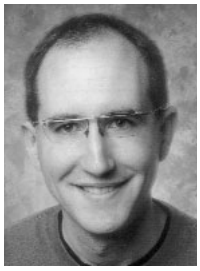
Boost the information processing capabilities of your ABAP programs using regular expressions with SAP NetWeaver

by Ralph Benzinger and Björn Mielenhausen



Ralph Benzinger
SAP NetWeaver
Application Server ABAP,
SAP AG

Björn Mielenhausen
SAP NetWeaver
Application Server ABAP,
SAP AG

(Full bios appear on page 20.)

A large part of information processing — especially in the business settings usually found in SAP environments — consists of *text processing*. Here, text processing applies to working not only with actual natural language text, but also with other data types stored in some textual representation, such as dates, currencies, or any XML data. Accessing, analyzing, and modifying such information is a common exercise encountered during some or most parts of any ABAP development process.

In general, we group text processing tasks into one of three categories:

- **Information validation:** Validate that information conforms to some specification based on syntactic properties
- **Information extraction:** Locate information in a larger body of data based on context rather than position
- **Information transformation:** Convert information between different structural representations

Regular expressions provide a standardized and widely used toolset for processing text-based information effectively and efficiently. While regular expressions have been available for limited use in ABAP programs via workarounds, SAP has not included native support for this functionality — until now. With SAP NetWeaver 2004s, regular expressions have been integrated directly into ABAP Objects to allow programmers to write faster and more robust code.

This article introduces regular expressions and details their integration into the ABAP world. A second companion article (which appears on page 21 in this issue of *SAP Professional Journal*) takes on more advanced topics, such as submatching and performance, and provides additional practical insights for using regular expressions effectively.

Before we delve into the details of how to use regular expressions, let's take a closer look at the previously mentioned categories of text processing tasks that regular expressions are ideally suited to address.

Three common types of text processing tasks

We'll first look at some small examples of the three types of text processing tasks — information validation, information extraction, and information transformation — and then examine how much effort is required to complete one of these using conventional ABAP Objects without the aid of regular expressions.

Information validation

Suppose we are writing an ABAP-based Web service to process some credit card information as part of an e-commerce system. The design of our method specifies that the credit card number and the expiration date be passed as text strings. In keeping with a secure design, it is probably a good idea to check the received data for validity, or at least for plausibility. In our case, we want to verify that the credit card number is a 15- or 16-digit number, and that the expiration date is a 2-digit number followed by a slash, followed by another 2-digit number:

```
1234567812345678  02/06  ->  accept
123456781234XXXX  03/07  ->  reject
1234567812345678  04.08  ->  reject
```

While such an input validity check for the credit card number is easily implemented in ABAP using the `CO` ("contains only") operator and the `strlen( )` ("string length") function, the check for the expiration date is slightly more involved.

In general, the more flexible the specification, the more difficult it becomes to craft a concise ABAP implementation. For instance, in the case of the date, if the specification were relaxed to allow single-digit months, and either a four-digit or two-digit year, the ABAP coding requirements would dramatically increase.

Information extraction

Another common information processing task is to extract some information from a larger body of data. Suppose we are writing a Web service that provides documents to users. The back end receives a URL from the front end that contains, among other things, the ID of the document requested:

```
http://docserve.sap.com/serve?user=ralph
&docid=NW2004&lang=EN
```

In this example, the code is required to extract the document name `NW2004` from the text string.

Other typical examples include the extraction of paths and file extensions from file names, or email addresses from the headers of email messages.

Information transformation

A final group of tasks involves the conversion of information from one structural representation to another. Suppose that our application stores phone numbers in some pretty-printed format that includes spaces, parentheses, and hyphens:

```
(800) 123-4567
+49 6227 747474
```

If we want to interface to an external application that requires phone numbers as a sequence of digits, we will have to write code that strips any non-digits from the information. If the number of non-digits is known and small, this is easily achieved with a number of ABAP `REPLACE` statements. This approach, however, quickly becomes cumbersome if the number of non-digit characters that must be tested becomes large.

Other examples of transforming information include replacing multiple identical characters by single characters or stripping HTML tags (`<...>`) from HTML-formatted documents.

Regular expressions to the rescue!

What all of the previous examples have in common is that while they are a very small part of a larger development task, their solutions require a disproportionately large amount of ABAP code. Fortunately, a remedy is at hand, since it is exactly these kinds of tasks where regular expressions really shine!

Unconvinced? Let's take a first glimpse at the power of regular expressions by writing some code to solve one of the previously outlined simple tasks. Here is some traditional ABAP code that removes all tags from an HTML document stored in variable `mytext`:

```
WHILE mytext CS '<'.
  begin = sy-fdpos.
  FIND '>' IN SECTION OFFSET begin
           OF mytext
           MATCH OFFSET end.
  len = end – begin + 1.
  REPLACE SECTION OFFSET begin
                  LENGTH len
          OF mytext WITH ''.
ENDWHILE.
```

As you can see, this code involves a loop with two searches and one replacement statement in order to solve the simple task of replacing "everything between and including angle brackets." How unsatisfying that this simple concept should demand so much code!

Suffice it to say for now that by using a regular expression, this ABAP code fragment turns into a simple one-liner that, depending on the amount of text to be processed, is also one or more orders of magnitude faster:

```
REPLACE ALL OCCURRENCES OF
        REGEX '<[^>]*>'
        IN mytext WITH ''.
```

After a brief introduction to regular expressions syntax and their integration into the ABAP Objects language in the next section, we show you how to use regular expressions in your own ABAP programs to complete the three main kinds of information processing tasks effectively and efficiently.

Regular expressions — some basics

Regular expressions, or "regexes" for short, are a by-product of theoretical computer science introduced in the 1950s by Canadian mathematician Stephen Kleene (of course, computer science was still called mathematics back then). In addition to their usefulness in mathematics, regexes gained wide popularity in information technology as part of the Unix operating system and its text-oriented tools, such as *grep*, *awk*, and *sed*.[1]

Today, virtually all modern programming languages, including ABAP Objects, support regular expressions, either natively or through some add-on libraries. Although previous releases of ABAP lacked built-in regular expression capability and required various workarounds to make use of external regex resources, native regex support is finally available as of SAP NetWeaver 2004s.

Basic expressions

A regular expression is a *textual pattern* that represents a set of one or more *text strings*. Regular expression patterns are built using ordinary characters as well as some special characters called *operators*, or *metacharacters*:

```
. * + ? ^ $ ( ) { } [ ] \ |
```

The *dot operator*, for example, is a wildcard symbol that represents a single arbitrary character. Thus, the pattern `cat` represents the string `cat`, whereas the pattern `c.t` represents the set of strings that consist of three characters and begin with `c` and end in `t`. Note that without any operators, a regex will represent only one text string, namely itself.

