How to Build a New Check for the Code Inspector

Version 2.0

April 2004

Table of Contents

Code Inspector: How to build a new check

The Code Inspector is a tool for the static analysis of ABAP programs and other SAP repository objects. The Code Inspector framework consists of a test driver that defines, stores and executes the test tasks (‘inspections’), and an extendable set of checks. The framework’s architecture allows it that different checks can jointly use the prepared data of the examined object. All needed dissections of an object (for example the tokenising of the object) only have to be performed once. The objects to be checked are always specified by their object type (for example ‘CLAS’ for ABAP OO classes) and their name. Up to now, only ‘main’ objects are checked (whole function groups instead of single function modules for example).

A Code Inspector check is (the instance of) a global class that contains all the relevant information needed to define, perform, and document the check: the check attributes, the check routine, message texts, pseudo-comments to hide messages, and so on. Every new check class extends the existing hierarchy of Code Inspector data preparation classes and check classes.

To build a new Code Inspector check, you have to do the following:

- Create a new global class that extends one of the existing Code Inspector classes
- Implement the instance constructor
- Implement the actual check routine in the RUN method
- Implement method GET_MESSAGE_TEXT to output message texts
  Starting with SAP_BASIS 6.40 message texts can be defined in the instance constructor
- Include the new check into the check variant tree

Additionally, if your check has user parameters:

- Add user parameters as private class attributes to your check class
- Set flag HAS_ATTRIBUTES in the instance constructor
- Implement methods GET_ATTRIBUTES and PUT_ATTRIBUTES to get and write away attributes
- Implement method QUERY_ATTRIBUTES of interface IF_CI_TEST to define the Pop-Up for the parameter input

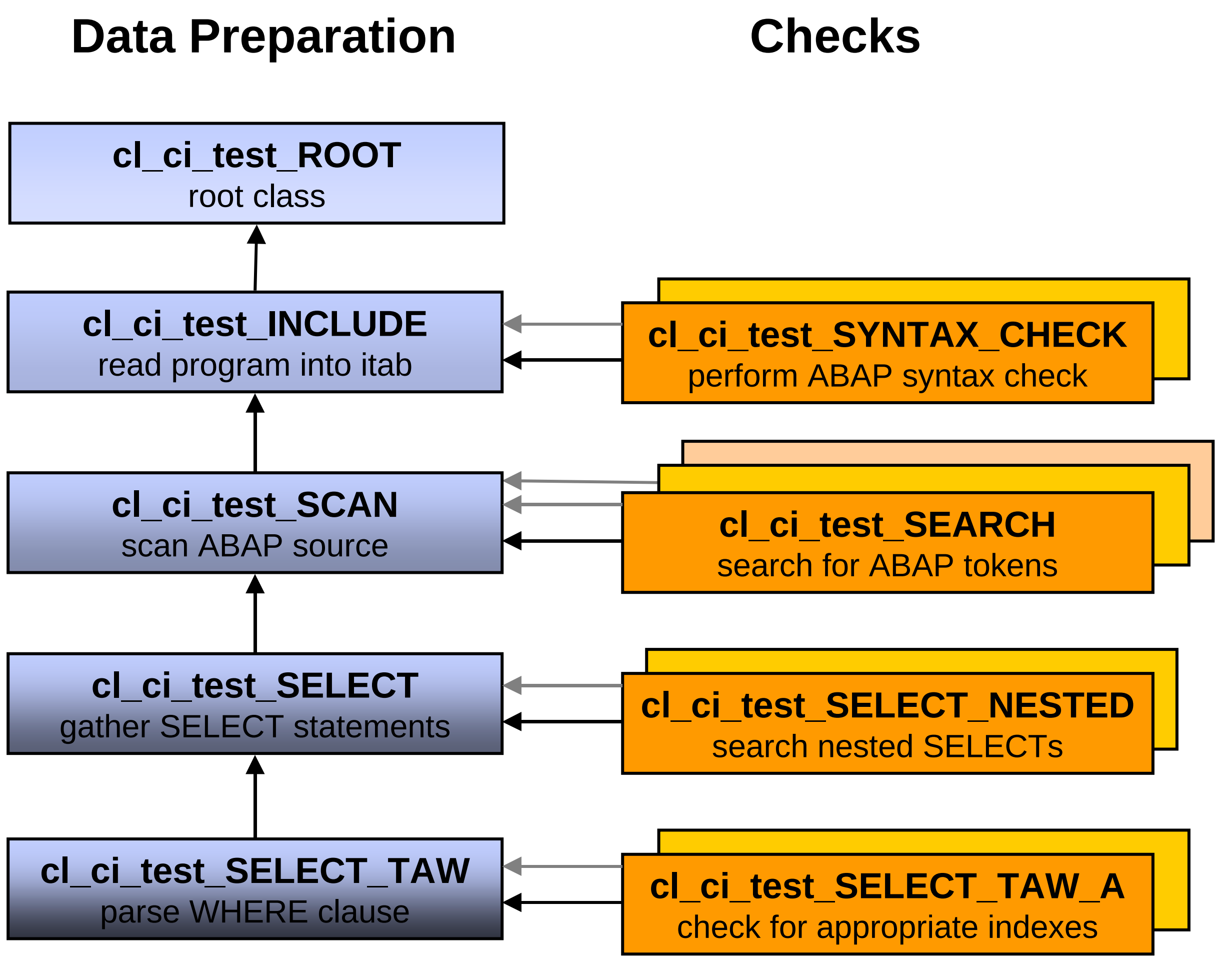
1 How to Create a New Code Inspector Check

