# An insider's guide to writing robust, understandable, maintainable, state-of-the-art ABAP programs

## Part 3 — additional best practices and administrative issues

by Andreas Blumenthal and Horst Keller

**Andreas Blumenthal**
Vice President,
SAP NetWeaver
Application Server ABAP,
SAP AG

**Horst Keller**
Knowledge Architect,
SAP NetWeaver
Application Server ABAP,
SAP AG

*(Full bios appear on page 26.)*

***Dedication***

This article is dedicated to the memory of Axel Kurka, who for many years pushed ahead SAP technology, and especially ABAP technology, in his roles as Developer, Development Manager, Product Manager, and last but not least as "Dr. ABAP" in the *SAP Technical Journal*. Axel died much too early in August 2005, but his actions have influenced SAP technology forever.

With the already large, and ever-increasing, amount of options available for ABAP programming, there has long been a need for a comprehensive guide to help both newcomers and veterans navigate through the maze of possibilities and pitfalls. While many guidelines on the use of ABAP already exist both inside and outside of SAP, most of these tend to focus on specific areas of the language.

Based directly upon the daily experiences of the ABAP Language Group at SAP, this article is the third installment of a three-part article series that aims to provide a cohesive set of recommendations on the use of key ABAP programming features that can serve as a foundation for or complement to specific guidelines for your own organization.

The first installment[1] introduced some fundamental rules and formal criteria for writing state-of-the-art ABAP programs. The most important of these were:

- Use ABAP Objects
- Use Unicode-enabling

[1] "An insider's guide to writing robust, understandable, maintainable, state-of-the-art ABAP programs: Part 1 — fundamental rules and formal criteria" (*SAP Professional Journal*, January/February 2006).

*No portion of this publication may be reproduced without written consent.*

In the second installment,[2] we started to drill down into some ABAP programming nitty-gritty and presented suggestions for how best to use the language elements allowed in Unicode-enabled ABAP Objects. In particular, we examined best practices for:

- ABAP Objects, modularization, and program flow
- Declarations
- Processing data

***Note!***

This article, like the previous two installments of this article series, applies to SAP Web Application Server (SAP Web AS) 6.10 and higher. Although some of the concepts mentioned in this article are available only as of SAP NetWeaver Application Server (SAP NW AS) ABAP 7.0 in SAP NetWeaver 2004s (the successor to SAP Web AS ABAP 6.40 in SAP NetWeaver '04), many of the same principles apply and you should be able to follow along easily.

***Note!***

Many of the guidelines presented in this article assume that you are following our first and foremost recommendation, presented in Part 1 of this article series — to use the ABAP Objects programming model, which is easier to use and provides enhanced programming capabilities, wherever possible.[3]

[2] "An insider's guide to writing robust, understandable, maintainable, state-of-the-art ABAP programs: Part 2 — best practices" (*SAP Professional Journal*, March/April 2006).

[3] For more on why ABAP Objects is the recommended approach, see Part 1 of this article series in the January/February 2006 issue of *SAP Professional Journal* and also the article "Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Programmer Should Give It a Second Look" in the September/October 2004 issue of *SAP Professional Journal*.

In this third and final installment, we'll continue looking at best practices for using ABAP programming features, including recommendations for:

- Data storage and retrieval
- Dynamic programming
- Error handling

Finally, we'll conclude by examining some key administrative issues that developers needs to be aware of, in particular:

- Testing programs
- Documenting programs
- Using packages

## Best practices for data storage and retrieval

In most cases, ABAP programs work with data that is retrieved from persistent storage media. For performance reasons, data can be stored in the shared memory of the application server intermediately. Access to data in shared memory is orders of magnitude faster than access to data in persistent media. In the next sections, we'll look at how best to use:

- Persistent data storage and retrieval
- Shared memory

## Persistent data storage and retrieval

You can store persistent data in several formats and in several media. It can be stored as:

- Relational database tables in the database
- Data clusters in the database
- Binary or text files on the application server
- Binary or text files on the presentation server

 ©2006 SAP Professional Journal. All rights reserved.

### Recommendation

You must plan carefully where to store and how to transport your data:

- Storing data in relational database tables is always the first choice. Object Services provides a framework to handle such data in an object-oriented way.[4]
- In the following cases, data clusters in database tables can be appropriate (statements `EXPORT/IMPORT TO/FROM DATABASE`):
  - For reasons of simplicity and performance, you can regard data objects that are derived from relational data (as a result of a sophisticated analysis, for example) as congregations and store them directly in a persistent data cluster to be reused by other programs.
  - In some cases, the data objects you need to work with might not be suitable to be directly stored in databases having the first normal form (1NF) — such as nested internal tables, for example. Instead of mapping the data to the first normal form before storing and reconstructing the data after reading, you can store it directly in persistent data clusters.
  - You cannot treat reference variables with `EXPORT/IMPORT`. Therefore, it is not possible to store anonymous objects that are created from `CREATE DATA` or `CREATE OBJECT`, or relations between objects, directly in data clusters. As a workaround, you can serialize reference variables that point to objects using `CALL TRANSFORMATION`. The result of such serializations can then be stored in data clusters. But note that for storing results of serializations only, you can also use database tables with string type columns directly.
- Files on the application server (and certainly files on the presentation server) are not suited for storing data that must be transported to other ABAP systems. Files can provide an interface to import or export data from or to non-ABAP systems, but they should not be used for data storage within an ABAP system. The reason is that there is no built-in support for handling the different code pages or byte orders that can occur in different ABAP systems, or even might differ between the different application servers of a single ABAP system. When working with files, you must take care of the encoding of the file contents yourself (see the guidelines for internationalization in Part 1 of this article series). For database tables and data clusters, this is done by the ABAP runtime environment. See the ABAP keyword documentation about the ABAP file interface and the possible modes for opening files.

When handling persistent data, follow these guidelines:

- Never reuse system-wide database tables or similar resources (table INDX, for example) to store anything other than temporary data. Use existing database tables only if there is an infrastructure available for their use. Create your application's own specific database table (and its respective infrastructure), even if the structure is identical to some other pre-existing table.
- Check the performance hints for database tables in the ABAP keyword documentation.
- For reading data from database tables, do not use logical databases. (As of SAP NetWeaver 2004s, you also have the option to use the Object Services Query instead.)
- Be aware of the compress options for data clusters (addition `COMPRESSION` for `EXPORT`).

## Shared memory

An application server's shared memory is the most important memory for transient and semi-persistent data (buffers). The shared memory is used implicitly for program data and buffers (e.g., SAP buffering) and can be accessed explicitly from an ABAP program via cross-transaction application buffers using:

```
EXPORT|IMPORT ...
  TO|FROM SHARED BUFFER|MEMORY ...
```

[4] For further information on this subject, see "Write Smarter ABAP Programs with Less Effort: Manage Persistent Objects and Transactions with Object Services" (*SAP Professional Journal*, January/February 2002).

and shared objects[5] (as of SAP NetWeaver ’04):

```
CREATE OBJECT|DATA ...
  AREA HANDLE ...
```

> ***Note!***
>
> Shared memory is frequently used and may therefore become a sparse resource, which can result in bottlenecks when explicitly storing data from ABAP programs. This can especially be a problem on 32-bit platforms, where the shared memory that is available for application programs can be rather limited. Transaction ST02 (SAP Memory Management) shows the current occupancy of the shared memory and the corresponding profile parameters that define the amount of memory that is available for different uses.