We say that a regular expression *matches* a given

[1] For readers unfamiliar with regular expressions, we recommend David Jenkins' excellent introductory article "Get Ready to Exploit Regular Expression Pattern Matching to Examine and Manipulate Complex Data in Your ABAP Programs" (*SAP Professional Journal*, January/February 2005). We repeat small portions of that information here in order to expand upon implementation details to which David did not have access at the time of his writing.

text string if that string is represented by the regular expression. Matching strings is one of the most frequent uses for regexes: To check if some string belongs to a set of strings with a certain property, we determine if the string is matched by some regex that encodes that property. In our example, regex `c.t` encodes the property "a three-character string that begins with `c` and ends in `t`," so the regex `c.t` matches the string `cut`, `cat`, and `cot`, for example (but does *not* match `car`).

Another way to represent more than one string with a regular expression is to list each *alternative* using the *bar operator* (`|`). The regex:

```
ten|(twen|thir)ty
```

matches the strings `ten`, `twenty`, and `thirty`, and only those. The above example also shows how we use *parentheses* to structure our expression, pretty much as we would in arithmetic expressions.

If we want to match a character that happens to be an operator, we can prepend that character with the *backslash operator* to turn it into a literal — an operation known as *escaping*. For example, `10\.0` matches only `10.0`. On the other hand, `10.0` also matches `10.0`, but it additionally matches `1000`, `10A0`, and so on. Omitting the backslash operator in front of decimal points or periods is a common beginner's mistake that often goes unnoticed, since the flawed regex will still match, albeit too much!

Sets and classes

Listing alternatives explicitly can get cumbersome if there are many of them. Fortunately, there are regular expressions called *sets* that define single character alternatives more concisely.

The set expression:

```
[abc123.!?]
```

matches any single character listed within the square brackets, and only those. Note that all metacharacters (except for the backslash) lose their special meaning inside sets, so the dot in the set shown above will match a literal dot, not an arbitrary character. Sets do, however, sport their very own operators used for defining *ranges* and *negation*.

A *range* provides an even more concise way to define sets. Instead of listing each character individually, we use the range operator (`-`) to specify the start point and the end point of a character interval. So instead of using `[0123456789abcdef]`, we could write the simpler `[0-9a-f]` to denote a hexadecimal digit.

Any set can be *negated* by including the caret operator (`^`) as the first character inside the set. A negated set will match any character *except* those listed inside the set. So, for instance, the expression `[^0-9a-f]` will match anything *but* a hexadecimal digit. Note that the caret needs to be in the first position, or it will be interpreted as a literal caret character.

To include one of the set operators or the closing bracket itself in the set, we simply escape it with the backslash operator, similar to the manner described in the previous section. For example, `[\^\-\]]` will match a caret, a dash, or a right square bracket character.

While ranges are very convenient, especially in technical domains, one caveat must be kept in mind: Ranges are always locale-dependent! For example, the characters contained in range `[ä-ü]` depend heavily on the current code page, making ABAP code using such ranges hard to maintain. Fortunately, most of the issues surrounding ranges are easily avoided by reverting to predefined character subsets called *classes*.

Characters contained in classes are still locale-dependent, but the intended meaning of each class is kept uniformly across all code pages. To use a class, simply write its class name or its shorthand (see **Figure 1**) inside a set definition. For example:

- The `[[:alpha:][:digit:].,?!\-"']` set definition contains characters typically used in prose.
- The `[[:word:]@\-+.]` set definition contains characters typically used in email addresses.

The shorthand notations can also be used outside

Class name	Shorthand	Characters	Negated shorthand
`[:alpha:]`		all letters (even squiggly ones!)	
`[:digit:]`	`\d`	all digits	`\D`
`[:upper:]`	`\u`	all uppercase letters	`\U`
`[:lower:]`	`\l`	all lowercase letters	`\L`
`[:word:]`	`\w`	all letters, digits, and the underscore character (_)	`\W`
`[:space:]`	`\s`	all whitespace characters (space, tab, line feed, carriage return, vertical tab, and form feed)	`\S`
`[:punct:]`		all punctuation characters	

Figure 1 Some predefined character classes

Quantifier	Meaning
`a+`	one or more repetitions of the character a
`a*`	zero or more repetitions of the character a
`a{n}`	exactly *n* repetitions of the character a
`a{n,m}`	at least *n* and at most *m* repetitions of the character a
`a?`	an optional character a (i.e., zero or one repetitions of the character a)

Figure 2 Regex quantifiers for repetitions

of sets, which helps reduce clutter in your code. Negated shorthands such as `\D` abbreviate negated character sets such as `[^\d]`; because of their implied negation, they cannot be used inside of set definitions.

Repetitions

So far, we have only introduced wildcards that match single characters. To define multi-character wildcards, we can select among a variety of repetition operators, also called *quantifiers* (see **Figure 2**).[2] A quantifier repeats the character immediately preceding it.

As with alternatives, we can use parentheses to extend the scope of quantifiers to larger subexpressions:

- `abc*` repeats `c` and matches `ab`, `abc`, `abcc`, `abccc`, and so on.
- `a(bc)*` repeats `bc` and matches `a`, `abc`, `abcbc`, `abcbcbc`, and so on.
- `[abc]*` repeats `[abc]` and matches `a`, `b`, `c`, `aa`, `ab`, `ac`, `ba`, `bb`, `bc`, `ca`, `cb`, `cc`, `aaa`, `aab`, and so on, plus the empty string.

As you can see in the last example, the quantified subexpression is re-evaluated for each repetition, i.e., the subexpression may match a different string in each iteration. Thus, `[abc]*` matches not only strings comprised of identical characters, but arbitrary combinations of characters `a`, `b`, and `c` as well.

The correct number of repetitions of each subexpression is determined "automatically" so that the regex will match if this is at all possible. In other words, when matching regex `a*aa` against text string

[2] As you can see, our quantifier zoo is much more populated than required in order to supply all of this functionality. For example, we could use `(a+)?` instead of `a*`, or `aa*` instead of `a+`. There is even an additional boundary notation `a{n,}`, which is equivalent to `a{n}a*`.

aaaa, the quantifier will match exactly the first two a's and leave the remaining a's for the literal character a in the regex.

Using regular expressions for validation

Now that we have introduced the most basic regular expression operators, we will turn our attention to completing some programming tasks. You will be surprised how far our previously introduced simple regexes will already take us!

In this section, we are given some input that we want to check for some specific traits. These traits are encoded in a regular expression that is matched against the input. Ideally, the regex matches the input if and only if the input shows the trait we are interested in. The problems in writing such a regular expression are generally *false positives* (i.e., inputs that match even though they do not show the desired trait) and *false negatives* (i.e., inputs that do not match even though they show the desired trait).