1.1 Create a new global class in the Code Inspector class hierarchy

Every check is (the instance of) a global class that extends the class hierarchy with root **CL_CI_TEST_ROOT**. Depending on the data needed, a new check class extends the respective data preparation class of the hierarchy. A check that is based on the analysis of the source code tokens extends the class **CL_CI_TEST_SCAN**, since one of the static attributes of this class is a token table that will be filled at run time using the ABAP statement SCAN ABAP-SOURCE. If you as a check author cannot make use of any existing data preparation, your new check class has to extend the root class CL_CI_TEST_ROOT itself.[1]

The following picture might give you an idea of how data preparation classes and check classes are grouped in the Code Inspector class hierarchy:

[1] If you accomplish some ‘interesting’ data preparation, you should extend the Code Inspector class hierarchy by an (abstract) class, and implement the data preparation in the GET method of this class storing the data as static attributes (e.g. internal tables or class references). You and other users will then benefit from your work by being able to extend the data preparation class with new check classes. To improve encapsulation, the actual data preparation can be accomplished in a separate class that is not part of the CL_CI_TEST_ROOT hierarchy. The check class then only holds a reference to this class as a static attribute.



The specialisation of the data preparation increases from the top (CL_CI_TEST_ROOT) to the bottom in this figure.

Note that a data preparation class itself can also perform checks. For example, when a program-like object is part of an object set, but cannot be ‘found’ in the SAP system during the inspection run, class CL_CI_TEST_INCLUDE will raise a message. This check is not part of any selected check variant, but will be executed implicitly.

In package S_CODE_INSPECTOR you find a template for checks that make use of the scanned or tokenised ABAP source (CL_CI_TEST_SCAN_TEMPLATE) and one for more general checks (CL_CI_TEST_ROOT_TEMPLATE).

To create a new check class, you can either

- copy one of the templates
- create a new class with public instantiation and specify one of the data preparation classes of the Code Inspector class hierarchy as the ‘superclass’ of the new class
- copy any of the existing check classes in package S_CODE_INSPECTOR and modify it to your purpose

1.2 Attribute C_MY_NAME

Create a private constant attribute C_MY_NAME, type SEOCLSNAME in the class interface and initialise it with the name of the new check class.

