your test does not check ABAP source code and there is no more suitable superclass for the objects you want to check, this is a valid option. If you are writing several checks that are based on the same information, consider writing a common superclass that encapsulated the information gathering.

3. Checks based on CL_CI_TEST_SCAN

With the advent of CL_CI_TEST_ABAP_COMP_PROCS, there is generally no reason to rely on the raw output of SCAN ABAP-SOURCE that CL_CI_TEST_SCAN provides. However, there are some edge cases, such as checks that are intended to process arbitrary comments in the source code, where the SCAN architecture is unavoidable. There already exists a brief description on how to write checks in the CL_CI_TEST_SCAN framework on the SCN wiki (link to PDF) and the SAP blog community.

Moreover, the book "SAP Code Inspector" published by Galileo Press contains relevant information for checks based on CL_CI_TEST_SCAN (ISBN: 978-3-8362-1706-4).

IMPLEMENTING YOUR CHECK

To obtain a check that can be executed, there are a few minimal implementation steps required so that the framework knows how to execute your check and display its results:

Setting up attributes of your check

The initialization of the important attributes of your check is done in the check class’s instance constructor. You should call the constructor of its superclass, if applicable, and then adjust the attributes as appropriate for your check. A typical constructor might look like this:

```
METHOD constructor .

  super->constructor( ).
  description = 'My class description'(001).
  category    = 'CL_CI_CATEGORY_MY_CATEGORY'.
  position    = '001'.
  version     = '000'.
  has_attributes      = abap_true.
  attributes_ok       = abap_true.
  has_documentation   = abap_true.
  remote_enabled      = abap_true.
  remote_rfc_enabled  = abap_true.
  uses_checksum       = abap_true.
  check_scope_enabled = abap_true.
```

- `DESCRIPTION` contains a short text description of the purpose of your check, which will be displayed in the SCI transaction when creating a check variant.
- `CATEGORY` is the technical name of the category (e.g. performance checks, S/4HANA readiness checks, etc.) your check falls into. A category is technically a class that inherits from `CL_CI_CATEGORY_ROOT`. To get the technical names of an already existing category, go to the check variant menu in the SCI transaction and choose the menu item *Checkvariant->Display->Technical Names*.
- `POSITION` is a numeric string that controls at which position your check is displayed within its category. The topmost check will be the one with the lowest (positive) value for position, the bottommost the one with the largest. If two checks have the same value for position, it is undefined which of them will be displayed first.
- `VERSION` is a numeric string that allows both you and the framework to identify if a check has changed substantially. Whenever you make an edit to your check that changes it in an incompatible way, you should increase this number. Check variants that include a test whose version has changed since it was created cannot be executed anymore. Consequently, the version should not be changed too often since check variants need to be redefined after each version change. But it is a advisable to increase the test version if the results of the check are significantly different. In that case, increasing the version forces all users to re-evaluate whether they want to run the check.
- `HAS_ATTRIBUTES` informs the framework if your check has attributes that a user can or must set before running it.
- `HAS_DOCUMENTATION` informs the framework if your check has attached documentation that can be displayed. You should always document your check! This documentation should be created via the SE61 transaction, see the section “Creating documentation” below.
- `REMOTE_ENABLED` allows to check with an outdated scenario (which uses a push approach rather than a pull approach by manually uploading the extracted source code to be checked). If your check is based `CL_CI_TEST_ABAP_COMP_PROCS` and is the flag `REMORE_RFC_ENABLED` is set to `ABAP_TRUE`, you can also set `REMOTE_ENABLED` to `ABAP_TRUE`.
- `REMOTE_RFC_ENABLED` indicates that your check can be carried out remotely. Note that it is your responsibility to ensure that your check can run remotely if you set these attributes as true – the attributes themselves are merely of an organizational nature and do not enable your test to run remotely! Additionally, beware that these attributes might be inherited from the superclass if you do not set them yourself, even if the classification does not apply to your check.
- `USES_CHECKSUM` means that your check will generate a checksum for each finding to identify its location. This is relevant, for instance, when you want to create ATC exemptions for a finding: The checksum is used to ascertain that the environment of the finding has not changed compared to the state when the exemption was issued.
- `CHECK_SCOPE_ENABLED` The default setting should be `CHECK_SCOPE_ENABLED = ABAP_FALSE`. It indicates that SAP objects cannot be checked in customer systems. If `CHECK_SCOPE_ENABLED = ABAP_TRUE`, modified SAP objects can be checked and findings are displayed if they are considered to be linked with the modification. To report findings related to modified SAP objects, the finding must properly report the origin of its findings, see the explanation of `INFORM` below.