## Recommendation

Compared to shared buffers (`EXPORT/IMPORT`), shared objects (`CREATE AREA HANDLE`) have some significant advantages:

- You can store object structures with complex dependencies.
- You can work with shared objects just as you can with objects located in your regular session memory.
- The same memory can be used simultaneously and copy-free by several users.

Therefore, it is recommended that you use shared objects instead of shared buffers. Don’t use the so-called “contexts” (statements `CONTEXTS`, `SUPPLY`, `DEMAND`), which are also stored in the shared memory, any longer, since their use can be replaced by shared objects.

[5] See “Reduce Memory Consumption and Improve Runtime Performance with ABAP Shared Objects: A New API for Shared Memory Programming” (*SAP Professional Journal*, March/April 2005).

There are two scenarios that illustrate the effective use of shared objects:

- A **shared buffer** is a data store that is rarely changed (once a day up to a maximum of once an hour), usually by a single user only. The amount of data can be very large. In general, many users have read access to a shared buffer at the same time. A typical use of a shared buffer is to store some kind of catalog or metadata.
- An **exclusive buffer** is a data store that is read or write accessed by one user only or — less frequently — by one user with write access and another with read access. The data stored in an exclusive buffer should be available longer term; that is, longer than a program’s lifetime. A typical use of an exclusive buffer is to store a shopping basket that is filled initially by the shopper and is then later read by the salesperson.

  Shared objects are ideal for either scenario.

Other uses (of general shared memory programming) are *not* recommended. The current locking logic does *not* enable you to set specific locks for the following requirements:

- Many parallel read and write accesses
- Frequent write accesses
- Division into changeable and non-changeable areas

Although the locking logic makes the first two points technically possible, they are not practical because most accesses would be rejected. Also be aware that for shared objects, memory overflow exceptions can only be handled in the release that follows SAP NetWeaver 2004s.

It is recommended that application programs do not access shared objects memory directly, but that read accesses to shared objects are instead wrapped in a wrapping class, and that only the methods that read the data are accessed by the individual programs. With such a wrapping, you gain:

- Central administration of an area in shared memory
- No need to worry about area handles and locks
- Central administration of locks

 ©2006 SAP Professional Journal. All rights reserved.

- Option to implement central authorization checks

Now that you have a solid understanding of how to use data storage and retrieval to your advantage, let's take a look at one of the more powerful ABAP features at your disposal, and how best to use it — dynamic programming.

## Best practices for dynamic programming

The concepts of dynamic programming are part of the unique features that ABAP offers for business programming. Exploiting the features of dynamic programming allows you to write powerful and flexible applications. On the other hand, programs or procedures exploiting dynamic programming features can become more complex and less maintainable compared to programs that use only static features. Furthermore, with dynamic programming, many checks that are normally done during compilation are postponed until runtime and must be done over and over again for each execution. Therefore, the performance of dynamic programs can be poorer than that of static programs.

Therefore, as a general rule we recommend the prudent use of dynamic programming. Don't use dynamic features where you don't need them. If you need dynamic programming, choose the means that are most appropriate for your task — for example, prefer dynamic token specification vs. code generation if possible.

In the following sections we list some recommendations for dynamic programming and point out some precautions you must take that are not necessary in static programming. (In static programming such errors can be found by the syntax checker.) We also provide some additional hints on performance.[6] Specifically, we will look at the following:

- Dynamic data objects
- Anonymous data objects
- Field symbols and reference variables
- Dynamic token specification
- Runtime Type Services — RTTI and RTTC
- Program generation

[6] For further information on this subject, see "Build Flexibility and Control into Your ABAP Programs with Dynamic Programming Techniques" (*SAP Professional Journal*, November/December 2001).

## Dynamic data objects

A *dynamic data object* is a data object for which the memory consumption is not determined by the data type. Strings and internal tables are examples of dynamic data objects. They are also *deep data objects*, where the actual data content is accessed by internal references (pointers).

The memory consumption of a deep data object consists of a constant eight-byte overhead (for the reference) plus dynamically required memory. The dynamically required memory consists of so-called *headers*, containing administrative information, and the actual working data. Dynamic memory (header and data) is requested when content is inserted into strings or internal tables. When a dynamic data object is initialized with `CLEAR`, `REFRESH`, or `FREE`, only the actual data is deleted, but the header is kept and reused when memory is requested again. As a rule of thumb, every dynamic data object needs about 100 bytes for internal administration.

### Recommendation

Besides the special recommendations for strings and internal tables (see the guidelines for processing data in Part 2 of this article series), we present here some general recommendations for dynamic data objects:

- Keep an eye on the memory consumption of your dynamic data objects and avoid runtime errors due to memory overflow. The maximum size of a dynamic data object is limited by the following factors:
  - The maximum memory size that the current internal session can request for data objects

- An upper limit of $2^{31}$-1 for the number of places in strings and rows in internal tables
- An upper limit set by the profile parameter ztta/max_memreq_MB for strings and hash tables (for strings, this profile parameter sets a direct limit for the memory; for hash tables, you can calculate the maximum number of rows by computing the largest power of two that's less than or equal to the value of the profile parameter, divided by 8)

***Note!***

The actual maximum size available to any given dynamic data object is generally smaller than the size given in these guidelines, since the overall available memory is normally not used by one string or internal table.

- To avoid memory bottlenecks, avoid large dynamic data where the memory consumption of the administrative overhead is in the same order of magnitude or larger than the application data. Especially avoid the nesting of dynamic data objects that contain only a small amount of application data in large internal tables — you could easily waste a lot of memory for nothing. You can check the exact memory requirement of your dynamic data objects in the ABAP Debugger via the function "Memory consumption" and by creating a memory snapshot for the Memory Inspector.[7]
- Avoid runtime errors that occur when you attempt to access non-existent parts of dynamic data objects. Such errors can commonly occur when using offset/length programming on strings. Since the length is not fixed, you can easily create runtime errors. We recommend that you always check the string length first. The same guideline holds true for accessing internal table lines via field symbols or reference variables — you must guarantee that such a field symbol `IS ASSIGNED` or the reference variable `IS BOUND`.
- Generally, you store dynamic data that is suited to be represented in one piece in strings, and data that is divided into sections in the lines of internal tables. There is one exception to that rule. If you have to work with very large dynamic data objects and there is a danger of running out of memory, it might help to split the data in such a way that it can be stored in an internal table instead of in a string. The reason is that an internal table requires its memory in blocks while a string requires its memory in one chunk. The two variants for storing a data cluster in dynamic data objects are an example. Normally, you would use `EXPORT TO DATA BUFFER` to store the cluster in one long byte string. If the data cluster becomes very large, you can try to prevent a memory overflow by using `EXPORT TO INTERNAL TABLE`.

[7] For more information on the Memory Inspector and its use, see "Analyze Memory-Related Problems in Your ABAP Programs in Less Time and with Less Effort Using the ABAP Memory Inspector" (*SAP Professional Journal*, November/December 2004).

## Anonymous data objects

An *anonymous data object* is a data object that cannot be addressed using a name. Anonymous data objects are created by a `CREATE DATA` statement and can be addressed using data reference variables. Anonymous data objects can be created from existing data types or from type objects resulting from RTTS (see the upcoming guidelines for Runtime Type Services).