In reality, a certain amount of false positives or false negatives (but rarely both) are acceptable for many applications. It then becomes a tradeoff between regex complexity and performance on the one hand, and the amount of false matches that we are willing to accept on the other hand. In order to keep our regular expressions legible in the examples throughout this article, we will accept a certain amount of false positives but no false negatives. Interested readers may want to refer to Jenkins' article[3] to see some elaborate expressions that go to great lengths to minimize the possibility of false positives.

Checking credit card numbers for plausibility

In the introduction, we talked about checking credit card numbers for plausibility. But what exactly *is* a plausible credit card number? Maybe our application requires that no invalid characters be included in the number, but allows for some pretty formatting. In this case we could say that a valid credit card number contains only digits, spaces, and hyphens. A regular expression matching those and only those credit card numbers is easily written:

```
(\d|\s|-)+
```

In other words, a valid credit card number is an arbitrary combination of digits, whitespace, and hyphens. To check the validity of some input value stored in variable cc, we have to test whether the above regular expression matches the contents of variable cc. A simple way of doing this in ABAP is to use class method matches( ) of class cl_abap_matcher:

```
IF cl_abap_matcher=>matches(
      pattern = '(\d|\s|-)+'
      text    = cc       ) = abap_false.
  " invalid credit card number
  RETURN.
ENDIF.
```

We will cover other ways of using regexes in ABAP in the next section.

Depending on your needs, you may want to reduce the number of false positives by checking if exactly 15 or 16 digits are present in the input. Ignoring the possibility of spaces and hyphens for now, we can match the string against the regex:

```
\d{15,16}
```

to check for the correct number of digits.

Combining both regexes to check for the correct number of digits and to allow for whitespace and hyphens is not quite as straightforward as one might think at first. Our initial try:

```
(\d|\s|-){15,16}
```

is incorrect, since spaces and hyphens are counted as digits as well, falsely matching a string of, say, 10 digits and 5 spaces. Clearly, this is not what we have in mind. To fix this, we must somehow attach the quantifier bounds to the \d pattern while still allowing for additional spaces and hyphens:

[3] "Get Ready to Exploit Regular Expression Pattern Matching to Examine and Manipulate Complex Data in Your ABAP Programs" (*SAP Professional Journal*, January/February 2005).


```
(\s|-)*(\d(\s|-)*){15,16}
```

Starting with the original `\d{15,16}` to count the correct number of digits, we inserted the highlighted parts to allow for additional whitespace and hyphens. If we unroll the loop in our minds, we see that each position of the credit card number may now contain an arbitrary number of spaces and hyphens.

Validating decimal numbers

In another example, we want to craft a regex that we can use to match decimal numbers. Informally, a decimal number is a bunch of digits, followed by a decimal point, followed by some more digits, but we want to account for the possibility that some of these parts may be missing (i.e., perhaps the leading digits may be missing). Translating this straight into a regex, we arrive at:

```
\d*\.?\d*
```

Note that the decimal point is escaped using the backslash operator — without the escaping, the dot character simply retains its metacharacter meaning. While the above expression generally works, it will also match a single decimal point (`.`), which is a false positive. If we want to exclude this pathological case, and to prohibit trailing decimal points — as in `10.` — along the way, we can regroup our regex into:

```
\d*(\.\d+)?
```

By changing the second asterisk operator to a plus operator, we make sure that the decimal point is followed by at least one digit. Moving the question mark operator to the end then enables this entire expression to optionally cover integers as well. We have thus bound together these previously disconnected parts so that the presence of the former always necessitates the presence of the latter.

To deal with large numbers of type `p` in ABAP, we might want to add support for thousands separators as well. The poor man's solution would be to relax the character constraint for the integral part of the number by replacing `\d*` with `(\d|,)*`. This, however, will match nonsensical strings such as `1,,,.0` as decimals.

How can we prevent this? To come up with a solution, we must first find a property concerning separators that is shared by all valid integers. By definition, separators are separated by exactly three digits. We can further simplify this characterization and state that each separator is followed by exactly three digits. Encoding this property in a regex, we arrive at:

```
\d{1,3}(,\d{3})*(\.\d+)?
```

The leading part `\d{1,3}` matches up to three digits before the first separator (if it exists), whereas the final subexpression `(\.\d+)?` again covers the fractional part of our number (if it exists).

Searching with regular expressions

In our previous discussion on regular expressions, we matched regexes against text strings — for example, to check whether the string satisfies a given structure. In this section we turn our attention to the next major application of regexes, i.e., locating information by *searching* for matches.

When we use regexes for searching, we are generally interested in the position in some string of text of one or all matches of that particular regex. Nevertheless, we often informally state that we want to "search for a regex" when we actually want to "search for matches of that regex." It is important to keep the distinction between matching and searching in mind when reading the remainder of this article. Also note that some other programming languages, most notably — if not notoriously — Perl, use the term "matching" for search operations as well, fueling the confusion further.

Leftmost longest matches

What happens when we search for a regex (or rather the *matches* of a regex) in a given text? Conceptually, every possible match of the regex contained in the text is located, and then, in each group of overlapping matches, the *leftmost longest match* is "found," i.e., returned as a result. For example, the complete set of

matches for regex `a[ao]*a` in the long string shown on the first line below is computed as follows:[4]

```
a[ao]*a:   ooaooaoaoXoooaaooooooaoo
             aooa                (1)
             aooaoa              (2)
                aoa              (3)
                        aa       (4)
                        aaoooooa (5)
                         aoooooa (6)
```

The first two matches (1 and 2) are the leftmost ones of the first group, and among those the second match is the longest, so this is returned as one result of the search. In the second group, matches 4 and 5 are the leftmost ones, and of those the second match is the longest, so this is returned as another result. While the results are thus uniquely determined, it is up to the caller of the search to select among them. Most search interfaces provide only the first leftmost longest match, or alternatively all leftmost longest matches.

The rule of returning the leftmost longest match (or matches) is arguably the most natural behavior for searches, which is probably why it has been defined as a POSIX standard for regular expressions. There are, however, other approaches to searching, which we briefly discuss in the section on technical considerations in our next article (see page 30 in this issue).

Because searches return the longest possible match, we say that matching is "greedy." This applies not only to the overall regex pattern, but also to each individual subexpression, most importantly to repetitions. For now, this fact is of no particular importance to us, but we will readdress this issue when we talk about submatches in the "Working with regular expressions" section later in this article.

To see simply if a match for some regex is contained in a given text, we can use class method `contains( )` of class `cl_abap_matcher`:

```
IF cl_abap_matcher=>contains(
      pattern = '\d+'
      text    = mytext ) = abap_true.
  " match found ...
ENDIF.
```

In the next section, we show you how to obtain detailed information about the matching text, such as the position or the length of the match found.

Anchors and boundaries

To further control where our regex matches inside our text, we can employ some operators that match *positions* rather than characters.