Attributes you do not set in the constructor are inherited from its superclasses, if any of them sets the attribute in question in its own constructor. If no superclass sets an attribute, it defaults to its initial value.

Defining the messages of your check

The second important thing you need to do in the instance constructor is to create and register the messages your check will output when it encounters a finding. These messages are stored in the instance attribute SCIMESSAGES, and you register them simply by inserting them into this table. Here is a sample insertion of a message:

```
INSERT VALUE scimessage( test = myname
                         code = c_code_terrible_error
                         kind = c_error
                         text = 'This is a terrible mistake!'(100)
                         pcom = c_pcom_is_alright
                         pcom_alt = '' )
       INTO TABLE scimessages .
```

- TEST is simply the name of your check class
- CODE is a 10-character code that encodes the type of finding. It is highly recommended that you do not pass a literal here but instead define a reusable constant containing your code for this message. The code identifies the relevant message for a finding. It forms the link between finding and the displayed message text.
- KIND is the priority of your finding, encoded in a single character. You can choose between an error, a warning or simply information. These are represented by the characters E, W and I, respectively, but CL_CI_TEST_ROOT also already provides you with the predefined constants C_ERROR, C_WARNING and C_NOTE.
- TEXT is the short text that is displayed as the description of a finding. It is highly recommended to use text pool elements for the description so that the text can be translated into other languages.
- PCOM and PCOM_ALT are 20-character codes that can be used to suppress the finding via pseudo comments. The framework will first look for the occurrence of the comment in PCOM and then for the one in PCOM_ALT. The naming convention is that pseudo comment codes always start with "CI_" to distinguish them from the obsolete pseudo comments for e.g. SLIN or the unit test framework. Please note that the "#EC (that is part of the actual pseudo comment) should be excluded from the PCOM and PCOM_ALT attributes if you are inheriting from CL_CI_TEST_ABAP_COMP_PROCS or CL_CI_TEST_SCAN. If you are directly inheriting from CL_CI_TEST_ROOT, the framework does not automatically check for pseudo comments, even if you supply them via INFORM( ).

The definition of the constant passed as the CODE parameter deserves special attention: If you want to display documentation specific to each of these codes, you *must* name the constant the same as its content, or the same as its content with MCODE_ prepended. For example, the constant holding the error code ERROR must be named ERROR or MCODE_ERROR.

The RUN method

The RUN method is what is called by the framework when it is told to execute your check. It is called once for each object to be checked. If you have initialization or finalization routines that should be executed only once per test run – and not per object – then you can implement these in redefinitions of the RUN_BEGIN and RUN_END methods. If your check is not meant for source code analysis, then you can skip the rest of this section, since it focuses on particular patterns to be followed when inheriting from CL_CI_TEST_ABAP_COMP_PROCS.

The start of the standard RUN method of CL_CI_TEST_ABAP_COMP_PROCS looks like this:

```
METHOD run.
  DATA:
    l_refs   TYPE t_infos.

  IF get( ) = abap_false.
    RETURN.
  ENDIF.

  analyze_start( EXPORTING p_ref_check = ref_check
                 IMPORTING p_refs      = l_refs ).
```

The GET method gathers all the data needed for the analysis, and returns ABAP_FALSE value if any part of this gathering fails. If you want to gather different or additional data compared to your superclass, then you need to redefine the GET method, too. In that case, you should always call SUPER->GET( ) at the beginning of your redefinition. Note that if the only information your check uses is already provided by CL_CI_TEST_ABAP_COMP_PROCS, it is automatically able to be executed remotely.