Anonymous data objects are closely related to objects of classes created by `CREATE OBJECT`. Like an object of a class, an anonymous data object is deleted from the internal session by the *garbage collector* if it is no longer referenced. While a general object is an instance of a class, an anonymous data object is an instance of a data type. Of course, the latter is true for all data objects. The most commonly used named data objects (declared by `DATA`, etc.) are instantiated automatically when their context is loaded into the internal session. While objects of classes can only be instantiated explicitly and at runtime via `CREATE OBJECT`, you have the choice to do so when creating data objects.

 ©2006 SAP Professional Journal. All rights reserved.

### Recommendation

You work with anonymous data objects in the following scenarios:

- If you need data objects of data types that are unknown statically — such as work areas for dynamic Open SQL, for example (see the upcoming guidelines on dynamic token specification)
- If you want to manage complex data structures such as linked lists or trees

Normally you do not need anonymous data objects to circumvent the high costs of instantiating huge objects where you don't need them, because ABAP's dynamic data objects (see the earlier guidelines for dynamic data objects) already fulfill that purpose.

Regarding memory consumption, the same recommendations principally hold for anonymous data objects as for objects of classes (see the guidelines for using ABAP Objects in Part 2 of this article series) — release unneeded objects to the garbage collector. Please note that for anonymous data objects, it is not necessarily only large data objects that might lead to memory bottlenecks; there is also the danger of holding references to large numbers of small or medium-sized data objects. In particular, internal tables containing reference variables might tie up a lot of memory without needing the memory space themselves. As for dynamic data objects, the Memory Inspector is a highly recommended tool for checking where your memory has gone when you are working with anonymous data objects. It offers various possibilities to analyze the memory held by a single data object.

## Field symbols and reference variables

A *field symbol* is a symbolic name for a data object to which you can assign actual memory areas at program runtime. A field symbol can be used as a placeholder for a data object at operand positions. A field symbol can be typed either *generically* or *completely* (see the guidelines for declarations in Part 2 of this article series). Field symbols are declared using the `FIELD-SYMBOLS` statement and memory areas are assigned to them with `ASSIGN` or the `ASSIGNING` additions of internal table statements.

A *reference variable* is a data object that contains a reference. Reference variables are further differentiated into *data reference variables* and *object reference variables*. Reference variables are typed with the addition `REF TO` that defines their static type. The static type is always more general or the same as the dynamic type (this is the "golden rule" for static and dynamic types). The dynamic type is the type of the referenced object.

- For data reference variables, the possible static and dynamic types are data types. The static type of a data reference variable is either the fully generic data type `data` or a data type where all technical properties (see the guidelines for declarations in Part 2 of this article series) are specified.
- For object reference variables, the possible static types are classes and interfaces (object types), and the possible dynamic types are classes. The static type of an object reference variable can be any class or interface, where `object`, the super class of all other classes, is the most general one.

The data or object reference inside a reference variable can be copied to or compared with other data or object references, respectively, but not accessed directly otherwise. Copying is possible, as long as the golden rule is fulfilled and can be divided into up casts (golden rule can be checked statically) and down casts (golden rule must be checked dynamically).

For accessing objects created via the `CREATE` statement — i.e., instances of classes or anonymous data objects (see the earlier guidelines on anonymous data objects) — the use of reference variables is technically mandatory. There are also other operand positions, where the use of either field symbols or data reference variables is prescribed. For dynamic programming with data objects, however, you can generally choose whether to work with field symbols or with data objects.

No portion of this publication may be reproduced without written consent.

### Recommendation

Whether to use field symbols or data reference variables to access data objects dynamically depends on the purpose. For all tasks that can be fulfilled by data reference variables, use them. Do not use field symbols for mere pointer tasks. Examples where field symbols are necessary are:

- De-referencing of data reference variables in order to access anonymous data objects if the static type is the generic type `data`
- Dynamic access to components of structures (`ASSIGN ... COMPONENT`)
- Dynamic access to attributes of classes and objects (`ASSIGN oref->(att_name)...`)
- Casting of data objects (`ASSIGN ... CASTING`)

Avoid the use of field symbols with character or byte type data objects if the only reason is to achieve dynamic offset/length programming. Almost all dynamic offset and length specifications are also possible when working with the data objects directly.

> ***Note!***
>
> Before accessing the data referenced by a field symbol or a reference variable, you must always ensure that the field symbol `IS ASSIGNED` or the reference variable `IS BOUND`. Otherwise, a runtime error can occur.

## Dynamic token specification

ABAP statements consist of *tokens*. Tokens are ABAP words, operands, and operators. Tokens are combined to form language elements or language element additions that express the semantics of a statement. In many statements, the ABAP syntax allows you to specify a single token or a sequence of tokens dynamically. To do so, you replace the token with the name (surrounded by parentheses) of a character type data object or an internal table with a character type line type. The system resolves the contents of the dynamic token specification only at runtime.

### Recommendation

As a general rule, dynamic token specification should be the main means of dynamic programming in ABAP. The language offers you enough possibilities for dynamic token specifications that program generation should be used in exceptional cases only (see the upcoming guidelines on program generation). The advantage of dynamic token specification vs. program generation is that only parts of a statement must be specified dynamically whereas the rest can be checked statically at compile time. Of course you still must take care to avoid possible exceptions coming from incorrect dynamic syntax or incorrectly named resources inside the dynamic token specification. Therefore, we would counsel you to never use dynamic token specification where static specification is possible.

Important statements where dynamic token specification should be used instead of code generation are:

- **Internal table operations:** You can address the components of internal tables dynamically in the statements for accessing internal tables. The dynamic `WHERE` clause in `LOOP AT`, `MODIFY`, and `DELETE` for internal tables will be available in the release following SAP NetWeaver 2004s.
- **Open SQL:** In all Open SQL statements, and especially in `SELECT`, you can specify all clauses via dynamic token specification.[8] (See Figure 3 in the upcoming guidelines for Runtime Type Services.)
- **Dynamic access:** In order to access data objects dynamically, you can specify them via dynamic tokens in an `ASSIGN` statement. There is a large number of syntax variants available for that purpose, providing great variability for the dynamic access of static and instance attributes of classes or of the components of structures.
- **Dynamic invocation:** In order to call procedures or other programs dynamically, you can specify their names dynamically in the respective

[8] For more information on this subject, see "Enhanced ABAP Programming with Dynamic Open SQL" (*SAP Professional Journal*, September/October 2001).

 ©2006 SAP Professional Journal. All rights reserved.

statements. Please note especially that code generation is no longer required for passing parameters to or handling exceptions of dynamically called procedures. When calling methods as well as function modules, you can pass special parameter tables or exception tables that specify the parameter passing and exception handling dynamically (see the sidebar "Passing parameters dynamically" below for an example).

- **Dynamic instantiation:** You can specify the object type (class) or data type in statements `CREATE OBJECT` or `CREATE DATA` as dynamic tokens. When creating objects, you can pass parameters to the instance constructor dynamically as you do during dynamic invocation. When creating data objects, you can refer to dynamic type objects from RTTC (see the upcoming guidelines for Runtime Type Services).

