Tool to generate a class implementation from its definition

A helper for the implementation of local classes

Link to Content's target Space:
http://wiki.sdn.sap.com/wiki/display/ABAP/Class+Builder

Applies to:
Any ABAP stack on any SAP system with SAP_BASIS >= 702

Summary
This paper exemplifies the use of the ABAP source code scanner. The result is a little program which helps to produce the local class implementation skeleton code from its definition.

Author: Rüdiger Plantiko

Company: Migros-Genossenschafts-Bund

Table of Contents

• A helper for the implementation of local classes
  • Link to Content's target Space:
  • Applies to:
  • Summary
  • Author: Rüdiger Plantiko
  • The ABAP source code scanner
  • Implementing the scanner
  • Note in passing: A string editor popup
  • The tool
  • Building it in as editor pattern
  • Related Content

The ABAP source code scanner

The ABAP source code scanner, accessible directly in the ABAP itself with the statement `scan abap-source;`, decomposes source code given as a table of code lines into a sequence of statements, and each statement into its elementary components (tokens). This is the first stage of the compilation process.

• Comments and whitespace are removed.
• Whitespace and dots separating the tokens and statements, are interpreted and then removed.
• The colon/comma notation is resolved to a sequence of single statements.

Also, the tokens representing symbolic names are converted to upper case.

The result is a simple stream of statements. For example, the code

```plaintext
* Test
data: a type i,
  b type c.
f().
```

results in a table with the following 10 tokens:
As expected, the table with the statements contains three entries. Important are the from and to components, representing the index of the first and last token belonging to the particular statement. In the example, we get:

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

The normalized code representation obtained by the source code scanner may be useful for several purposes:

- **Writing tools which support the code editing.** This wiki actually gives an example for this usage.
- **Mechanical code changes on a larger code base.** Correction reports can be performed on large parts of code. An example for this is the program RS_TOOL_SPLIT. Its purpose is to split a given large module pool along its modularization units - e.g. form routines - into includes, so that each form routine is moved into a new, separate include program. This is useful for support purposes, since it reduces the dependencies between corrections of different units. It was this program, in fact, which has been used for splitting the subroutine pools concomitant to the main module SAPMV45A of the SD order processing into smaller pieces.
- **Checking code of a subset of the ABAP language in an internal DSL.** You may provide parts of your program as a DSL, i.e. allowing only a restricted set of ABAP instructions dedicated for a certain purpose or aspect of the system. To verify that these restrictions are not violated, you need an extended syntax check for this code part.

## Implementing the scanner

It is a good practice to wrap the `scan abap-source` into a scanner class. Instances of this class correspond to scanned source code, which can be traversed with several
convenience methods. Here is an example design of such a scanner class (as local
class, to keep the complete report below self-contained and runnable without
dependencies):

```plaintext
class lcl_scanner definition.
  public section.
  methods:
    constructor importing it_source type stringtab
      raising cx_abap_error_analyze,
    get_first_token importing is_stmnt type sstmnt
      returning value(ev_token) type char30,
    get_second_token importing is_stmnt type sstmnt
      returning value(ev_token) type char30,
    get_nth_token importing iv_n type i
      is_stmnt type sstmnt
      returning value(ev_token) type char30,
    get_first_stmnt importing iv_first_token type char30
      returning value(es_stmnt) type sstmnt,
    get_stmnts importing iv_first_token type char30
      returning value(et_stmnt) type sstmnt_tab.
  data: gt_token type stoken_tab read-only,
      gt_stmnt type sstmnt_tab read-only.
endclass.                    "lcl_scanner DEFINITION

The constructor receives the table it_source containing the source code. It then calls
the scanner on it and populates the tables gt_token and gt_stmnt of this instance,
which are made available from outside as read-only attributes.

```
ev_classname = lo_scanner->get_second_token( ls_stmnt ).

**Note in passing: A string editor popup**

Unfortunately, in a report there is no option for parameters enabling input through a long text window, like this:

*Error rendering macro 'code': Invalid value specified for parameter 'lang'*

```
parameters: p_text type string as longtext. * Would be nice to have!
```

To circumvent this,

- one may design dynpros containing these edit controls (with always the same boilerplate code to instantiate it and provide the values)
- or, a general, reusable popup for text input is sent.

For self-contained reports to be taken over by copy/paste of the source code - like the example program of this Wiki topic - only the second option is feasible. So one has to look for appropriate popup functions.