ANALYZE_START is where the bulk of the check starts. The class now initiates the analysis of the individual procs. In general, most checks you want to write will need to redefine the ANALYZE_PROC method to implement their logic. You can and should assume that the code you write in ANALYZE_PROC will run at least once over every proc in the code you are checking, i.e. over the entire source code. The analysis will proceed in the order in which the programming blocks occur in the source code.

The ANALYZE_PROC method

This method is the main part of your source code analysis so it is important you use its parameters correctly. Its signature is the following:

```
METHODS analyze_proc
  IMPORTING
    p_proc TYPE cl_abap_comp_procs=>t_proc_entry
    p_from TYPE i DEFAULT 1
    p_proc_ids TYPE t_proc_ids OPTIONAL
    VALUE(p_level) TYPE i DEFAULT level
    p_params TYPE t_params OPTIONAL
  CHANGING
    p_collect TYPE t_collect.
```

- P_PROC contains the proc that is currently to be analyzed.
- P_FROM contains the number of the statement from which the analysis is supposed to begin (this is e.g. useful if you aborted the analysis of this proc earlier and now want to resume it where you left off)
- P_PROC_IDS contains a list of T_PROC_ID that you can use for various purposes; it has no dedicated purpose by default. !!!Check what this actually *is*!!!
- P_LEVEL contains the number of levels left until the analysis is aborted, e.g. a value of 5 means that the analysis will resolve calls of methods and other reusable components up to a depth of five before returning with an exception.

- P_PARAMS contains parameters that might have been passed to the proc
- P_COLLECT contains various information that persists at a level coarser than individual procs, e.g. it contains a call stack.

The main information you likely want to use in your source code analysis is contained in the proc entry itself. The most important field of the type T_PROC_ENTRY is STMTS, which contains the table of all tokenized statements within the current proc. The statement type is one of the most important types you will encounter in this context:

```
BEGIN OF t_stmt,
    keyword       TYPE string,
    tokens        TYPE t_tokens,
    comments      TYPE t_comments,
    include       TYPE program,
    line          TYPE i,
    column        TYPE i,
    non_buf_db_op TYPE abap_bool,
    idx           TYPE i,
    links_origins TYPE SORTED TABLE OF i WITH UNIQUE KEY table_line,
    link_blocks   TYPE i,
END   OF t_stmt .
```

-

- KEYWORD is the keyword associated with the statement. This is often, but not always, the first token of the statement, but there are statements that do not begin with a keyword. Most relevant among these are ordinary assignments, which implicitly begin with the superfluous keyword COMPUTE, and static functional method calls that do not assign a returning parameter, which implicitly begin with the fake keyword +CALL_METHOD
- TOKENS is simply the table of actual tokens the statement consists of
- COMMENTS is the table of comments that belong to the statement
- INCLUDE is the name of the include the statement appears in
- LINE and COLUMN are the statement's position within its include

In many use cases, a large part of your ANALYZE_PROC method will consist of iterating over the statements of the current proc and detecting those you are interested in by examining their keywords, e.g. via a multi-pronged "CASE <STMT>-KEYWORD" statement. If what you want to do does not fit within a few lines of code for each WHEN statement, it is highly advisable to extract such code into its own methods, since otherwise the ANALYZE_PROC method rapidly becomes cluttered and difficult to read.

Reporting a finding

CL_CI_TEST_ABAP_COMP_PROCS uses a two-step approach to findings. If you wish to report finding while not inheriting from this class, please skip to the next section. During the analysis, each finding is first registered via the ADD_INFO method. After the analysis and additional optional processing of the registered findings, the findings are reported to the end user as in all other cases.

These findings are stored in the instance attribute infos and returned as the P_REF parameter of the ANALYZE_START method. The standard implementation of run in CL_CI_TEST_ABAP_COMP_PROCS simply iterates over all findings and calls inform as described above on them, so if you want to change the information that's passed onto inform or change the way the findings are displayed, you will need to redefine the run method in your check class.