In order to prevent runtime errors resulting from dynamic token specification, don't forget to always check if the used resources are available and/or provide for appropriate exception handling, which can differ from statement to statement. For example, the following code excerpt shows the dynamic

## Passing parameters dynamically

The coding sequence below shows the dynamic call of the static method `gui_download` of global class `cl_gui_frontend_services` for storing the content of an internal table in a file on the current presentation server. The names of the class and method are specified in the strings `class` and `meth`. The interface parameters are passed in the internal table `ptab`, and return values assigned to the exceptions of the method are assigned using table `etab`. Exceptions that occur during the dynamic method call itself are handled in a `TRY` control structure with statement `CATCH`.

```
DATA: meth  TYPE string,
      class TYPE string,
      ptab TYPE abap_parmbind_tab,
      ptab_line TYPE abap_parmbind,
      etab TYPE abap_excpbind_tab,
      etab_line TYPE abap_excpbind.

class    = 'CL_GUI_FRONTEND_SERVICES'.
meth     = 'GUI_DOWNLOAD'.
filename = 'c:\temp\text.txt'.
filetype = 'ASC'.

ptab_line-name = 'FILENAME'.
ptab_line-kind = cl_abap_objectdescr=>exporting.
GET REFERENCE OF filename INTO ptab_line-value.
INSERT ptab_line INTO TABLE ptab.

ptab_line-name = 'FILETYPE'.
ptab_line-kind = cl_abap_objectdescr=>exporting.
GET REFERENCE OF filetype INTO ptab_line-value.
```

*Continues on next page*

*No portion of this publication may be reproduced without written consent.*

*Continued from previous page*

```
INSERT ptab_line INTO TABLE ptab.
ptab_line-name = 'DATA_TAB'.
ptab_line-kind = cl_abap_objectdescr=>changing.
GET REFERENCE OF text_tab INTO ptab_line-value.
INSERT ptab_line INTO TABLE ptab.

ptab_line-name = 'FILELENGTH'.
ptab_line-kind = cl_abap_objectdescr=>importing.
GET REFERENCE OF fleng INTO ptab_line-value.
INSERT ptab_line INTO TABLE ptab.
etab_line-name = 'OTHERS'.
etab_line-value = 4.
INSERT etab_line INTO TABLE etab.

TRY.
    CALL METHOD (class)=>(meth)
      PARAMETER-TABLE
        ptab
      EXCEPTION-TABLE
        etab.
    CASE sy-subrc.
      WHEN 1.
        ...
      ...
    ENDCASE.
  CATCH cx_sy_dyn_call_error INTO exc_ref.
    exc_text = exc_ref->get_text( ).
    MESSAGE exc_text TYPE 'I'.
ENDTRY.
```

specification of the memory area in an `ASSIGN` statement, where an error must be handled via examining `sy-subrc`:

```
ASSIGN (token) TO <fs>.
IF sy-subrc = 0.
  ...
ENDIF.
```

This second example shows the dynamic specification of a class and a static method in a `CALL METHOD` statement, where an error must be handled via the `CATCH` statement of a `TRY` block:

```
TRY.
    CALL METHOD (token1)=>(token2).
  CATCH cx_sy_dyn_call_error.
    ...
ENDTRY.
```

This third example shows the dynamic specification of the database table and the `WHERE` clause in a `DELETE` statement, where syntax errors must be handled via the `CATCH` statement of a `TRY` block. Furthermore, the case of an initial token for the `WHERE` clause must be especially handled; if not, all lines of the database should be deleted:

 ©2006 SAP Professional Journal. All rights reserved.

```
IF token2 IS NOT INITIAL.
   TRY.
      DELETE FROM (token1) WHERE (token2).
      CATCH cx_sy_dynamic_osql_error.
        ...
   ENDTRY.
ENDIF.
```

# Runtime Type Services — RTTI and RTTC

The ABAP Runtime Type Services (RTTS) comprise an object-oriented framework for getting information about data objects at runtime (Runtime Type Information, or RTTI) and for creating dynamic data types (Runtime Type Creation, or RTTC). The RTTS are implemented through a hierarchy of type classes that contain the methods for RTTC and RTTI.

In RTTS, the properties of data types are represented by attributes of *type objects*. For each kind of data type (elementary data type, structured type, table type, object type, and so on), there is a type class with special attributes for special type properties. The class hierarchy of the type classes corresponds to the hierarchy of the ABAP data types.

Type objects can only be created through the methods of the type classes. You can either call methods to get type objects that describe existing types or you call methods that create type objects for new types. Type objects are reused, meaning that for a given data type only one type object is created.

## Recommendation

Let's first look at the considerations involved in using RTTI, and then turn our attention to RTTC.

### *Using RTTI*

The RTTI abilities of RTTS should always be used wherever statement `DESCRIBE FIELD` is insufficient. This restricts the use of `DESCRIBE FIELD` to the simplest cases of checking elementary data objects. **Figure 1**

```
DATA: wa1     TYPE scarr,
      wa2     TYPE spfli,
      tdescr1 TYPE c LENGTH 1,
      tdescr2 TYPE c LENGTH 1,
      mess    TYPE string.

FIELD-SYMBOLS: <wa1> TYPE data,
               <wa2> TYPE data.

ASSIGN: wa1 TO <wa1>,
        wa2 TO <wa2>.

DESCRIBE FIELD: <wa1> TYPE tdescr1,
                <wa2> TYPE tdescr2.

IF tdescr1 = tdescr2.
  <wa1> = <wa2>.
ENDIF.
```

**Figure 1** A failed attempt at avoiding a runtime error

and **Figure 2**, for example, show that RTTI is required for checking structures in order to prevent a runtime error during assignments in a Unicode program.[9]

Figure 1 shows an inappropriate attempt to avoid a runtime error during the assignment between two field symbols of generic type `data` (see the guidelines for declarations in Part 2 of this article series). The `DESCRIBE FIELD` statement returns the value "u" for both field symbols, meaning "flat structure." Unfortunately, the structures are not convertible in Unicode programs and a runtime error occurs.

Figure 2 shows how the runtime error is successfully avoided using RTTI. Instead of using the `DESCRIBE` statements, the method `describe_by_data` of class `cl_abap_typedescr` returns type objects for the field symbols. Only if their types are the same, the same type object is returned and the reference variables `descr_ref1` and `descr_ref2` have the same contents. Of course, RTTI can be used in much more sophisticated scenarios than the one shown here.

[9] Remember the difference between a Unicode program and a Unicode system. A Unicode system is a single code-page system in which characters are encoded in Unicode character representation. While all ABAP programs on a Unicode system must be Unicode programs, Unicode programs can also be executed in non-Unicode systems.

### *Using RTTC*

The RTTC abilities of RTTS should always be used where the type constructing abilities of the statement `CREATE DATA` are not sufficient. You can do almost the same things with `CREATE DATA` as with `TYPES` or `DATA` for creating new elementary types or table types, but you cannot create new structures. For that purpose you can use the `HANDLE` addition to `TYPE` in the `CREATE DATA` statement. Here you can specify a reference variable pointing to any type object, especially structures that you have created via RTTC. With this ability, one of the last reasons for program generation becomes obsolete. **Figure 3** shows how you can create an internal table with a dynamic structured line type that serves as the target area of a fully dynamic `SELECT` statement.

```
DATA: wa1        TYPE scarr,
      wa2        TYPE spfli,
      descr_ref1 TYPE REF TO cl_abap_typedescr,
      descr_ref2 TYPE REF TO cl_abap_typedescr,
      mess       TYPE string.

FIELD-SYMBOLS: <wa1> TYPE data,
               <wa2> TYPE data.

ASSIGN: wa1 TO <wa1>,
        wa2 TO <wa2>.

descr_ref1 = cl_abap_typedescr=>describe_by_data( <wa1> ).
descr_ref2 = cl_abap_typedescr=>describe_by_data( <wa2> ).

IF descr_ref1 = descr_ref2.
  <wa1> = <wa2>.
ENDIF.
```