As far as I see, there is only one popup function displaying a text edit control for entering or changing a table of text, which is available in all SAP applications - since it belongs to the SAP_BASIS layer: The function module **TERM_CONTROL_EDIT**. If you don't care about the length of the text lines, you will use the data type **stringtab** - which is a standard table of elementary line type **string**. The function module works well with this data type. Also, the code handling statements like **read report, insert report**, **scan abap-source, syntax-check** and **generate subroutine pool** work well with **stringtab's**. This makes them the data type of choice for working with source code.

The tool

I now present an example of the scanner's usage. The task of the tool:
Given the definition part of a local class, generate its implementation part, preparing a method sequence for each declared method.

In fact, this is a code transformation program. Following Robert C. Martin that the code should be readable from top to down, like a newspaper, going downwardly into deeper and deeper levels of the code, I start with the traditional input - process - output scheme:

* ---
form start.
  data: lt_def type stringtab,
      lt_impl type stringtab,
      lo_ex type ref to cx_abap_error_analyze.
  try.
    perform : input_def changing lt_def,
             convert using lt_def changing lt_impl,
             output_impl using lt_impl.
    catch cx_abap_error_analyze into lo_ex.
    message lo_ex type 'I'.
  endtry.
endform.  "start

Let's focus on the middle step, the conversion. It should

- find the class name from the class definition,
- determine a list of all declared methods, and
- finally produce the class implementation code skeleton.

The first two tasks belong to the code analysis, the third task is a code building task. Therefore, the next level of specialization looks like this:

form convert using it_def type stringtab changing et_impl type stringtab.
  data: lv_classname type seoclsname,
        lt_methnames type ty_methname_tab.
  perform analyze_def using it_def changing lv_classname
        lt_methnames.
  perform make_impl using lv_classname lt_methnames
             changing et_impl.
endform.

For the analysis of the class definition code, method analyze_def, we need a scanner instance. Then we perform to separate tasks for determining class name and method names:

Error rendering macro 'code': Invalid value specified for parameter 'lang'
form analyze_def
  using
    it_def type stringtab
  changing
    ev_classname type seoclsname
    et_methnames type ty_methname_tab
  raising cx_abap_error_analyze.

data: lo_scanner type ref to lcl_scanner.
clear: ev_classname, et_methnames.
check it_def is not initial.

create object lo_scanner
  exporting
    it_source = it_def.

perform extract_classname using lo_scanner
  changing ev_classname.

perform extract_methnames using lo_scanner
  changing et_methnames.
endform.                    "analyze_def

Let's look, just for example, into the next level - the determination of the method names. I first extract all statements from the source code which start with keyword METHODS. Then, in a loop, I collect for each of these statements its second token which must be the method name:

```
form extract_methnames
  using    io_scanner type ref to lcl_scanner
  changing et_methnames type ty_methname_tab.

data: lt_stmnts type sstmnt_tab,
    lv_methname type seocpdname.
field-symbols: <ls_stmnt> type sstmnt.

lt_stmnts = io_scanner->get_stmnts( 'METHODS' ).
loop at lt_stmnts assigning <ls_stmnt>.
  lv_methname = io_scanner->get_second_token( <ls_stmnt> ).
  translate lv_methname to lower case.
  append lv_methname to et_methnames.
endloop.
endform.                    "extract_methnames
```

As mentioned, the scanner class should be a real workbench class. It is local here only for reasons of self-containment.

Here is the complete code of the tool:

```
*&--------------------------------------------------------------
-------*
*& Report  Z_IMPL_FROM_DEF
*&--------------------------------------------------------------
-------*
*& Makes a class implementation skeleton from its definition
*& Interface methods are not resolved yet
*&--------------------------------------------------------------
-------*

report  z_impl_from_def.

types: ty_methname_tab type standard table of seocpdname.

start-of-selection.
  perform start.
```
* ---
form start.
  data: lt_def type stringtab,
         lt_impl type stringtab,
         lo_ex type ref to cx_abap_error_analyze.
  try.
    perform : 
      input_def changing lt_def,
      convert using lt_def changing lt_impl,
      output_impl using lt_impl.
    catch cx_abap_error_analyze into lo_ex.
      message lo_ex type 'I'.
  endtry.
endform.                    "start

* ---
form input_def changing et_def type stringtab.
  perform edit_text
    using 'Enter section or class def'
    changing et_def.
endform.                    "input_def