The signature of the ADD_INFO method is as follows:

```
METHODS add_info
  IMPORTING
    p_include TYPE program OPTIONAL
    p_line TYPE i OPTIONAL
    p_column TYPE i OPTIONAL
    p_kind TYPE sci_errc
    p_source TYPE csequence OPTIONAL
    p_name TYPE csequence OPTIONAL
    p_stack TYPE t_stack_entries OPTIONAL
    p_stack_of_var TYPE t_stack_entries OPTIONAL
    p_proc_pos TYPE scis_proc_pos OPTIONAL
    p_no_moves TYPE sci_no_moves OPTIONAL
    p_comments TYPE cl_abap_comp_procs=>t_comments OPTIONAL
    p_program TYPE program OPTIONAL
    p_checksum TYPE sci_crc64 OPTIONAL
    p_proc TYPE cl_abap_comp_procs=>t_proc_entry OPTIONAL
    p_stmt TYPE cl_abap_comp_procs=>t_stmt OPTIONAL
    p_stmt_supplied TYPE abap_bool DEFAULT abap_true.
```

- Once again, P_INCLUDE, P_LINE AND P_COLUMN indicate the position of the finding in the source code.
- P_KIND denotes the 10-character error code you have already used in the instance constructor.
- P_STACK and P_STACK_OF_VAR are two stacks you can attach to your finding. The default expectation is that P_STACK contains a call stack that leads from the tested program to the actual finding.
- P_COMMENTS contains a list of comments belonging to the finding. The main purpose is to store all pseudo comments related to the finding to examine them in a next step for pseudo comments that would suppress this finding.
- P_CHECKSUM is a checksum generated from the finding's surroundings to recognize in later runs of the same test whether the surrounding code (or even the location of the finding itself) was changed.
- P_PROC and P_STMT are again indicators of the position of the finding. The default expectation is that a finding corresponds to a single statement. If you have findings that do not relate to a specific statement and therefore do not want to pass a statement as an argument, you need to set the next parameter, P_STMT_SUPPLIED, to ABAP_FALSE.

Reporting a finding directly via INFORM

If you do not want to sort your findings, or if you are not implementing a subclass of CL_CI_TEST_ABAP_COMP_PROCS you can directly call the INFORM method to report a finding. Findings reported this way will show up in whatever UI the end user is using to display the result. The INFORM method is redefined in CL_CI_TEST_ABAP_COMP_PROCS and behaves slightly differently from the general case:

```
methods INFORM
  importing
    P_SUB_OBJ_TYPE type TROBJTYPE optional
    P_SUB_OBJ_NAME type SOBJ_NAME optional
    P_POSITION type INT4 optional
    P_LINE type TOKEN_ROW optional
    P_COLUMN type TOKEN_COL optional
    P_ERRCNT type SCI_ERRCNT optional
    value(P_KIND) type SYCHAR01 optional
    P_TEST type SCI_CHK
    P_CODE type SCI_ERRC
    P_SUPPRESS type SCI_PCOM optional
    P_PARAM_1 type CSEQUENCE optional
    P_PARAM_2 type CSEQUENCE optional
    P_PARAM_3 type CSEQUENCE optional
    P_PARAM_4 type CSEQUENCE optional
    P_INCLSPEC type SCI_INCLSPEC optional
    P_DETAIL type XSTRING optional
    P_CHECKSUM_1 type INT4 optional
    P_COMMENTS type T_COMMENTS optional
    P_FINDING_ORIGINS type CL_CI_SCAN=>T_ORIGIN_TAB optional .
```