1.3 Instance constructor of the new check class

Implement the instance constructor of the new check class. It should look like this:

```
method CONSTRUCTOR.

  SUPER->CONSTRUCTOR( ).
  DESCRIPTION = '<test description>'(000).    "required
  CATEGORY    = '<category_class>'.           "required
  VERSION     = '000'.                        "required
  ADD_OBJ_TYPE( <object_type_to_check> ).     "required
  POSITION    = '010'.                        "optional

endmethod.
```

- **`DESCRIPTION`** This description appears as text in the tree of the check variant
- **`CATEGORY`** The check category to which the new check will belong. The following check categories (which are also represented by global classes) already exist:
  - CL_CI_CATEGORY_PERFORMANCE
    with checks relevant for performance related issues
  - CL_CI_CATEGORY_SECURITY
    with checks relevant for program stability and security
  - CL_CI_CATEGOTRY_ABAP_COMPILER (before: CL_CI_CATEGORY_SLIN)
    with ABAP syntax check and extended syntax check
  - CL_CI_CATEGORY_SEARCH
    Contains search functions
  - CL_CI_CATEGORY_UNIVERSAL, CL_CI_CATEGORY_GENERAL
    For additional checks

  Class CL_CI_CATEGORY_TEMPLATE is a template for new check categories.
- **`VERSION`** When the attributes of a productive check are changed or new ones are added, this version number should be modified. This way the Code Inspector is able to identify old check variants and to invalidate them. They have then to be saved again with new attribute settings.
- **`ADD_OBJ_TYPE`** This method gathers those object types that the check is able to handle. Predefined object types are:
  - C_TYPE_PROGRAM (program-like objects: programs, module-pools, classes, interfaces, function groups, ...),
  - C_TYPE_INCLUDE (programs, ‘PROG‘),
  - C_TYPE_FUNCTION_GROUP (function groups, ‘FUGR‘),
  - C_TYPE_DDIC_TABLE (data dictionary tables, ‘TABL‘).
- **`POSITION`** Determines the relative position of a check in it’s category when the check variant tree is displayed.

Starting with SAP_BASIS 6.40, further attributes of the check class can be defined in the instance constructor: message texts, message priorities and pseudo-comments. In return, method GET_MESSAGE_TEXT is not needed any longer, and the call to trigger messages is simplified.

Include the following definition into the instance constructor:

```
define FILL_MESSAGE.
  clear SMSG.
  SMSG-TEST = C_MY_NAME.
  SMSG-CODE = &1.  "message code
  SMSG-KIND = &2.  "message priority
  SMSG-TEXT = &3.  "message text
  SMSG-PCOM = &4.  "pseudocomment
```

```
    INSERT SMSG INTO TABLE SCIMESSAGES.
  end-of-definition.
```

Single messages are then defined with:

```
*---------------- Check Message
  FILL_MESSAGE '0001' C_NOTE text-101 '"#EC MYPSEUDOCOM'.
```

Here ‘0001’ is the message code, C_NOTE stands for the message type (C_NOTE = information, C_WARNING = warning, C_ERROR = error message), text-101 is the message text (which can contain replacement characters &1…&4) and `"#EC MYPSEUDOCOM` is the pseudo-comment that allows to suppress the message. If pseudo-comments are unwanted, insert the constant C_EXCEPTN_IMPOSIBL instead. If, on the other hand, exceptions shall only be possible by applying for and approving to entries in an exception list, insert C_EXCEPTN_BY_TABLE_ENTRY.
While it is still possible to raise messages with different priorities using the same message code, with SAP Web AS 6.40 we strongly recommend to have separate message codes when messages of a single check need to have different priorities. Doing so, check priorities can be customized using transaction SCI with Goto → Maintenance of → Check Priorities.