* ---
form convert using it_def type stringtab
  changing et_impl type stringtab.
  data: lv_classname type seoclsname,
        lt_methnames type ty_methname_tab.
  perform analyze_def using it_def changing lv_classname
        lt_methnames.
  perform make_impl using lv_classname lt_methnames
        changing et_impl.
endform.                    "convert

* ---
form make_impl
  using 
    lv_classname type seoclsname
    lt_methnames type ty_methname_tab
  changing
    et_impl type stringtab.
  clear et_impl.
  data: lv_line type string,
        lv_methname type string.
  loop at it_methnames into lv_methname.
    concatenate '  method ' lv_methname '.' into lv_line
    respecting blanks.
    append lv_line to et_impl.
    append '  endmethod.' to et_impl.
    append initial line to et_impl.
  endloop.
  if lv_classname is not initial.
    perform wrap_class_impl using lv_classname
      changing et_impl.
  endif.
endform.                    "make_impl

* ---
form wrap_class_impl
  using lv_classname type seoclsname
  changing et_impl type stringtab.
  data: lv_line type string,
        lv_clsnname type string.
  lv_clsnname = lv_classname.
  translate lv_clsnname to lower case.
concatenate 'class ' lv_clsname ' implementation.' into lv_line respecting blanks.
insert lv_line into et_impl index 1.

* Clear last initial line, for esthetical reasons
describe table et_impl lines sy-tfill.
read table et_impl into lv_line index sy-tfill.
if lv_line is initial.
delete et_impl index sy-tfill.
endif.
append 'endclass.' to et_impl.
endform.                    "wrap_class_impl

* ---
form output_impl using it_impl type stringtab.
if it_impl is not initial.
perform edit_text
   using 'Transformation result'
   changing it_impl.
else.
message 'Nothing to convert' type 'S'.
endif.
endform.                    "output_impl

* ---
class lcl_scanner definition.
public section.
methods:
  constructor importing it_source type stringtab
    raising cx_abap_error_analyze,
  get_first_token importing is_stmt type sstmt
    returning value(ev_token) type char30,
  get_second_token importing is_stmt type sstmt
    returning value(ev_token) type char30,
  get_nth_token importing iv_n type i
    is_stmt type sstmt
    returning value(ev_token) type char30,
  get_first_stmt importing iv_first_token type char30
    returning value(es_stmt) type sstmt,
  get_stmts importing iv_first_token type char30
    returning value(et_stmt) type sstmt_tab.
data: gt_token type stoken_tab read-only,
gt_stmnt type sstmnt_tab read-only.
endclass.                    "lcl_scanner DEFINITION

* ---
form analyze_def
   using
     it_def type stringtab
     changing
       ev_classname type seoclsname
       et_methnames type ty_methname_tab
     raising cx_abap_error_analyze.

data: lo_scanner type ref to lcl_scanner.
clear: ev_classname, et_methnames.
check it_def is not initial.
create object lo_scanner
  exporting
    it_source = it_def.
perform extract_classname using lo_scanner
  changing ev_classname.
perform extract_methnames using lo_scanner
  changing et_methnames.
endform.                    "analyze_def

* ---
form extract_classname
  using io_scanner type ref to lcl_scanner
  changing ev_classname type seoclsname.

data: ls_stmnt type sstmnt.
  ls_stmnt = io_scanner->get_first_stmnt( 'CLASS' ).
  if ls_stmnt is not initial.
    ev_classname = io_scanner->get_second_token( ls_stmnt ).
  endif.
endform.                    "extract_classname

* ---
form extract_methnames
  using io_scanner type ref to lcl_scanner
  changing et_methnames type ty_methname_tab.

data: lt_stmnts type sstmnt_tab,
  lv_methname type seocpdname.

  lt_stmnts = io_scanner->get_stmnts( 'METHODS' ).
  loop at lt_stmnts assigning <ls_stmnt>.
    lv_methname = io_scanner->get_second_token( <ls_stmnt> ).
    translate lv_methname to lower case.
    append lv_methname to et_methnames.
  endloop.

endform.                    "extract_methnames

* ---
form edit_text using iv_title
  changing ct_text type stringtab.
  call function 'TERM_CONTROL_EDIT'
    exporting
      titel = iv_title
      tables textlines = ct_text
      exceptions
        user_cancelled = 1
        others = 2.
  if sy-subrc eq 1.
    leave to screen 0.
  elseif sy-subrc ne 0.
    message id sy-msgid type 'A' number sy-msgno
      with sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
  endif.
endform.                    "edit_text