The *anchors* `^` and `$` match the beginning or the end of a line, respectively. Thus, both method calls in:

```
mytext = 'the cat sat on the mat'.
rc1 = cl_abap_matcher=>contains(
         pattern = '^cat'
         text    = mytext ).
rc2 = cl_abap_matcher=>contains(
         pattern = 'cat$'
         text    = mytext ).
```

return `abap_false` since `cat` is neither at the beginning nor at the end of string `mytext`. Because of their Unix legacy, anchors react to newline characters embedded in the string as well. If this behavior is undesired, we can instead revert to anchor variants `\A` and `\z` that match the beginning or the end of the entire string.[5]

Word *boundaries* `\<` and `\>` match the beginning or the end of a word, respectively:

```
\<.at:    Cathy's cat spat at Matt
.at\>:    Cathy's cat spat  at Matt
\<.at\>:  Cathy's cat spat at Matt
```

Technically, word boundaries match the position between a `\w` and a `\W` character, and vice versa. Thus, using a regex such as `\<\w+\>` may seem perfectly

[4] Note that in actuality, ABAP is searching quite cleverly and does not internally compute *all* matches prior to returning the leftmost longest one — it helps, however, to imagine this principal idea when thinking about the search process.

[5] In case you are wondering about the difference in capitalization, there is yet another operator `\Z` that works almost like `\z`, except that it ignores any newline characters at the end of the buffer. Apparently, when the regex operators were originally codified, letters were still abundant, and having the right operator for any conceivable situation seemed the way to go.

sensible, but adding the boundaries to `\w+` is actually redundant in this case because of the leftmost longest match rule.

There are additional boundary operators: `\b` is equivalent to `\<|\>`, and the more interesting `\B` matches the inner parts of words, i.e., positions between two consecutive `\w`'s.

Locating information based on context

A typical scenario for using regexes in searches is to locate some information based on *context* rather than *position*. Recall our introductory information extraction example, where we wanted to extract the document ID from a given URL.

The position of the information is identified by some *delimiters*, i.e., unique characters or keywords that unambiguously indicate the presence of the information we are interested in. In our example, the value of the document ID is pinpointed by the delimiting keyword `&docid=` on the left and the trailing character `&` on the right:

```
http://docserve.sap.com/serve?user=ralph
&docid=NW2004&lang=EN
```

To find this information quickly, we could search using regex:

```
&docid=\w+&
```

if we assume that the document ID is composed of letters, digits, and underscores. Note that this will not cover cases in which the `docid` is the first or the last argument in the URL, so `(\?|&)docid=\w+(&|$)` is a much better solution.

Regular expressions in ABAP

Now that we are able to write our first regexes for matching and searching, we will shift our focus to ABAP and describe how you can harness the power of regular expressions in your ABAP coding. It is helpful to be familiar with general text processing techniques in ABAP in order to follow this exposition. To brush up your knowledge, we recommend our previous article addressing this subject.[6]

Searching and replacing with FIND and REPLACE

The main powerhouses for text processing in ABAP are undoubtedly the `FIND` and `REPLACE` statements. As of SAP NetWeaver 2004s, both statements natively support regular expressions — regexes can be utilized simply by including the `REGEX` keyword in front of the search pattern.[7]

Match results are returned by the `MATCH LINE`, `MATCH OFFSET`, and `MATCH LENGTH` additions. Therefore, to find a string matching regex `mypattern` in string `mytext`, we write:

```
DATA: length TYPE i, offset TYPE i.
FIND REGEX mypattern IN mytext
     MATCH OFFSET off MATCH LENGTH len.
check sy-subrc = 0.
WRITE: / 'found match of length', len,
         'at position', off.
```

Note that unlike when we are performing a `SUBSTRING` search, the length of the matched text may differ from the length of the search pattern. To get at the actual text string that matched, we use the offset-length access idiom that is typical for ABAP code:

```
DATA text_matched TYPE string.
text_matched = mytext+off(len).
```

To search for *all* matches contained in our text, we can include the `ALL OCCURRENCES OF` addition in our `FIND` statement and use the `MATCH COUNT` addition to obtain the number of matches found. When the

[6] "Processing text-based information in ABAP — best practices for improving performance" (*SAP Professional Journal*, May/June 2006).

[7] The `FIND` and `REPLACE` statements also received a minor overhaul regarding the processing of internal tables and the harmonization of additions for the latest ABAP release: You can now search and replace line-by-line in internal tables of `charlike` line type with the `IN TABLE itab` addition. The `FIND` statement also got a new `ALL OCCURRENCES OF` addition, similar to the one already present with the `REPLACE` statement.

Name	Type	Remark
line	i	line of match found/replacement done (for internal tables, 0 otherwise)
offset	i	position of match found/replacement done
length	i	length of match found/replacement done
submatches	TABLE OF submatch_result	internal table of submatches (see the section on submatching in our next article, on page 22 in this issue)

Figure 3 Data type match_result

previous MATCH … additions are used, however, only information about the *last* match found will be returned. To obtain the complete list of matches, we instead use the RESULTS addition that fills an internal table of type match_result_tab with line type match_result (see **Figure 3**).

For each match found, the FIND statement inserts a new line of type match_result into the table. To access this information, we usually employ a LOOP statement to iterate through the individual matches:

```
DATA res TYPE match_result_tab.
FIND ALL OCCURRENCES OF
     REGEX mypattern IN mytext
     RESULTS res.
FIELD-SYMBOLS <m> TYPE match_result.
LOOP AT res ASSIGNING <m>.
  WRITE / mytext+<m>-offset(<m>-length).
ENDLOOP.
```

The RESULTS addition also accepts flat structures of type match_result, but as with the MATCH … additions, only the information pertaining to the last match will be retained. The intended use of RESULTS with structures then is to access the submatch information in the FIRST OCCURRENCE case (see the section on submatching in our next article, on page 22 in this issue, for details).

The well-known IGNORING CASE addition can also be used with regexes to search text case-insensitively. The IN BYTE MODE addition, however, does not work with regexes, since binary patterns are not supported.

All of our previous discussions hold equally true for the REPLACE statement, but unsurprisingly, the corresponding additions are called REPLACEMENT LINE, REPLACEMENT OFFSET, REPLACEMENT LENGTH, and REPLACEMENT COUNT (phew, what a mouthful!).

For further information about the new FIND and REPLACE statements, and some best-practice techniques concerning strings, see our previous article, published in the May/June 2006 issue of *SAP Professional Journal*.

Object-oriented regular expressions

The preferred programming model for modern ABAP programs is to employ an object-oriented methodology. To facilitate this methodology in regex programming, ABAP provides the two classes cl_abap_regex and cl_abap_matcher to work with regular expressions in object-oriented code.

The regex class cl_abap_regex stores the actual regular expression in an internal, preprocessed form (see the section on regex creation in our next article, on page 33 in this issue, for details). The matcher class cl_abap_matcher associates a cl_abap_regex object with some text and acts as the central interface for further processing. In particular, this class offers methods for searching, querying, and replacing matches on a match-by-match basis.