- `P_SUB_OBJ_TYPE` and `P_SUB_OBJ_NAME` are the location of the finding. Usually you do not need to worry about these parameters since using the `ADD_INFO` method described below will automatically generate the correct values. If you need to set this information manually, though, you need to be rather diligent about the value of `P_SUB_OBJ_NAME`, e.g. for a method of a class this will contain the name of the automatically generated include this method appears in instead of the class name.
- `P_LINE` and `P_COLUMN` specify the position of the finding within the sub-object.
- `P_ERRCNT` is an obsolete1 parameter provided for backwards-compatibility. Don't use it.
- `P_KIND` is an obsolete parameter with which you can override the priority set for the finding's error code passed in `P_CODE`. Usually you should not use this, but use different error codes for findings with different priorities instead.
- `P_TEST` is the name of the check that has raised the finding, i.e. you should only in exceptional cases pass something different than your check class's own name here.
- `P_CODE` is one of the 10-character codes you defined in the instance constructor
- `P_SUPPRESS` is an obsolete parameter to which you can pass a pseudo comment that can suppress this finding. You should not use this, but use the `PCOM` and `PCOM_ALT` fields of the corresponding error code's entry in `SCIMESSAGES` instead.
- `P_PARAM_<N>` are parameters for use with error codes whose messages contain placeholders of the format &<n>. The method will automatically substitute &<n> with the content of `P_PARAM_<N>` when displaying the findings.
- `P_DETAIL` is where you put any additional information not covered by other parameters. EXPORT any additional information to this raw byte string. Note that unless you also write code that reads this information at another place, e.g. the in result class discussed below, it will never be accessed by the default behavior of the framework.
- `P_CHECKSUM_1` is a checksum by which you can recognize changes to the finding's surroundings. If you want to manually compute a checksum, you should use the utility class `CL_CI_CHECKSUM_PROVIDER`, but the inform method of `CL_CI_TEST_ABAP_COMP_PROCS` also already does this automatically for you if you do not pass a value here.

- P_COMMENTS is a table of pseudo comments which is checked against the pseudo comments defined in the SCIMESSAGES table of the test. The comments are provided in the same tabular format in the PROC_DEFS[X]-STMTS[Y]-COMMENTS. Note that due to restrictions in the parser the framework is not able to process more than one comment per line.
  Caution: If your test class is inheriting from CL_CI_TEST_SCAN, this field is obsolete and has no effect, but the position specification is used to search for pseudo comments instead in the SCAN output.
- P_FINDING_ORIGINS is a table related to the classification of the code and is automatically generated from the statement and stack of a finding reported via ADD_INFO. In detail, it is a table of all classifications (SAP code, customer code, automatically generated code,…) applying to code involved in this finding. Usually, it should contain the classifications applying to all statements involved. The classification of a statement is encoded in its LINKS_ORIGINS component, which is the index for the classification in the ORIGINS table component of the proc the statement belongs to.
- In the standard configuration, the end user will see the finding together with the message you defined for this error code in the instance constructor and can navigate to the position specified by the SUBOBJECT, LINE and COLUMN. To replace this with custom behavior, you need to implement a custom result class, see below.

Verification of test prerequisites

Your check may require certain prerequisites in order to be executed correctly. If these only depend on the system and not on the specific objects used, you can emit these messages not as a finding, but as a verification message by implementing the method VERIFY_TEST. If you emit a verification message with a priority higher than that of an information – i.e. a message with the attribute KIND set to CL_CI_TEST_ROOT=>C_ERROR or CL_CI_TEST_ROOT=>C_WARNING – then the inspection will terminate and no check in the current check variant will be executed at all.

If you do not wish to abort the execution of the whole inspection, but you still want to notify the user of these problems, then the ATC also has the concept of a *tool failure* finding, which will not show up in the ordinary results of an ATC run but instead in its own place. To mark a finding as a tool failure, the entry of its message code in the SCIMESSAGES table needs to have the CATEGORY attribute set to one of the following constants (all are members of CL_CI_TEST_ROOT):

Constant	Meaning
C_CAT_NOT_EXECUTABLE_AT_ALL	The check is not executable at all in the current system, no object will be checked by it.
C_CAT_SPEC_OBJ_NOT_CHECKABLE	The check is not able to check this specific object, other objects will be checked.
C_CAT_CHECK_PART_NOT_EXEC	A part of the check is not executable in the current system, but its other part will correctly check all objects.

Activating your check

Whichever option you chose, at some point after you have created an initial implementation you will very likely want to test your check by letting it run over some test code. By default, the ATC detects your check internally but will not display it in the list of available checks when creating a new check variant. To make your check available there, start the SCI transaction and go to the *Code Inspector->Management of->Tests* menu item (or press *Shift+Alt+F5*). You will see a list of all available check classes. Find yours in this list and check the check box in front of its name, then save the selection. Your check should now be available during the definition of a new check variant, with its category and description determined by what you wrote earlier in the instance constructor.

If you want to disable your check again, visit the same management screen and uncheck the check box again.