**Figure 2** Successfully avoiding the runtime error using RTTI

 ©2006 SAP Professional Journal. All rights reserved.

```
PARAMETERS: left  TYPE tabname DEFAULT 'SFLIGHT',
            right TYPE tabname DEFAULT 'SCARR'.

DATA: struct_type TYPE REF TO cl_abap_structdescr,
      table_type  TYPE REF TO cl_abap_tabledescr,
      comp_tab1   TYPE cl_abap_structdescr=>component_table,
      comp_tab2   TYPE cl_abap_structdescr=>component_table,
      comp1       LIKE LINE OF comp_tab1,
      comp2       LIKE LINE OF comp_tab2,
      select      TYPE TABLE OF edpline,
      sublist     TYPE edpline,
      from        TYPE string,
      first_on    TYPE abap_bool VALUE abap_true,
      tref        TYPE REF TO data.

FIELD-SYMBOLS <itab> TYPE STANDARD TABLE.

first_on = abap_true.
CLEAR: select, sublist, from.
READ CURRENT LINE LINE VALUE INTO right.

struct_type ?= cl_abap_typedescr=>describe_by_name( left ).
comp_tab1 = struct_type->get_components( ).

struct_type ?= cl_abap_typedescr=>describe_by_name( right ).
comp_tab2 = struct_type->get_components( ).

CONCATENATE left ' join ' right ' on ' INTO from.

LOOP AT comp_tab1 INTO comp1.
  CONCATENATE left '~' comp1-name INTO sublist.
  APPEND sublist TO select.
ENDLOOP.

LOOP AT comp_tab2 INTO comp2.
  READ TABLE comp_tab1 INTO comp1
    WITH TABLE KEY name = comp2-name.
  IF sy-subrc <> 0.
    APPEND comp2 TO comp_tab1.
    CONCATENATE right '~' comp2-name INTO sublist.
    APPEND sublist TO select.
  ELSE.
```

**Figure 3** Creating an internal table with a dynamic structure using RTTC

*Continues on next page*

*No portion of this publication may be reproduced without written consent.*

***Figure 3** continued*

```
    IF first_on = abap_false.
      CONCATENATE from ' and ' INTO from.
    ELSE.
      first_on = abap_false.
    ENDIF.
    CONCATENATE from left '~' comp2-name ' = ' right '~'
      comp2-name INTO from.
  ENDIF.
ENDLOOP.

struct_type = cl_abap_structdescr=>create( comp_tab1 ).
table_type  = cl_abap_tabledescr=>create( struct_type ).
CREATE DATA tref TYPE HANDLE table_type.
ASSIGN tref->* TO <itab>.

TRY.
  SELECT (select) INTO TABLE <itab> FROM (from).
    CATCH cx_sy_dynamic_osql_error.
      ...
ENDTRY.
```

The first part of Figure 3 shows mainly how an inner join can be constructed to form a dynamic token specification (see the earlier guidelines for dynamic token specification) for the `FROM` clause of a `SELECT` statement with the help of RTTI. This capability has been available since SAP Web AS 6.10 and is discussed in the article "Enhanced ABAP Programming with Dynamic Open SQL" (*SAP Professional Journal*, September/October 2001). What has been missing up until SAP NetWeaver '04 is shown directly in front of the `SELECT` statement in Figure 3.

As you can see, by passing the internal table `comp_tab1` that describes the components of the needed structure to static method `create` of class `cl_abap_strucdescr`, we create a type object for a structure. Using RTTC, we could use this type object behind `TYPE HANDLE` of statement `CREATE DATA` to create a structured work area. Here, we even go a step further and create another type object for an internal table of that line type and create such a table using `CREATE DATA`. Finally, we assign that table to a field symbol `<itab>` that can be used in the `INTO` clause of the following dynamic `SELECT` statement.

Prior to SAP NetWeaver '04, you had to work with program generation if you wanted to create target areas of appropriate data type or to write into containers made up from character or byte fields.

## Program generation

Dynamic program generation allows you to prepare complete ABAP programs as contents of internal tables with character type line type and to either create temporary subroutine pools in the internal session of a program via `GENERATE SUBROUTINE POOL`, or save the program persistently in the ABAP Repository of SAP NW AS ABAP via `INSERT REPORT`.

 ©2006 SAP Professional Journal. All rights reserved.

### Recommendation

Dynamic program generation is the most advanced but also the most labor-intensive level of dynamic programming. Use it only in cases where the other means of dynamic programming, especially dynamic token specification and the use of RTTS (see the earlier guidelines on dynamic token specification and Runtime Type Services) are not sufficient, because:

- Generated programs are difficult to test and maintain.
- Generated programs can be critical in regard to security issues, since the generated programs cannot be checked statically.
- In general, program generation is more time-consuming and memory-intensive than dynamic token specification. An exception to this rule can be the repeated use of statements with dynamic tokens or of RTTS, where the dynamic features are stable, meaning you use the same dynamic specifications several times. In such cases, the costs for interpreting and checking the same dynamic parts over and over again might exceed the cost of a one-time program generation. When in doubt, it can be helpful to perform runtime measurements with real-life data for both variants in order to find the best way.

Normally, dynamic program generation is reserved for experts. For experts, program generation can be helpful if the dynamic features of the program are relatively stable during the lifetime of a system. For example, you can generate correct and robust programs that depend on the settings of the system and that are not changed as long as the system settings are stable. However, in order to keep such programs synchronized with the system settings, you must additionally implement a mechanism that regenerates the dynamic programs if necessary. Furthermore, such programs should be connected to test tools such as ABAP Unit or Code Inspector (see the upcoming guidelines for testing programs) in order to guarantee their stability.

> ***Note!***
>
> As of SAP NetWeaver '04, you'll no longer need to utilize program generation for dynamic structure definition, since RTTC (introduced with that release as part of RTTS) offers the respective functionality. With the new RTTC method `CREATE` of the RTTS classes, you can create arbitrary type objects to be used in the `CREATE DATA` statement.

Unfortunately, no matter how well you use data storage and retrieval, dynamic programming, or any other ABAP features, errors are bound to occur. In the next sections, we will present recommendations for handling potential errors in your ABAP programs.

## Best practices for error handling

Software development and operation — and ABAP is of course no exception here — are intrinsically prone to errors in a number of ways:

- Internal errors may arise from incorrect implementations and improper use of underlying services.
- When there's interaction with external resources (such as the user, system memory, or disk storage), errors can occur due to invalid input or unexpected resource limitations.

In the next sections we'll look at how to handle the following common types of errors:

- Exceptions
- Assertions
- Messages (dialog and exit)

The ABAP language and infrastructure offers different ways for handling these kinds of errors, both those that are recoverable (exceptions and dialog messages) and those that are non-recoverable (assertions and exit messages). We'll show you the best approaches for each type.

No portion of this publication may be reproduced without written consent.

## Exceptions

ABAP offers different ways to handle recoverable exception errors:

- **Class-based (new) exceptions:** Class-based exceptions are available as of Release 6.10. They replace classic exceptions and catchable runtime errors. A class-based exception is a treatable (i.e., capable of being caught) exception, which is represented by an exception object of an exception class.[10] If an exception occurs, an object of an exception class is generated[11] and can be propagated across call levels. The propagation behavior depends on the kind of exception class. (See the sidebar "Kinds of exception classes" below.)