To process some text, we first create a regex object with the CREATE OBJECT statement:

```
DATA myregex TYPE REF TO cl_abap_regex.
CREATE OBJECT myregex
       EXPORTING
```


Search methods	Query methods	Replace methods
find_next( )	get_match( )	replace_found( )
match( )	get_line( )	replace_next( )
	get_offset( )	
find_all( )	get_length( )	replace_all( )
	get_submatch( )	

Figure 4 Methods of cl_abap_matcher

```
          pattern     = 'a*b'
          ignore_case = abap_true.
```

The optional parameter `ignore_case` controls whether matches against this regex should be performed case-insensitively or not; the default is to match case-sensitively. There are some other optional parameters, a few of which we will address in a moment.

Having created our regex object, we then create a matcher object to associate our new regex with the text to process:

```
DATA: mytext    TYPE string,
      mymatcher TYPE REF TO
                    cl_abap_matcher.
CREATE OBJECT mymatcher
       EXPORTING regex = myregex
                 text  = mytext.
```

A regex object can be reused in any number of matcher objects (and reused it should be, for performance reasons that we will discuss in our next article).

Alternatively, the matcher class also provides a factory method to set up both regex and matcher conveniently in a single step (note, however, that the factory method does not allow for any regex object to be shared):

```
mymatcher = cl_abap_matcher=>create(
              pattern = 'a*b'
              text    = mytext ).
```

The matcher object keeps an internal copy of the text to process, so the contents of local variable `mytext` passed to the constructor will not be affected by subsequent match or replace operations. Remember to look at `mymatcher->text` instead of `mytext`, then, when you are looking for the result of your replacement operations! As we described in our May/June 2006 article, the initial copy of `mytext` is highly efficient if `mytext` is of type `string`, thanks to ABAP's built-in string-sharing mechanism. Be aware, though, that any write access to this string through some replace operation will trigger the potentially costly *un*sharing of the string.

Once we have obtained the matcher, we are ready to go. The matcher class provides a variety of methods for searching, querying, and replacing matches, as shown in **Figure 4**.

Interaction with a matcher object is based around the concept of the *current match*, i.e., the last match found during the search process. The current match is set by method `find_next( )` and destroyed by method `replace_found( )`. The numerous get methods require a current match but do not affect it.

Initially, there is no current match, so `find_next( )` must be called first. Once we have a current match, we can query it with the get methods to get information such as its offset and length. By calling `find_next( )` again, we advance to the next match, if there is one; the success of the `find_next( )` call is returned as a boolean value. Once we have visited every match, the current match is invalidated and further calls to `find_next( )` will fail with return value `abap_false`. **Figure 5** on the next page shows how this pattern can be used to sum up all integers that are contained in some given text.

```
DATA: sum       TYPE i,
      mymatch   TYPE match_result,
      mymatcher TYPE REF TO cl_abap_matcher.
mymatcher = cl_abap_matcher=>create( pattern = '\d+' text = mytext ).
WHILE mymatcher->find_next( ) = abap_true.
  mymatch = mymatcher->get_match( ).
  ADD mymatcher->text+mymatch-offset(mymatch-length) TO sum.
ENDWHILE.
```

Figure 5 Summing up all integers listed in some given text

```
DATA: value      TYPE i,
      value_text TYPE string,
      mymatch    TYPE match_result,
      mymatcher  TYPE REF TO cl_abap_matcher.
mymatcher = cl_abap_matcher=>create( pattern = '\d+' text = mytext ).
WHILE mymatcher->find_next( ) = abap_true.
  mymatch    = mymatcher->get_match( ).
  value      = mymatcher->text+mymatch-offset(mymatch-length) + 1.
  value_text = value.
  mymatcher->replace_found( value_text ).
ENDWHILE.
```

Figure 6 Incrementing all integers in some given text by 1

Every time the matcher has stored a current match, we can call the `replace_found( )` method to replace that current match with some new text. This will destroy the current match so that subsequent calls to any of the query methods will raise an exception. Calling `find_next( )`, however, will locate the next unprocessed match and set the current match again.

By using alternate calls to `find_next( )` and `replace_found( )`, we can conveniently iterate through the matches in our text and perform some action based on each individual match. This enables us, for instance, to increment all integers listed in some text document by 1, since the replacement string can be computed *after* the match has been found, as shown in **Figure 6**.

Figure 7 shows the state changes of a matcher object during a typical usage cycle.

The matcher class provides a few additional methods for searching and replacing. The `replace_next( )` method is simply a shorthand for calling `find_next( )` immediately followed by `replace_found( )`, so the method does not require that a current match be stored. The `find_all( )` and `replace_all( )` methods operate on all remaining matches in the text; both calls also invalidate the current match, since there is nothing left to do upon completion. The `match( )` method matches the regex against the remainder of the text not already processed. In general, `match( )` is called right after the matcher is created to match the entire text, but this is not mandatory.