1.4 Method RUN

During an inspection run, the test driver sequentially loops over all objects of an object set (or, when the inspection is run in parallel, over all objects of one work package). Object type, object name, and program name (if object is program-like) of the single object as well as its prepared data are the basis for the check. These data can be accessed directly as they are attributes of the check class.
In the RUN method you implement the proper check routine. As a first step you should verify whether the check is able to handle the current object type:

```
 check OBJECT_TYPE in TYPELIST.
```

If program-like objects (PROG, CLAS, INTF, FUGR, ...) are to be checked, you should test whether the attribute PROGRAM_NAME is set for the current object:

```
 check PROGRAM_NAME is not initial.
```

In a next step, the check should initiate the needed data preparation, if this not has already been accomplished by another check:

```
 if NEEDED_DATA is initial.
   check GET( ) = 'X'.
 endif.
```

NEEDED_DATA can be any of the attributes your check needs for execution, for example the reference REF_SCAN that points to a class that contains the scanned and tokenised code.
If your check is based on the standard data preparation, calling the GET method will initiate the object dissection by the Code Inspector class hierarchy in a kind of chain reaction. If you are doing data preparation on your own, you should implement it in the GET method of your (data preparation) class.

Following these statements the actual implementation of your check begins. As an example, your check might perform a loop over the ABAP token table REF_SCAN->TOKENS, searching for some special ABAP word. Or your check might examine the technical settings of the objects.

To output the findings of your check, it has to inform the Code Inspector test driver. There are two possibilities to do this:

1. A message can be created by raising the **event ‘MESSAGE’**. If you do not need exception handling, you should use this possibility.

2. If your check extends the check hierarchy below classes CL_CI_TEST_SCAN or CL_CI_TEST_DDIC, **the method ‘INFORM’** can be called. This method call allows for exception handling by the usage of pseudo-comments.

In the following we will describe the interface of the INFORM method; the parameters of the event MESSAGE are nearly identical. That’s how the method INFORM is called:

```
INFORM( P_SUB_OBJ_TYPE = C_TYPE_INCLUDE         "required for program-like
        P_SUB_OBJ_NAME = <include>              "required for program-like
        P_POSITION     = <position>             "required for program-like
        P_LINE         = <line>                 "required for program-like
        P_COLUMN       = <column>               "required for program-like
        P_ERRCNT       = <errnumber>                         "required
        P_KIND         = C_NOTE / C_WARNING / C_ERROR        "required
        P_TEST         = C_MY_NAME                           "required
        P_CODE         = <errcode>                           "required
        P_SUPPRESS     = <'"#EC CI_comment'>                 "optional
        P_PARAM_1      = <parameter 1>                       "optional
        P_PARAM_2      = <parameter 2>                       "optional
        P_PARAM_3      = <parameter 3>                       "optional
        P_PARAM_4      = <parameter 4>                       "optional
                           ).
```

- **P_SUB_OBJ_TYPE --** required for program-like objects
  Type of the checked Include, normally C_TYPE_INCLUDE
- **P_SUB_OBJ_NAME --** required for program-like objects
  Name of the checked Include as provided by method GET_INCLUDE
- **P_POSITION --** required for program-like objects
  The position of the checked statement in the statement table returned by `SCAN ABAP-SOURCE` has to be passed to this parameter to handle pseudo-comment exceptions. This parameter can only be passed to the INFORM method, not to the event MESSAGE.
- **P_LINE --** required for program-like objects
  Line of the source code that caused the message
- **P_COLUMN --** required for program-like objects
  Column of the source code that caused the message
- **P_ERRCNT –** required. Not required with SAP_BASIS 6.40 or higher
  Counter that has to be incremented with every message. Different objects may have identical message numbers, but for one single object identical numbers will lead to an error.
- **P_KIND –** required. Not required with SAP_BASIS 6.40 or higher when check messages are defined in instance constructor (see chapter 1.3)
  Message category. Predefined values are C_ERROR (‘E’: error, serious message), C_WARNING (’W’: warning), C_NOTE (’N’: note, information)
- **P_TEST –** required
  Name of the check class as stored in class attribute C_MY_NAME
- **P_CODE –** required
  Identifier for the sub-check of a check class. This message code is used when displaying message texts and documentation.
- **P_SUPPRESS** --- optional. Can be defined in instance constructor with SAP_BASIS 6.40 or higher (see chapter 1.3)
  In the INFORM method, the pseudo-comment needed to suppress the message is passed to this parameter. In the event MESSAGE, this parameter is a flag. When it is set to TRUE, the message will be suppressed.
- **P_PARAM_1** – optional
  Free parameter of type STRING. Will be included in the message text on position &1
- **P_PARAM_2** – optional
  Free parameter of type CHAR20. Will be included in the message text on position &2
- **P_PARAM_3** – optional
  Free parameter of type CHAR20. Will be included in the message text on position &3
- **P_PARAM_4** – optional
  Free parameter of type CHAR20. Will be included in the message text on position &4