Allowing configurable settings for your check

When creating a check variant, the SCI transaction offers a “properties” button for checks that have the instance attribute HAS_ATTRIBUTES set to ABAP_TRUE. To not generate runtime errors in the transaction, you should only set this attribute to true after you have implemented the methods QUERY_ATTRIBUTES, GET_ATTRIBUTES and PUT_ATTRIBUTES belonging to the IF_CI_TEST interface.

The latter two methods typically just require you to export respectively import the configurable attributes of your check to/from a byte string that is passed in their sole parameter P_ATTRIBUTES.

The QUERY_ATTRIBUTES method, however, implements the dialog that is shown to the user so that they can configure the attributes manually.

As a first step, you need to define a table of type SCI_ATTTAB that will hold one row for each configurable attribute. Of the column, three are relevant for you: ref expects a reference to the parameter that should be configured, text is the text shown to the user in the configuration dialog and kind is a one-character flag indicating the type of the attribute. If you leave it initial, the user will see a standard text box to enter arbitrary strings for this attribute. To present a checkbox to the user, i.e. indicate a Boolean type, set the flag to ‘C’. To allow the user to enter a table instead of single values, set the flag to ‘L’. A typical statement for an attribute might look like this:

```
INSERT VALUE #( ref = REF #( level ) text = 'External analysis depth'(002)
                kind = ' ' ) INTO TABLE attributes.
```

Note that the logic that automatically generates the settings dialog from this table only works for DDIC types, i.e. the variable whose reference you pass in REF must have a type from the dictionary, and not a locally defined type or a type from a type group.

In this case, level could be an instance attribute of the check class that controls how deep the check descends into method, function or perform calls into other programs.

The table summarizes different options for the field KIND:

KIND =	meaning
'G'	horizontal line as optical separator, the value of the TEXT field is displayed as heading, the REF field has no meaning. However, any reference may be passed.
'R'	radio button
'C'	checkbox
'S'	select option, the type of the reference passed should be a RANGE OF
'L'	List box (drop down), for parameters whose domain has fixed suggested values
''	text field allowing for text input

After filling the attribute table with all attributes, you want the user to be able to configure in this way, you need to display the configuration dialog to the user by calling the class method CL_CI_QUERY_ATTRIBUTES=>GENERIC, which has the following signature:

```
class-methods GENERIC
  importing
    P_NAME type SCI_CHK
    P_TITLE type C
    P_ATTRIBUTES type SCI_ATTTAB
    P_MESSAGE type C optional
    P_DISPLAY type FLAG
  returning
    value(P_BREAK) type SYCHAR01 .
```

- P_NAME is the name of your check class
- P_TITLE is the title the configuration dialog should display
- P_ATTRIBUTES is your table of attributes you filled in the first step
- P_MESSAGE is a message that is displayed to the user in case their entries are erroneous
- P_DISPLAY is a parameter of QUERY_ATTRIBUTES that should simply be passed on

The return value of this function becomes ABAP_TRUE if the user cancels the dialogue. Be aware that calling this method will overwrite the contents of whatever was in the parameters referenced by the rows of P_ATTRIBUTES, so it is not recommended to directly pass references to your instance attributes unless you are fine with potentially arbitrary user input ending up in them.

Of course, if you did not pass references to the instance attributes but to local variables you need to write the content of the local variables to the instance attributes after GENERIC has finished without the user cancelling.

RESULTS AND RESULT CLASSES

If you want to output your results in a form different from the standard output, you will need to implement this output logic in its own dedicated result class inheriting from CL_CI_RESULT_ROOT. The naming convention is that the result class belong to a check class CL_CI_TEST_MYTEST is called CL_CI_RESULT_MYTEST, however this is not enforced. You need to register the result class in your test class by redefining the method GET_RESULT_NODE to return an instance of your new custom result class. However, this is rarely necessary, and in addition such result classes only affect the display of the results in the SCI transaction. If you are executing checks through the ATC – as is recommended – then these classes have no effect.