  There are predefined exception classes for exceptions of the runtime environment, and self-defined exception classes for custom applications. A class-based exception that is handled in the program does not lead to a runtime error. A class-based exception can be handled with a `CATCH` statement if it occurs in the `TRY` block of the respective `TRY/ENDTRY` control structure.[12]

[10] An exception class is a class that inherits from the predefined abstract class `CX_ROOT`.

[11] For performance reasons, the object is in fact only generated when it's actually addressed during exception handling in the program.

[12] For further information on this subject, see the article "A Programmer's Guide to the New Exception-Handling Concept in ABAP" (*SAP Professional Journal*, September/October 2002).

### Kinds of exception classes

There are three kinds of exception classes — *static check*, *dynamic check*, and *no check* — which are defined by the class's super class:

- **Static check** exception classes inherit from `CX_STATIC_CHECK`. Such exceptions can be propagated from a procedure if they are explicitly declared in the interface of the procedure. This is checked during compile time. Use this kind of exception if you want the caller of a procedure to explicitly take care of the error situation. If the exception can safely be avoided by additional checks beforehand, it might be better to define a dynamic check exception.

- **Dynamic check** exception classes inherit from `CX_DYNAMIC_CHECK`. For propagation from procedures, the same holds as for static check exceptions, with the difference that the declaration is not checked at compile time but only after an exception has actually occurred at runtime. Use this kind for exceptions a caller usually can avoid — either because they know that it can't possibly occur or because an appropriate check has been made before calling a certain routine. It is a good idea to document such preconditions for procedures. In a properly designed application, most exceptions will probably be of type dynamic check.

- **No check** exception classes inherit from `CX_NO_CHECK`. Such exceptions can be propagated from a procedure without explicit declaration in the interface of the procedure. Use this kind of exception for errors where you cannot expect a caller to handle it directly. Typical examples are exhausted resources, configuration problems, etc. Normally there should not be many situations in an application program where a no check exception is appropriate.

 ©2006 SAP Professional Journal. All rights reserved.

- **Classic (old) exceptions:** Classic exceptions can be defined in the interfaces of function modules and methods. The treatment of classic exceptions is made possible by the addition `EXCEPTIONS` of the statements `CALL METHOD` and `CALL FUNCTION`. Numeric values are assigned to the exceptions, which are used to fill the system field `sy-subrc` when the exception occurs. The actual error handling takes place after the call, when your program evaluates `sy-subrc`.

- **Catchable runtime errors:** Before Release 6.10, treatable exceptions for error situations of the runtime environment were defined exclusively as catchable runtime errors. All catchable runtime errors are predefined in the system and assigned to the ABAP statements during the execution of which they may occur. Several runtime errors can be grouped together in an exception group and can be handled together under the name of the group. The statement for handling catchable runtime errors before Release 6.10 was `CATCH SYSTEM-EXCEPTIONS`. As of Release 6.10, an exception class is assigned to each catchable runtime error and they can be handled as class-based exceptions.

## Recommendation

As a rule, use mainly class-based exceptions. The other exceptions (classic exceptions, catchable runtime errors) are only there for backward compatibility and are mostly superseded by class-based exceptions. In addition:

- Do not define classic exceptions in methods or function modules any longer.

- Classic exceptions still play a role in existing procedures, so you will have to deal with them when calling or modifying such procedures. It is good practice to catch classic exceptions and map them to class-based exceptions.

- Consider the use of `MESSAGE RAISING` instead of `RAISE` in existing procedures that are governed by classic exceptions. With `MESSAGE RAISING` you can pass textual information about the situation to the handler of the exception. Therefore, such messages can be seen as a precursor to exception texts in exception classes.

- Catchable runtime errors are now completely obsolete, since they have been replaced by corresponding exception classes. Instead of `CATCH SYSTEM-EXCEPTIONS`, use `TRY ... CATCH ... ENDTRY` only.

In the following sections, we will give you some hints on working with class-based exceptions.

### *Creating exception classes*

If you discover a situation where your program might create an error, adhere to the following procedure:

1. Check if an exception is appropriate:

   - In a situation where an exception is too strong, alternatives to exceptions are self-defined return codes or flags — showing if a read operation was successful or not, for example.

   - In a situation where a treatable exception is too weak — e.g., when no recovery is possible — alternatives to exceptions are assertions (see the upcoming guidelines for assertions) or exit messages (see the upcoming guidelines for messages).

2. Describe the situation. If an exception is justified, describe its important characteristics. The description should contain more semantic than technical information (such as the place in the code, for example). Violations of this rule can hinder the reuse of an exception and make handling more difficult if the exception is propagated.

3. Check for an existing exception class. Search for an existing exception class with text that describes the situation well, is of the right type (see the sidebar "Kinds of exception classes" on page 18), and has the appropriate attributes.

If an exception class has more attributes than needed, refrain from using it.

4. Refine an existing exception class. If you found an exception class that almost fits, think about refining it:
   - If the exception text does not describe the exact situation, define an additional text.
   - If you need additional attributes, consider inheriting from the existing exception class. But only do so if your error situation is really a special situation of the general error described by the super class.[13]
5. Create a new exception class. If you haven't found a reusable exception class, define a new one. The most important decision is to determine the right exception type: *static check*, *dynamic check*, or *no check* (see the sidebar "Kinds of exception classes" on page 18).

### *Handling class-based exceptions*

The general rule is to *catch selectively*. A handler should usually only catch exceptions it can handle properly and propagate all others to the next level of the call stack.

Catching exceptions by specifying the abstract root classes of the exception tree (`CX_ROOT`, `CX_STATIC_CHECK`, `CX_DYNAMIC_CHECK`, `CX_NO_CHECK`) is almost never a good idea, except at the highest levels of a software architecture (e.g., in some global dispatcher).

### *Propagating class-based exceptions*

Exceptions that can be propagated from a procedure are part of the procedure's parameter interface (see the guidelines for parameter interfaces of procedures in Part 2 of this article series). The interface should describe these exceptions from the caller's point of view, not the implementer's.

When you propagate exceptions in a call stack, you generally move from lower, more basic levels up to higher, more abstract levels. You should also support this abstraction when propagating exceptions by mapping implementation-specific exceptions to exceptions that are understood by the caller. Technically this means that you catch the implementation-specific exceptions and raise new exceptions that are a suitable abstraction. In order to reconstruct the chain of causation for the new exception, it is good practice to pass the old exception to the `previous` attribute of the new one.

> ***Note!***
>
> Each exception changes the normal flow of control and may present a serious risk for the consistency of an application. Therefore, be aware of the possibility of a `CLEANUP` clause in a `TRY` block. If necessary, use it to bring your application back into a consistent state.

## Assertions

Assertions are closely related to error handling; with assertions you express conditions that must hold true. They can be checked at runtime, and if they fail an error has been found. Assertions are available in ABAP as of Release 6.20, Support Package 29, via the `ASSERT` statement. An assertion is either always active or can be activated via the maintenance transaction SAAB. When a program reaches an active assertion, it evaluates the corresponding condition. If the condition is violated, the program terminates with a runtime error, accesses the ABAP Debugger, or creates a log entry. The behavior is controlled by the activation settings.

[13] Before inheriting from an exception class, in some cases it might even be useful to first create a new super class for related existing exception classes and to inherit the existing classes and your new one from that super class.

 ©2006 SAP Professional Journal. All rights reserved.

### Recommendation

Since assertions can be activated from outside a program and inactive assertions produce no additional load on an application, use them frequently. But do not use an assertion when an exception is required. Impose assertions only on conditions for program states that are fully under your control. Never formulate an assertion that is based on input parameters or the availability of some external resource. The violation of such conditions will result in an exception so that an external caller can catch it and react to it.