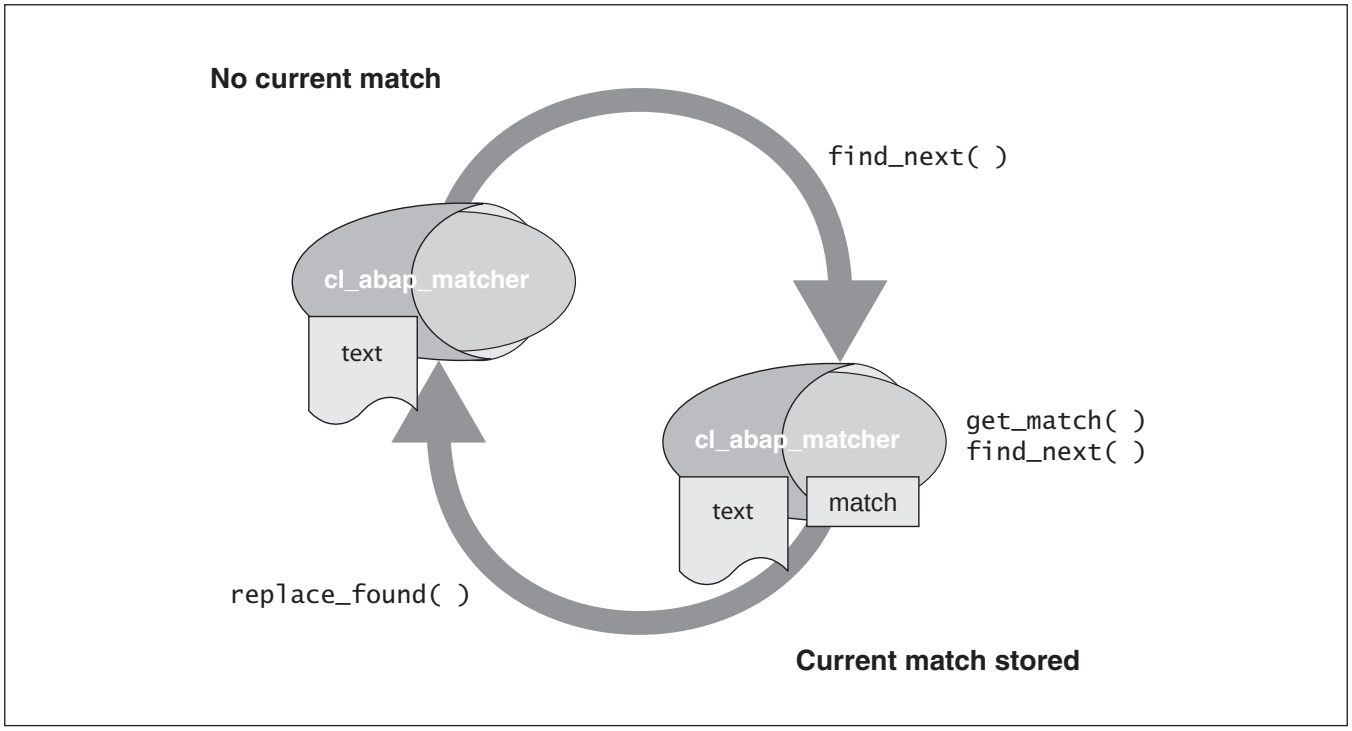


Figure 7 Typical usage of cl_abap_matcher object

Class methods

We have already mentioned class methods `matches( )` and `contains( )` in our earlier introduction to regular expressions. Both class methods are intended to be used in conditional statements as replacements for string operators such as `CS` and `CP`.

Quite frequently we are only interested in the boolean information indicating whether or not our regex has matched the text or whether or not matches are to be found in the text. It is also possible, however, to obtain further information about this match, similar to the old-fashioned (or dare we say, obsolete) way of using `sy-fdpos` with operators such as `CS`. To get at this information, we invoke class method `get_object( )` to obtain a matcher object that represents the state of the previous `matches( )` or `contains( )` operation. We can then use our conventional get methods to query this object further:

```
IF cl_abap_matcher=>contains(
      pattern = mypattern
      text    = mytext    ) = abap_true.
  mymatcher =
      cl_abap_matcher=>get_object( ).
  off = mymatcher->get_offset( ).
  WRITE: / 'found match at offset', off.
ENDIF.
```

Theoretically, it is even possible to call search and replace methods on this object, even though this practice is not recommended.

Working with regular expressions

Regular expressions are not only a part of the ABAP language but have also been integrated into the ABAP development environment. As of SAP NetWeaver 2004s, the new ABAP Debugger supports regexes in many of its search dialogs, with the extension of the remaining screens slated for a later release. Future tool developments, including the new ABAP Editor

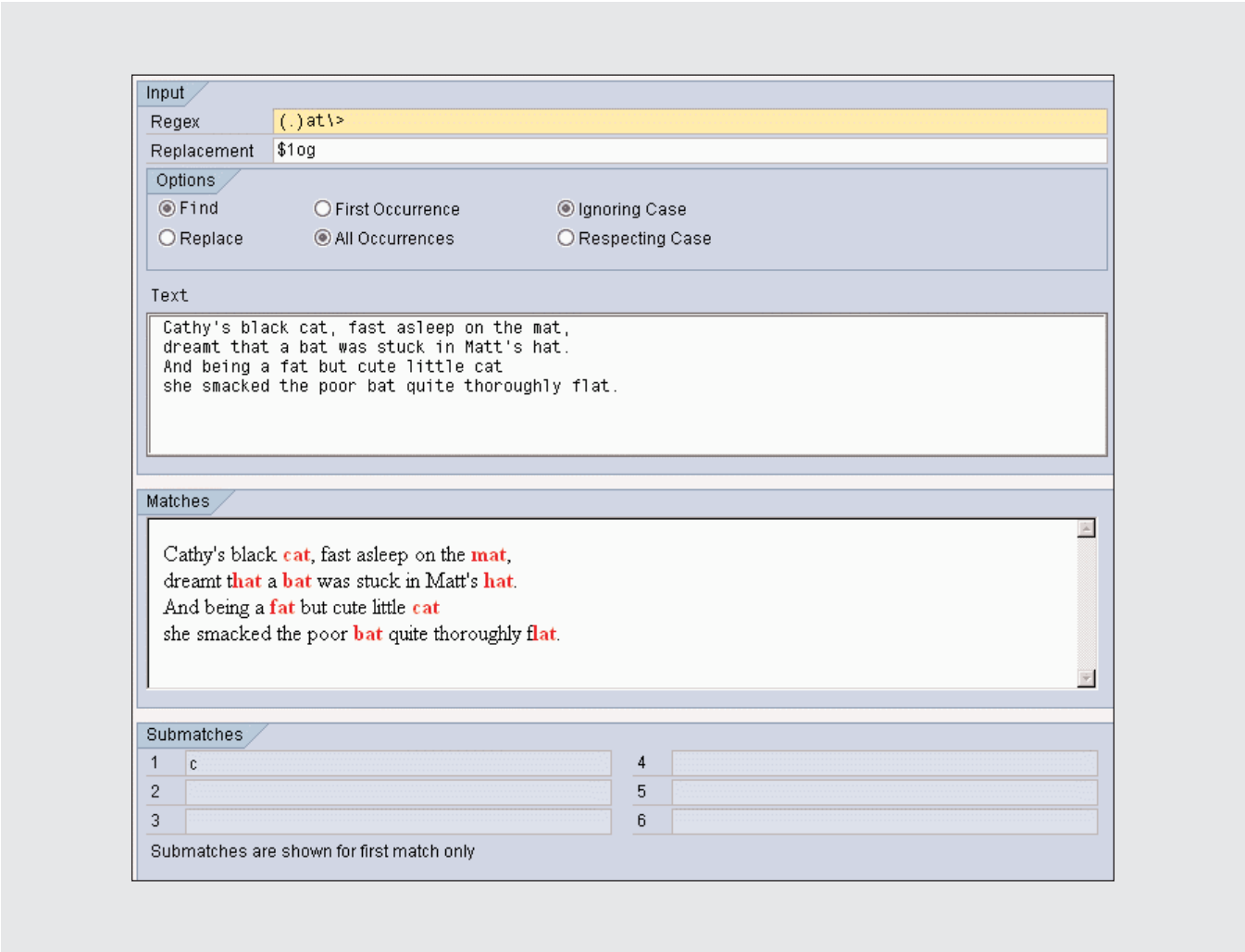


Figure 8 The Regex Toy

and the new refactoring tool, will likewise offer full regular expression support in their user interfaces.

In the remainder of this article, we showcase the Regex Toy and the Code Inspector as two examples of ABAP Workbench tools that offer useful regex functionality.