Navigation and stacks

If you passed a valid source code position when reporting your finding via INFORM, then navigation to the location of the finding works automatically in both SAPGUI and ADT. However, sometimes you might want to enable the user to additional locations in the source code – the archetypal example is when your finding is related to the structure of a call stack, and you want to display the call stack and allow the user to navigate to the source code position of the individual entries.

To achieve this, you should export a table of type SCIT_WB_NAVIGATION under the name WB_NAVIGATION to the details buffer of the finding. The row type SCIS_WB_NAVIGATION has the following components:

```
define structure scis_wb_navigation {
  object_type      : sci_wb_object_type;
  object_name      : sci_wb_object_name;
  enclosing_object : sci_wb_enclosing_object;
  description      : sci_wb_description;
  position         : sci_wb_position;
}
```

POSITION is a line number in the source code-like object given by the triple OBJECT_TYPE, OBJECT_NAME, ENCLOSING_OBJECT. In this case, OBJECT_TYPE and OBJECT_NAME refer to the include, and

ENCLOSING_OBJECT to the TADIR object the include belongs to. DESCRIPTION is the text that will be displayed as the text of the link that leads to the position specified by the other components.

UNIT TESTS FOR YOUR CHECK

Testing the functionality of your check manually at each step of the development process is time consuming and prone to errors. Fortunately, the ABAP Unit Test framework together with the class CL_CI_TEST_VERIFY provides a convenient way to automate the testing process. Your unit test classes should therefore inherit from this class.

Running your check during a unit test

CL_CI_TEST_VERIFY offers a run method that will carry out your check on a single ABAP dictionary object or program. Its signature is given below, but regardless of your specific use case you need to have created a *global* check variant that runs only your test and nothing else.

```
class-methods RUN
  importing
    P_VARIANT type SCI_CHKV
    P_OBJ_TYPE type TROBJTYPE
    P_OBJ_NAME type SOBJ_NAME
    P_OBJ_PARAMS type SCIT_OBJ_PAR optional
    P_NOSUPPRESS type SCI_NOSUP optional
    P_SYSID type SYST_SYSID optional
    P_DESTINATION type RFCDEST optional
    P_ALLOW_EXCEPTIONS type ABAP_BOOL default ABAP_FALSE
  raising
    CX_CI_CHECK_ERROR .
```

- P_VARIANT is the name of your global check variant as a string
- P_OBJ_TYPE and P_OBJ_NAME are the TADIR keys of the object you wish to check
- P_OBJ_PARAMS is an optional parameter in which you can pass the parameters the framework would normally determine by itself during an ordinary inspection, such as if the object to be checked is an SAP object, or which namespace it belongs to. For the kinds of allowed parameters refer to the definition of C_OBJ_PARAM_KINDS in CL_CI_OBJECTSET.
- P_NOSUPPRESS is a flag to deactivate the suppression of findings via pseudo comments for this check run.
- P_SYSID is the RFC source in case you want to test RFC functionality. The same caveats as for P_DESTINATION below apply.
- P_DESTINATION is the RFC destination in case you want to test RFC functionality. This is generally advised against for elementary unit testing since you make yourself dependent on the remote availability of the destination, but may be useful in some cases. If you do want to include remote executions of your check in your unit tests, consider making their failure tolerable so that the unavailability of the remote system does not prevent the execution of other tests and is not seen as a fatal error.
- P_ALLOW_EXCEPTIONS determines if the exceptions for a missing check variant or a missing object to be checked cause a failure of the unit test (this happens if this parameter is ABAP_FALSE) or the corresponding exceptions are merely propagated by the method.

Usually you only need to pass the required parameters to do useful unit testing. After this method has been called, the results of the check will be stored in the instance attribute RESULT_LIST of your check class.