1.5 Method RUN_END

The method RUN_END runs after the RUN method has finished the last object of an object list or a work package. For some types of checks it can be reasonable to use the method RUN only as an object collector and to perform the real check in method RUN_END instead.

Example: Class CL_CI_TEST_DDIC_TABLES checks the technical settings of ABAP data dictionary tables. The RUN method just reads the names of the tables to be checked from the object list into an internal table. The true check method is RUN_END, where the needed information is provided for many tables at the same time and the checks are executed.

1.6 Method GET_MESSAGE_TEXT

With SAP_BASIS 6.40 or higher, the message text can be defined in the instance constructor (see chapter 1.3), so the usage of this method is not longer recommended for new checks.

This method is called from the display of results to provide a message text for every sub-check depending on its message code (cf. parameter P_CODE in chapter 1.4). As a check author you should create a message text for every message code:

```
case P_CODE.
     when '<error_code_1>'.
       P_TEXT = '<Error message with parameter &1>'(101).
     ...
endcase.
```

The parameters P_PARAM_1, ..., P_PARAM_4 that have been passed to the test driver when calling the method INFORM or raising the event MESSAGE can be inserted into the message text by using the replacement characters &1, ..., &4.

1.7 How to insert a new check into the check variant tree

To make your new check visible (and thereby usable) in the check variant tree of the Code Inspector, in transaction SCI choose from the menu Goto → Test Maintenance (SAP_BASIS 6.40: Goto → Management of → Tests). You can then activate your check by setting the ok-flag in front of the check. You will be asked for a transport request since the entry for your check in the list should be transported together with your check class.

2 How to add user parameters to a check

Code Inspector checks with parameters are preceded with an arrow icon when displayed in the check variant tree. When pressing the arrow icon a Pop-Up is generated where the user can select the check parameters. These parameters can be flags, single fields, tables, and select-options.

If you want to offer check parameters to the user of your check, you have to create additional attributes and implement the methods PUT_ATTRIBUTES, GET_ATTRIBUTES, and IF_CI_TEST~QUERY_ATTRIBUTES.

2.1 Additional instance attributes

Create the needed user parameters in the class interface as private instance attributes.

2.2 Instance constructor

In the instance constructor you have to set the flag

```
HAS_ATTRIBUTES = C_TRUE.
```

The attributes can already be initialised in the instance constructor of the check class. Set the attribute ATTRIBUTES_OK = ‘X’ if the check can be executed with this initialisation. Example:

```
P_FLAG = C_TRUE.
ATTRIBUTES_OK = C_TRUE.
```

If ATTRIBUTES_OK is not set to ‘X’ in the constructor method, a user always has to customise the check before it can be included into a check variant. This is reasonable for checks where no parameter setting can be anticipated.

2.3 Methods PUT_ATTRIBUTES and GET_ATTRIBUTES

To be able to read and write the user parameters of a check, you have to implement the methods PUT_ATTRIBUTES and GET_ATTRIBUTES. In these methods the user parameters are exported to or imported from an ABAP data buffer. Example:

```
method GET_ATTRIBUTES.
  export  FLAG  = P_FLAG
          ITAB  = P_ITAB
          FIELD = P_FIELD
          SO    = P_SO
          to data buffer P_ATTRIBUTES.
endmethod.
```

In this example P_FLAG, P_ITAB, P_FIELD, and P_SO are the user parameters (instance attributes) as introduced in chapter 2.1. In method PUT_ATTRIBUTES the statement **`import ... from data buffer`** has to be used.