Getting your feet wet — playing with the Regex Toy

SAP NetWeaver 2004s ships with a handy little program called the Regex Toy that allows developers to test their regular expressions quickly and in a hassle-free way. (If you find that this program is missing from your installation, you will have to upgrade your system to Support Package 7.) The interactive Regex Toy displays when and where matches of regular expressions are found in a user-supplied text. After entering a regex and some text, as shown in **Figure 8**, the program will highlight the first match or all matches of that regex within the text, respectively. At the bottom of the screen, the submatches corresponding to the first match found will be displayed. By selecting the Replace option you can also replace the matches found by some replacement text. Just give it a try — it's really easy!

Note!

If you experience any difficulties in locating the Regex Toy, we suggest that you give the Regex Coach (http://weitz.de/regex-coach/) a try instead. This small standalone application is very similar in functionality to the Regex Toy and is also available free of charge. A quick Web search for "regex" will yield a plethora of additional tools whose number appears to grow almost daily. While some programs may offer some more advanced functionality than the Regex Toy, few, if any, address ABAP-specific issues such as leftmost longest matching or an unlimited number of capture groups.

Analyzing ABAP sources with the Code Inspector

If you're a seasoned ABAP developer you probably have your own story of how at some stage in your career you regularly used some code constructs you now wish you hadn't. Or maybe you just want to improve the quality of some inherited ABAP code by searching for code patterns that potentially deteriorate performance or pose security risks.

The `LOOP AT` statement, for example, has two major variants that use the `INTO wa` and the `ASSIGNING <fs>` additions, respectively. The latter incarnation is generally preferable, as the current line of the table being looped at is accessed by reference rather than by value, thus improving performance. `CALL name` system calls, on the other hand, may inadvertently introduce security hazards if they are not properly guarded by ABAP code that performs appropriate authentication checks.

Suppose now that based on this information, we want to improve our coding and perform a simple code audit — how should we proceed to locate relevant statements? The standard find-and-replace dialog of the classic ABAP Editor is stricken with certain restrictions that severely limit its usefulness for our task at hand:

- Searching is done on a line-by-line basis; consequently, searching for pattern `LOOP*INTO` will not locate `LOOP` statements that span multiple rows.
- Colon-comma expressions obfuscate statements, so `LOOP` statements built out of a `LOOP AT:` prefix will not be located.
- Comments and character literals may contain the search terms, yielding false positives.
- Negative searches — i.e., searches for words that do not match the specified search term — are not supported. This is required by our second example (`CALL name` system calls), however, to exclude perfectly innocuous `CALL SCREEN`, `CALL FUNCTION`, or `CALL METHOD` statements, to name just a few.[8]

The Code Inspector is a comprehensive source code analysis framework that provides just the right kind of tool for this task. The program ships with all SAP NetWeaver releases and supersedes older tools (such as SAMT) that ABAP veterans will most likely be familiar with. The Code Inspector does not suffer from any of the previously listed issues: it works on a statement-by-statement basis on normalized source code — i.e., with colon-comma expressions expanded — and allows for the exclusion of comments. While we cannot provide a detailed introduction to the Code Inspector in this article, we do want to walk through a complete example that shows how regexes can be used to your advantage. Interested readers will find additional information about the tool in a previous *SAP Professional Journal* article.[9]

After starting the Code Inspector with transaction SCI or via the menu entry in the ABAP Editor, we first define a new object set `RALPH_OBJECTS` that contains all programs we want to examine. We then

[8] Note that while searching classically for `CALL '` *will* locate system calls without tripping over `CALL SCREENS` and the like, system calls identified by variables or constants may still be missed that way.

[9] "Evaluating the Quality of Your ABAP Programs and Other Repository Objects with the Code Inspector" (*SAP Professional Journal*, January/February 2003).

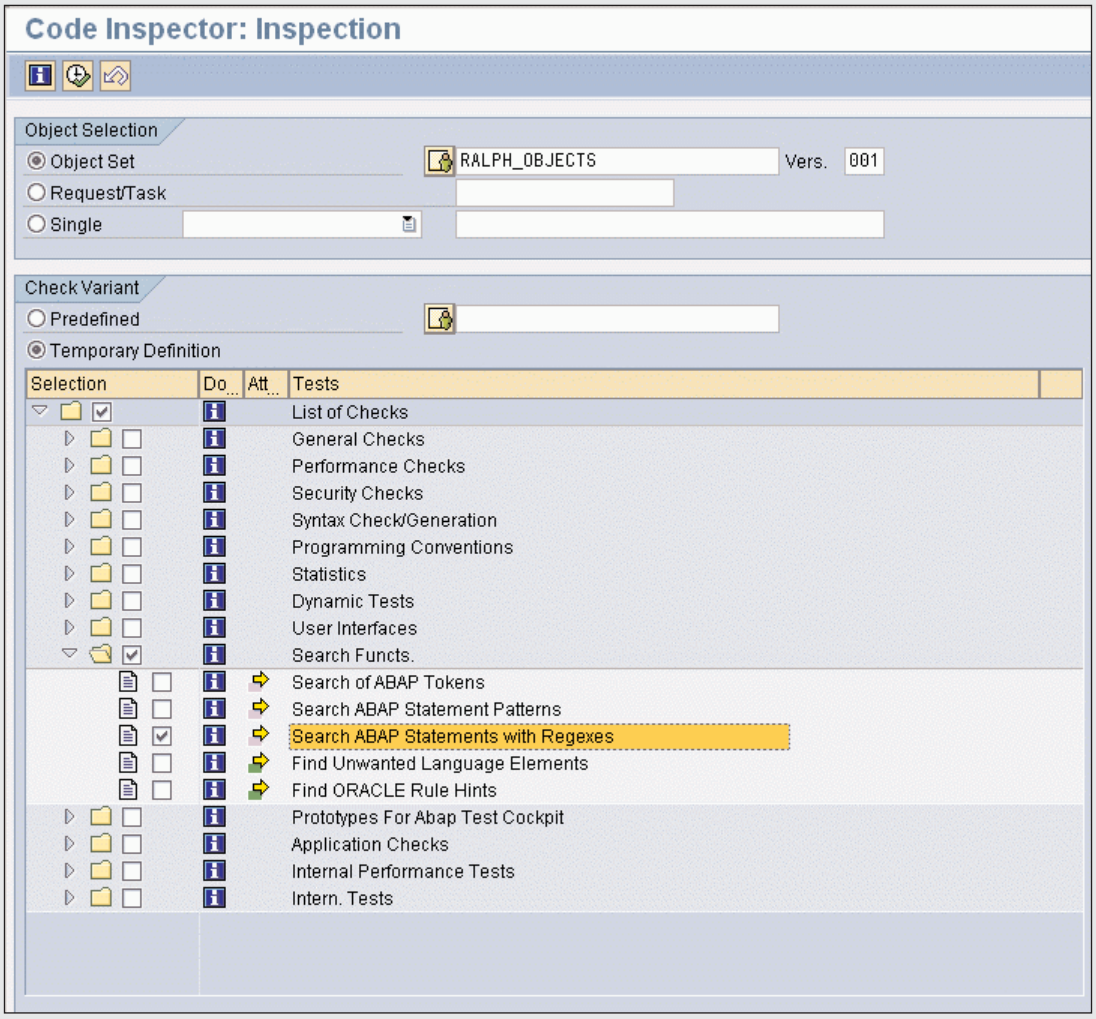


Figure 9 Creating a new inspection in the Code Inspector

create a new inspection to scan all objects that are part of this object set, as shown in **Figure 9**.