Checking your findings against your expectations

The simplest check is of course to verify that you have the expected number of findings, i.e. that `LINES( RESULT_LIST )` has the expected value, but generally you will want to ensure all findings are also raised with the correct error codes and positions. `CL_CI_TEST_VERIFY` provides the check method for this purpose.

```
class-methods CHECK
  importing
    P_CODE type SCI_ERRC
    P_SOBJTYPE type TROBJTYPE
    P_SOBJNNAME type SOBJ_NAME
    P_LINE type I
    P_COL type I
    P_PARAM_1 type STRING optional
    P_PARAM_2 type STRING optional
    P_PARAM_3 type STRING optional
    P_PARAM_4 type STRING optional
    P_DETAIL type SCIT_DETAIL optional
    P_CHECKSUM1 type I optional
    P_TEST type SCI_CHK optional
  returning
    value(P_MESSAGE) type STRING .
```

You will certainly note that, not coincidentally, these parameters are the same as those for the `INFORM` method. Simply pass the values you expect for your findings to this method. If the actual finding matches the expected finding, the returned `P_MESSAGE` string will be empty, otherwise it will contain information about the mismatch, usually the data of the finding that was expected.

Pay attention to the unusually spelled parameter `P_SOBJNNAME` – preserved this way for backward-compatibility – as well as once again to the fact that this `SUBOBJECT` identifier refers to the full name of the actual include, not e.g. the class, a finding was reported in.

CREATING DOCUMENTATION

You can create documentation that will automatically be displayed in both the SCI transaction and the ATC, regardless of whether it is being used through SAPGUI or the ADT. There are two basic types of documentation, the documentation of the check itself and the documentation of each error code of a check. Note that you need to set the attribute `HAS_DOCUMENTATION` to `ABAP_TRUE` for any documentation to be displayed.

The documentation of the check itself is always created in the SE61 transaction by documenting the dummy class attribute `0000` of your check class. This documentation will appear when a user clicks on the blue information icon in the SCI transaction next to your check, and will also be displayed by the ATC for each finding of your check.

You can furthermore document each error code of your check by documenting class attributes in SE61 whose names agree with the literal values of your error code constants, not with these constants' names. Alternatively, you can document an attribute named `MCODE_<LITERAL>`. Note that this means that your error code constants must be named exactly like their values, or like their values with `MCODE_` prepended, otherwise SE61 will not allow you to create documentation – you can't document class attributes that do not exist. This documentation – if necessary – should contain e.g. the reason this finding is being displayed and advice on how to fix this specific finding.

You should consider the documentation of the check itself mandatory, since otherwise only you will be able to know what the check even is supposed to do. In contrast, documentation for individual error codes should only be considered in cases where the short text of the finding is not self-explanatory or where the course of action to fix the finding is not obvious. You do not need to explicitly document any pseudo comments that suppress a finding; the framework will automatically display up to two pseudo comments that can suppress the finding in its description.

Custom documentation access

You are not restricted to the behavior for defining and retrieving documentation outlined above. The result class of your check steers how documentation for specific findings is retrieved, the above is just the default behavior. If you want to display documentation in a different way, you should implement the method `GET_DOCU_FOR_TEST_CODE` in your result class.

ENSURING REMOTE ABILITY OF YOUR CHECK

ATC checks can analyze code on other SAP systems in the same landscape. The source code information – and any other local information a check needs – needs to be retrieved from the target system via RFC.

There is a series of blog entries on how to perform remote analysis via ATC once you have a remote-enabled check.

For your check to be able to run successfully in a remote scenario, you need to avoid any explicit dependence on local data of the check system. For instance, a common pitfall is to rely on reading the `TADIR` or other tables with object information – of course that will only get you information about the objects in your check system, not about the objects in the checked system. You should also be wary of using kernel methods to provide information, or using the RTTI (**R**un **T**ime **T**ype **I**nformation) functionality.

If you use only the information and methods provided by `CL_CI_TEST_ABAP_COMP_PROCS`, you can be sure that all relevant data is acquired from the checked system. However, if you want additional data that is not yet provided by this class, you currently need to write your own RFC function module that will make the necessary data available to your check class. An API/framework that will simplify creating data providers that are able to function in a remote scenario is currently in the works.

Of course, when you manually call an RFC module in your test, you need to ensure that the destination is correct. The destination is not directly passed to your check, but in an enriched format called the `SCR_SOURCE_ID`. The currently valid source ID is held globally in the class attribute `CL_CI_TEST_ROOT=>SRCID`. To translate this into an RFC destination, you need to use the method `CL_ABAP_SOURCE_ID=>GET_DESTINATION`.

www.sap.com/contactsap

THE BEST RUN SAP®