2.4 Method QUERY_ATTRIBUTES

To create the Pop-Up for selecting the user parameters you have to implement method QUERY_ATTRIBUTES of interface IF_CI_TEST. Example:

```
method IF_CI_TEST~QUERY_ATTRIBUTES.
*-- the grey coding is always needed
data:
  L_ATTRIBUTES   type SCI_ATTTAB,
  L_ATTRIBUTE    like line of L_ATTRIBUTES,
  L_MESSAGE(72)  type C.
define FILL_ATT.
  get reference of &1 into L_ATTRIBUTE-REF.
  L_ATTRIBUTE-TEXT = &2.
  L_ATTRIBUTE-KIND = &3.
  append L_ATTRIBUTE to L_ATTRIBUTES.
end-of-definition.
data: L_OK type flag.
class CL_CI_QUERY_ATTRIBUTES definition load.

*-- fill attribute table – depends on the attributes of your check
FILL_ATT P_FLAG  '<User parameter #1>'(201)    'C'.   "Boolean Flag
FILL_ATT P_SO    '<User parameter #2>'(202)    'S'.   "Select-Options
FILL_ATT P_FIELD '<User parameter #3>'(203)     ''.   "Field
FILL_ATT P_ITAB  '<User parameter #4>'(204)     ''.   "Table

L_OK = C_FALSE.
while L_OK ne C_TRUE.
      if CL_CI_QUERY_ATTRIBUTES=>GENERIC(
                              P_NAME       = C_MY_NAME
                              P_TITLE      = '<my parameter pop-up>'(005)
                              P_ATTRIBUTES = L_ATTRIBUTES
                              P_MESSAGE    = L_MESSAGE
                              P_DISPLAY    = P_DISPLAY ) = 'X'.
*-- = 'X' --> 'Exit' Button pressed on PopUp
         return.
      endif.

*-- Are entries on Pop-Up correct ? (e.g. minimal length of an entry)
      if EVERYTHING_IS_OK.
        L_OK = C_TRUE.
      else.
        L_MESSAGE = '<error_when_selecting_attributes>' (900).
      endif.
endwhile.
*-- Can check be executed with the actual setting of attributes ?
if not ( P_FLAG  is initial and P_SO   is initial and
         P_FIELD is initial and P_ITAB is initial ).
  ATTRIBUTES_OK = C_TRUE.
else.
  ATTRIBUTES_OK = C_FALSE.
endif.
endmethod.                    "if_ci_test~query_attributes
```

The parameter table L_ATTRIBUTES has type SCI_ATTTAB with the following components:

- **REF:** Reference to local parameter
- **TEXT:** Text to be displayed on the Pop-Up for this parameter
- **KIND:** Qualifier `´, `S´, or `C´ that determines the type of the Pop-Up field depending on the parameter’s data type:

Data type of parameter	Qualifier	Field-Type on Pop-Up
Simple Type	`´	**Field**
Simple Type	`C´	**Boolean Flag**
Table Type	`´	**Table**

Table Type	\`S´	**Select-Option**
[SY-INDEX]	[\`G´]	[generates heading for group frame]

The method call **`CL_CI_QUERY_ATTRIBUTES=>GENERIC( ... )`** generates the Pop-Up. If the user presses 'Exit' on the Pop-Up, the return value of the method GENERIC equals \`X', and the Method **`IF_CI_TEST~QUERY_ATTRIUBTES`** should be exited. When the entries on the Pop-Up are not correct, you should pass a message to the user by setting the parameter P_MESSAGE.

3 Documentation of a check in the Code Inspector

For every check that is displayed in the check variant tree, and for every message in the tree of results, an information icon is offered to the user. When pressing this icon, the Code Inspector online documentation concerning the check or message text is provided to the user.