* ---
class lcl_scanner implementation.
method get_nth_token.
  field-symbols: <ls_token> type stoken.
  data: lv_index type i.
  lv_index = iv_n - 1 + is_stmnt-from.
  read table gt_token assigning <ls_token> index lv_index.
  if sy-subrc eq 0.
    ev_token = <ls_token>-str.
  endif.
endmethod.                    "get_nth_token

method get_first_token.
  ev_token = get_nth_token( iv_n = 1 is_stmnt = is_stmnt ).
endmethod.                    "get_first_token

method get_second_token.
  ev_token = get_nth_token( iv_n = 2 is_stmnt = is_stmnt ).
endmethod.                    "get_second_token

method constructor.
  data: lv_msg type string.
  scan abap-source it_source tokens into gt_token
    statements into gt_stmnt
    message into lv_msg.
  if sy-subrc ne 0.
    if lv_msg is initial.

lv_msg = 'Scanner error'.
endif.
raise exception type cx_abap_error_analyze exporting
  message = lv_msg.
endif.
endmethod.                    "constructor
method get_first_stmnt.
data: lt_stmnts type sstmnt_tab.
lt_stmnts = get_stmnts( iv_first_token ).
read table lt_stmnts into es_stmnt index 1.
endmethod.                    "get_first_stmnt
method get_stmnts.
field-symbols: <ls_stmnt> type sstmnt.
loop at gt_stmnt assigning <ls_stmnt>.
  if get_first_token( <ls_stmnt> ) cp iv_first_token. " cp,
  not eq, for being case-insensitive
  insert <ls_stmnt> into table et_stmnt.
endif.
endloop.
endmethod.                    "get_stmnts
endclass.                    "lcl_scanner IMPLEMENTATION

Building it in as editor pattern

In the editor, there is a Pattern button which helps to generate code fragments while programming. The SAP standard provides a couple of patterns. Developers may add their own patterns using the menu path Utilities(M) -> More Utilities -> Edit Pattern -> Create Pattern

It is possible that a pattern generates its code proposal dynamically, depending on some user input or - as in our case - on the actual source code being edited.

For this, I create a pattern of the name Z_IMPL_FROM_DEF, containing only this line:

*$_&$MUSTER

If this pattern is selected by the developer, the workbench tries to call a function module of the name Z_IMPL_FROM_DEF_EDITOR_EXIT (i.e. the concatenation of the pattern name with the fixed string _EDITOR_EXIT). The function module needs a table parameter BUFFER of type RSWSOURCET and may raise the exception CANCELLED if it had sent a dialogue popup that has been quit by the user. The table BUFFER is the code fragment that will be inserted into the current source code.

In our case, an additional difficulty arises: The exit mechanism doesn't provide the current actual source code table. If we had it, we could implement our function as a pattern as follows:

Error rendering macro 'code': Invalid value specified for parameter 'lang'
function z_impl_from_def_editor_exit.
  **--------------------------------------------------------------
  **   Lokale Schnittstelle:
  **   TABLES
  **      BUFFER TYPE  RWSOURCET
  **   EXCEPTIONS
  **      CANCELLED
  **--------------------------------------------------------------

  data: lt_source type rswsourcet.

  clear buffer[].

  try.
    perform get_current_source changing lt_source.

    * CLASS ... IMPLEMENTATION aus CLASS ... DEFINITION
      perform convert in program z_impl_from_def
      using lt_source
      changing buffer[].

    catch lcx_source_not_found.
      endtry.

  endfunction.

How to get the current source? Since the source is not passed to the exit when the
workbench calls it, I use the following dynamical assign to obtain the data:

Error rendering macro 'code': Invalid value specified for parameter 'lang'

form get_current_source  changing et_source type rswsourcet
  raising lcx_source_not_found.

  field-symbols: <it_content> type rswsourcet.

  assign ('(SAPLLOCAL_EDT1)CONTENT[]') to <it_content>.
  if sy-subrc eq 0.
    et_source = <it_content>.
  else.
    raise exception type lcx_source_not_found.
  endif.

  endif.                    " GET_CURRENT_SOURCE

I am not sure whether this method is always reliable, but I didn't find another way to get
the source code.

All future pattern exits will be implemented in this function group, and every request of
the current source code will factor over this subroutine. Thus there is precisely one
place to change if a fix or extension might become necessary.

Related Content

Please include at least three references to SDN documents or web pages.
Reference 1
Reference 2
Reference 3

Useful Information

I discuss the usages of the ABAP source code scanner and give a
concrete example: A tool for generating the code skeleton for
the implementation part of a local class.