We could pick one of the many predefined analyses here, but for our quick one-time run we decide to use a temporary definition instead. After expanding the Search Functs. subtree, we click on the small arrow next to the Search ABAP Statements with Regexes item to bring up the regex search dialog shown in **Figure 10**.

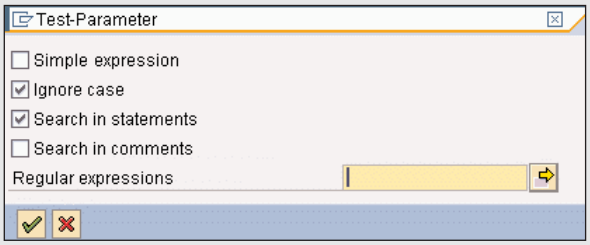


Figure 10 The regex search dialog

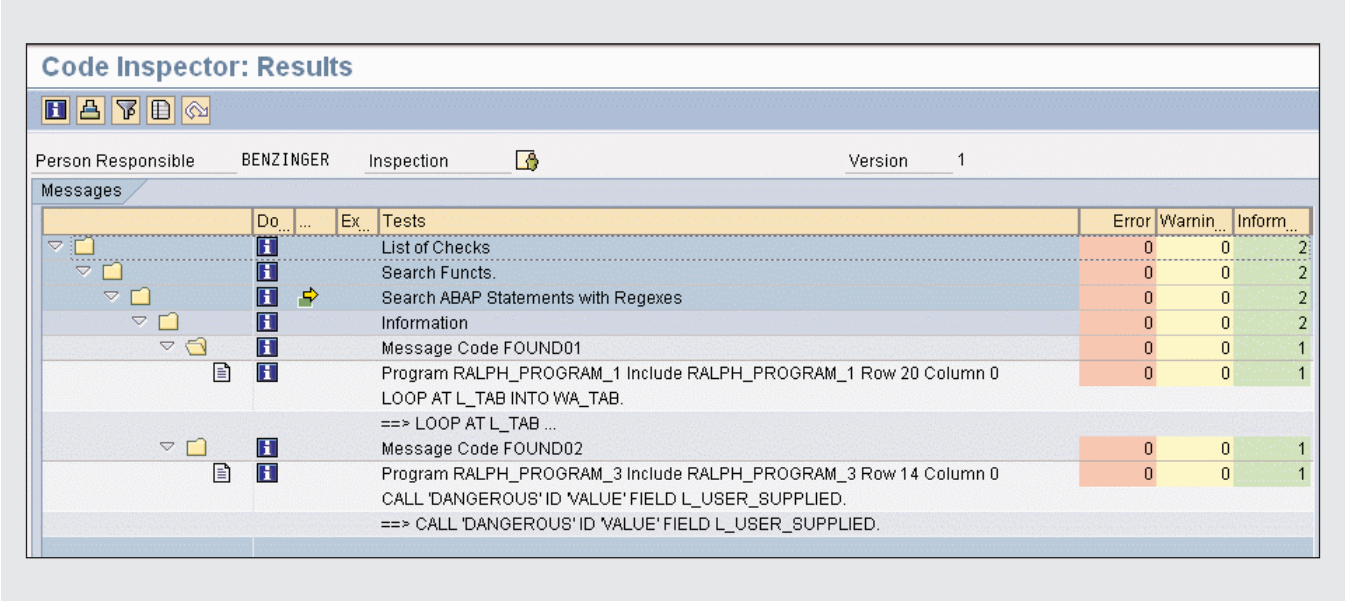


Figure 11 The results of the inspection

Make sure that Search in statements is checked and enter the regex into the text entry box:

```
(\<LOOP\s+AT\>(?!.+\<ASSIGNING\>))|
(\<CALL\s+(?!FUNCTION|METHOD|SCREEN))
```

This pattern will search for `LOOP` and `CALL` candidates that fit our description of questionable programming constructs. To increase the accuracy of our matches, we made ample use of word boundaries and whitespace classes. The new operator `(?!...)` that we snuck in there is called *negative lookahead* and will match if the text enclosed inside the parentheses does *not* match. Let's not get sidetracked for now, though, as this useful operator will be described in detail in our next article. You will notice that the regex entry field can be expanded to specify exclusions and alternatives. Other than increasing readability, however, this does not add new functionality to the search as regexes are already powerful enough to express these conditions by themselves.

After closing the dialog by clicking on the enter button (), we are now ready to start our inspection run. Clicking on the Execute button at the top of the screen or simply pressing the F8 key will send the system spinning, and after a short while we should get a success message and a new button to display the results (**Figure 11**).

In our example run, the Code Inspector found two candidates matching our selection criteria. By double-clicking on the result lines we can quickly navigate to the source code and see if we can enhance the coding by rewriting those sections.

While it is possible to write new Code Inspector modules from scratch, this is rarely required thanks to the large existing modules base. For special requirements, you can find the Code Inspector user manual available for download on SDN (http://sdn.sap.com).

Summary and outlook

Writing effective regular expressions requires some experience, but once you master the basic concepts you will find yourself crafting increasingly sophisticated regexes where you would once have written many lines of ABAP code.

In this introductory article, we explained the rationale behind regular expressions, described their basic use, and presented their integration into the

ABAP language, hopefully whetting your appetite for more. Our next article about regular expressions, which appears on page 21 in this issue of *SAP Professional Journal*, covers advanced matching techniques, some best practices, and technical considerations to boost your ABAP programs even more.

In the meantime, we encourage you to start thinking regexes whenever you write ABAP code: Might this loop around my `REPLACE` statement be eliminated? Do I really need successive `FIND` statements to locate my information? In situations like these, see if a smart little regex might not come to your rescue!

Ralph Benzinger joined SAP in 2003. He is currently a developer in the ABAP language core development group, where he is working on the ABAP compiler and runtime environment. Ralph received a Ph.D. in Computer Science from Cornell University in the US. Prior to joining SAP, he worked as a management consultant in the Business Technology Office of McKinsey & Company. You can reach him at ralph.benzinger@sap.com.

Björn Mielenhausen joined SAP in 1998. He holds a degree in Computer Science from the University of Oldenburg in Germany. He has been working on the ABAP compiler and runtime environment for several years. Since 2003, he has been the Development Manager for the ABAP core team. You can reach him at bjoern.mielenhausen@sap.com.