This online documentation is stored in the form of documentation of class attributes. These documents can be created with transaction SE61 (document maintenance). Note that there are no real class attributes associated with these documents. The documentation for attribute '0000' contains general information about the check, other attributes have a name according to the value of P_CODE in the call of method INFORM or the MESSAGE event (cf. chapter 1.4 and 1.6). The documentation of these attributes provides the information for the respective single messages.

General information about the check:
class attribute <my_check_class>
0000

Information referring to the single message of a sub-check:
class attribute <my_check_class>
<code>

4 Existing data preparation classes

Data preparation class	**Additional attribute**	**Information provided by attribute**
CL_CI_TEST_PROGRAM	TRDIR	TRDIR-entry (program repository information)
CL_CI_TEST_INCLUDE	REF_INCLUDE	Reference to read Include (code lines of program)
CL_CI_TEST_SCAN	REF_SCAN	Reference to parsed Include (tokens, statements, levels and structures of program as provided by SCAN ABAP-SOURCE statement)
CL_CI_TEST_SELECT	SELECT_STATEMENTS	Position of SELECT statements in a program
CL_CI_TEST_IMUD	IMUD_STATEMENTS	Position of INSERT, MODIFY, UPDATE and DELETE statements in a program
CL_CI_TEST_SELECT_TAW	SELCT	Parsed SELECT statements (WHERE clauses)
CL_CI_TEST_IMUD_TAW	SELCT	Parsed UPDATE and DELETE statements (WHERE clauses)
CL_CI_TEST_DDIC	TABLIST	List of DB-tables (from object set with DDIC-objects)

5 What is the output of SCAN ABAP-SOURCE?

Please refer to ABAP keyword documentation for statement SCAN ABAP-SOURCE

6 Methods of class CL_CI_TEST_SCAN

These methods help to analyse the (current statement of the) scanned ABAP source.

Method	**What it does**
KEYWORD	determine ABAP keyword of a statement ('COMPUTE' for simple operations)
NEXT1	get token following a single given token
NEXT2	get token following two consecutive tokens
POSITION_1	get position following a single given token
POSITION_2	- not yet implemented
ANALYSE	analyses a token of type A+B(C)
GET_LINE_REL	get line number of given token
GET_COLUMN_REL	get column number of given token
GET_TOKEN_ABS	get token (by absolute index)
GET_TOKEN_REL	get token (by index relative to current statement)
GET_INCLUDE	determine include name
GET_LINE_ABS	get line number (by absolute index)
GET_COLUMN_ABS	get column number (by absolute index)
GET_LINE_COLUMN_ABS	get line and column number (by absolute index)
GET_LINE_COLUMN_REL	get line and column number (by index relative to current statement)

7 Helper classes

To support your checks, in package S_CODE_INSPECTOR exist the following helper classes:

• CL_CI_TEST_DDIC_BUFF	provides buffer settings of DDIC tables
• CL_CI_TEST_DDIC_INDEX	provides primary and secondary indexes of DDIC tables

8 Tips & Tricks for the development of new checks

The following example shows the typical procedure when developing a new check:

Idea for the new check: “Do all developers test whether the internal table ITAB used in the statement SELECT ... FOR ALL ENTRIES IN ITAB has any entry at all?”

Reason for this check: An empty table ITAB will lead to a full table scan on the database table.

Proceeding:

- With help of the Code Inspector search function find a reasonable large amount of statements using SELECT ... FOR ALL ENTRIES IN ITAB
- Examine some concrete implementations: where and how do developers test that the ITAB is not empty when it will be used in a FOR ALL ENTRIES statement?
- Create a test program that contains all relevant types of implementations

- Implement the new check using the data preparation of class CL_CI_TEST_SCAN with the code dissection into levels, structures, statements and tokens
- Verify your check against the created test program
- Improve your check
- If you are confident with the check results, test your check using a very large object set
- Try to evaluate the confidence of your check and how many messages will be created by your check when applied system-wide