Assertions always must be implementation-specific. Only preconditions for internally used functionality can be checked by assertions (e.g., some private method of a class, which only you can use in your implementation). Assertions are typically used for invariants and post-conditions where you are simply stating that your implementation does what you specified.

If a productive system is in a normal state, *activatable* assertions are normally switched off.

> ***Note!***
>
> *Logging* provides a mechanism complementary to error handling with exceptions or assertions. While exceptions or assertions focus on changing the control flow in case of a locally detected error, logging addresses a broader view by leaving traces in critical situations. This enables a global analysis of system behavior, where the single events are put into a temporal context. If logging is necessary, use a framework that is generally available, such as the Application Log, which is based on function modules starting with `BAL_`. There's no need for you to create your own logging environment — as of SAP NetWeaver 2004s, there is also a dedicated ABAP statement `LOG-POINT` for logging purposes.

If checking is always crucial, you can use *always active* assertions. If the condition of an always active assertion is violated, the program terminates with the runtime error `ASSERTION_FAILED`. If you want a caller to be able to react in such a case, you must use an exception instead of an assertion.

## Messages

Messages are basically texts that can be displayed via the `MESSAGE` statement as *dialog messages* during screen (Dynpro) processing. Messages can be classified with a message type that defines the display type and the program behavior (message handling). Possible message types are termination message (A), error message (E), information message (I), status message (S), or warning (W). On the application logic side, messages can be used as a surrogate for exceptions that are combined with a text. This is made available by the predefined classic exception `ERROR_MESSAGE` in function modules or by using the `RAISING` addition with the `MESSAGE` statement. Furthermore, there is a special message type X. When classified with X, a message is an *exit message* that terminates a program via a runtime error.

### Recommendation

Messages are mainly intended for use during classic dialog processing involving Dynpros, and especially during the event PAI. During PAI processing, messages of type E and W permit an error dialog when used in connection with the Dynpro statement `FIELD`. Therefore, never use messages — and in particular messages of type E or W — in contexts other than PAI processing.

On the application logic side, do not use messages for exceptions — use class-based exceptions instead (see the earlier guidelines for exceptions). If existing procedures send messages, it is good practice to catch and map them to class-based exceptions. If an exception should be connected to an end-user-specific dialog text, you can implement the predefined interface `IF_T100_MESSAGE` in the exception class (as of SAP NetWeaver '04). After catching such an

No portion of this publication may be reproduced without written consent.

exception using `TRY ... CATCH ... ENDTRY`, you use the exception object directly in the `MESSAGE` statement to get a proper error message during dialog processing.

Since they terminate an application without any possibility of recovery, use exit messages of type X only with extreme care. In most cases throwing an exception or using an assertion is the better approach (see the earlier guidelines for exceptions and assertions for when to use which of these). Cases for exit messages could be the prevention of database inconsistencies or fatal errors in the user interface. In such circumstances, an exit message might be favored over assertions because it allows you to display a meaningful message text in the short dump resulting from the runtime error. For basic application functionality that is widely reused, however, never use exit messages, since they can hinder the reuse of parts of your application to a great extent.

Armed with the programming best practices you have learned, you are ready to put your newfound knowledge to use. Before you get started, however, there are a few administrative issues you need to keep in mind. We'll take a look at these next.

## Administrative issues

Implementing an application consists of much more than merely writing the code. Developers have additional tasks to fulfill, such as testing and releasing their programs. The remaining sections of this article provide recommendations for:

- Testing programs
- Documenting programs
- Using packages

## Testing programs

The ABAP development environment offers a rich variety of test tools. The most important are:

- **ABAP Unit:** ABAP Unit[14] is a test tool integrated into the ABAP runtime environment that can be used for checking the functions of code sections in a program (unit tests). The actual tests are implemented as test methods of test classes. Tests can be grouped together into test tasks.
- **eCATT:** The extended Computer Aided Test Tool (transaction SECATT)[15] is a tool that allows storing of application process flows for test purposes.
- **Memory Inspector:** The Memory Inspector (transaction S_MEMORY_INSPECTOR)[16] is a tool for displaying and analyzing memory snapshots. A memory snapshot contains information on all data objects and instances in an internal session along with their memory consumption. It is created in the ABAP Debugger or by entering /hmusa in the input field on the standard toolbar. As of SAP NetWeaver 2004s, you can also choose System → Utilities → Memory Analysis → Create Memory Snapshot.
- **Performance Analysis and Runtime Analysis:** The Performance Analysis (transaction ST05) and Runtime Analysis (transaction SE30) tools[17] help to find and fix performance problems.
- **Code Inspector:** The Code Inspector (transaction SCI)[18] is a tool that bundles several tools for the static analysis of ABAP programs, including performance, security, syntax, adherence to naming

[14] See the article "Attention ABAP Developers! Boost the Quality of Your Programs by Writing Automated Unit Tests with the ABAP Unit Test Tool" (*SAP Professional Journal*, January/February 2005).

[15] See the article "Extend the Range and Reduce the Costs of Your SAP Testing Activities with eCATT" (*SAP Professional Journal*, January/February 2003).

[16] See the article "Analyze Memory-Related Problems in Your ABAP Programs in Less Time and with Less Effort Using the ABAP Memory Inspector" (*SAP Professional Journal*, November/December 2004).

[17] See the articles "Performance Problems in ABAP Programs: How to Find Them" (*SAP Professional Journal*, May/June 2003) and "Performance Problems in ABAP Programs: How to Fix Them" (*SAP Professional Journal*, July/August 2003).

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

 ©2006 SAP Professional Journal. All rights reserved.

conventions, and ABAP Unit. Tests are organized as test variants.

- **Coverage Analyzer:** The Coverage Analyzer (transaction SCOV)[19] is a tool for recording how often programs or individual *processing blocks* are executed, classified by time period, user, or subject area.

For an overview of ABAP testing and error analysis, see the article series "An Integrated Approach to Troubleshooting Your ABAP Programs" in the March/April, May/June, and July/August 2004 issues of *SAP Professional Journal*.

## Recommendation

It is strongly recommended that you cover 100% of your coding by appropriate tests. Here's a list of test types, and the circumstances under which the use of each is appropriate:

- **Development tests with ABAP Unit:** Use ABAP Unit tests to cover all of your operational statements. This means all methods that are not connected to presentation logic should be tested via ABAP Unit. The execution of ABAP Unit tests is triggered with the Code Inspector (more on this in a moment).
- **Usage tests with eCATT:** Methods that are connected to events from the presentation logic must be tested by eCATT. Again, 100% test coverage of all such methods should be mandatory.
- **Memory tests with the Memory Inspector:** Use the Memory Inspector to track down memory leaks. For example, you can find objects not deleted by the garbage collector, because there are still unwanted references from event handler tables or frameworks, or from internal tables that use too much memory because they are inaptly designed.
- **Performance tests with Performance Analysis and Runtime Analysis:** Use these tools to find performance flaws.[20] During development it can also help to write little test programs in order to check small coding sequences directly with the statement `GET RUN TIME FIELD`.
- **Static tests with the Code Inspector:** For testing the compliance of ABAP code with the premises of the ABAP programming guidelines, a new variant should be created, based on the existing standard variant, that fulfills the following requirements:
  - Performance is tested as in the standard variant.
  - Security test is generally switched on (exceptions can be achieved via pseudo comments).
  - Syntax is always tested with the "Extended Program Check" (see the guidelines for writing correct ABAP in Part 1 of this article series).
  - Programming conventions are not tested.
  - Dynamic checks with ABAP Unit are switched on.
  - Screens (Dynpros, GUI Status) are checked for usability/accessibility.

  Do not release any program as long as any of these tests reports an error.
- **Dynamic tests with the Coverage Analyzer:** Use the Coverage Analyzer to check the test coverage. You can also use the Coverage Analyzer to check if a program is appropriately organized. You can recognize, for example, if a procedure is almost never used and could be outsourced to a service program.

[19] See the article "Improve Testing by Tracing ABAP Program Execution: Take a Closer Look with the Coverage Analyzer" (*SAP Professional Journal*, September/October 2002).

[20] See the articles "Performance Problems in ABAP Programs: How to Find Them" (*SAP Professional Journal*, May/June 2003) and "Performance Problems in ABAP Programs: How to Fix Them" (*SAP Professional Journal*, July/August 2003).

No portion of this publication may be reproduced without written consent.

# Documenting programs

The ABAP Workbench offers a complete set of documentation tools for all repository objects. In the area of programming, you can especially write documentation for:

- ABAP programs (function groups, executable programs, etc.)
- Global classes
- Methods of global classes
- Function modules

## Recommendation

If you implement coding — especially in classes or function modules — that can be used by programs other than your own, you must release them accordingly. The prerequisite for releasing a class or a function module is the documentation of its public (and protected) interfaces. After release, all interfaces must be kept stable.

All programs and program interfaces must be documented via the documentation capabilities of the ABAP Workbench. This holds especially for global classes and function modules. If the name of a (public) component and its short text are sufficient, you may not necessarily need to provide a long text for that component. For example, if a global class contains an attribute X, a method `GET_X` is sufficiently documented by a short text such as "Returns attribute X," while a method `CALCULATE_SOMETHING_FROM_X` might require a longer text. If a class, the components of a class, or a program are described somewhere else (in the SAP Knowledge Warehouse, for example), you must link from the specific documentation to this documentation.

# Using packages

Prior to Release 6.10, development classes were the only SAP-provided means for organizing repository objects. Their main purpose was to group repository objects of development projects and to connect them to the Transport Organizer. As of Release 6.10, development classes have been replaced by *packages*. When you create a repository object, you must assign it to a package. Packages are containers of (closely related) development objects with some additional semantics. Packages have the same transport-related properties as development classes:

- Original system
- Transport layer
- Software component

Packages enhance the features of development classes, and allow you to control access to repository objects at design time. SAP plans to enhance the access control into the runtime realm in an upcoming release. Packages can be nested; nesting of packages means layering — an embedded package can't access the contents of its surrounding packages (this limitation will be overcome with the planned new package concept).

## Recommendation

When you create and use a package, leverage the new features that are available as of Release 6.10. Without further specification, a package is no more than a simple development class. Therefore, for new packages, the "Package check as server" checkbox should be selected. This ensures that only objects of package interfaces defined for this package are usable outside the package. In order to use the objects contained in a package interface, its client packages have to create *use accesses* to the used package interface.

Be as specific as possible when assigning an application component to a package during package definition and when you assign repository objects to a given package. The reasons for this recommendation are:

- A package defines the deployment path of its repository objects.

 ©2006 SAP Professional Journal. All rights reserved.

- The component is important for the maintenance of the package's repository objects since it is used for dispatching OSS messages.
- A package controls the access to its repository objects.

Organize your repository objects in packages in such a way that you have a strong cohesion of repository objects within a package and a loose coupling between repository objects of different packages.

By all means, avoid the creation of objects in a package that isn't original in the current system. If an urgent repair is needed, always create the object in the original system of the package first. Then, the object can be created as a copy of the original. The implementation of the new object must be semantically identical in both systems.

***Note!***

The term "package" as it is used here is ABAP-specific and is not the same as a Java package. Within the context of SAP NW AS, the closest relatives to ABAP packages are Java development components (which can consist of a number of Java packages and other deliverables, as well as "public parts" that identify the objects that can be used externally and "usage relations" to the public parts of other development components), where the public parts of development components are similar to the package interfaces of an ABAP package.

## Conclusion

Let's conclude this third and final installment of our article series by summarizing the most important best practices and administrative issues that are necessary to write up-to-date ABAP programs, from our point of view:

- Use relational database tables for general persistent data storage, but be aware of other techniques for special purposes.
- Exploit shared objects in the shared memory, but keep to the recommended scenarios — use them as a shared buffer or as an exclusive buffer.
- Leverage the concepts of dynamic programming according to your needs — using language elements that are overdone for a given task is tantamount to swatting a fly with a sledgehammer.
- Distinguish clearly when to use which means for error handling — return values, exceptions, assertions, or messages.
- Test all parts of your programs with the appropriate tools.
- Document your programs.
- Leverage the full concept of packages.

In the three parts of this article series, we have presented a large number of ABAP programming guidelines, reaching from the general to the very specific. We are aware that this collection of recommendations represents our personal view of how things should be done. We are also aware that our collection may be overly heterogeneous and is almost assuredly incomplete. Therefore, if you have more or even better ideas, or if you just want to tell us why one or the other of our rules isn't feasible, don't hesitate to contact us.

Let's conclude this article series by summarizing what it means to write up-to-date ABAP programs, from our point of view:

- Keep your coding structured — always use ABAP Objects.
- Keep your coding clean — don't use language elements that are obsolete.

No portion of this publication may be reproduced without written consent.

- Keep your coding modern — keep an eye on ABAP language news and apply new constructs in your coding as they become available. And yes, it might be worth the effort to rewrite some of your outdated application programs.

Incorporating the recommendations presented in this three-part article series into the guidelines you already have in place is bound to not only improve the quality of your programs, but also make your programming life much easier — a double win for your organization and for you.

## Acknowledgements

The authors want to express their gratitude to the following for their help in preparing this series of articles: Gerd Kluger, who contributed to the section on error handling, Jürgen Lehmann, who helped in the early preparation and participated in many discussions, and David F. Jenkins, who did a terrific job in editing this article and thereby helped to transform the first draft versions into something that could be published.

*Andreas Blumenthal studied linguistics, mathematics, philosophy, and Jewish studies at the University of Heidelberg, Germany, where he received his Master's degree in 1985. After two years of research in the area of computer linguistics, he joined SAP in 1987. Andreas has participated in the R/3 project from the beginning. Working in the R/3 technology department, he has considerably contributed to the growth of ABAP into a modern, object-oriented programming language for business applications. Andreas became Development Manager of the "ABAP Language Group" in 1996. He is currently a Vice President and responsible for SAP NetWeaver Application Server ABAP.*

*Horst Keller holds a PhD in physics from the Technical University of Darmstadt, Germany. He joined SAP in 1995 after spending several years involved in research projects at various international institutions. As a Knowledge Architect within the SAP NetWeaver Application Server ABAP department, he is mainly responsible for the documentation and rollout of ABAP and ABAP Objects, while also developing the programs for formatting and presenting the ABAP documentation, including the related search algorithms. Horst is the author of the books "ABAP Objects – An Introduction to Programming SAP Applications" (Addison-Wesley Professional, 2002), "The Official ABAP Reference" (SAP PRESS, 2005), and the "ABAP Quick Reference" (SAP PRESS, 2005). Numerous other publications and workshops on this subject round off his work.*

 ©2006 SAP Professional Journal. All rights reserved.