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

Part 1 — fundamental rules and formal criteria

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.

You know the situation — you need to accomplish some given task, and you're going to need to write some ABAP code to do it. If you're new to ABAP, you research the available ABAP language documentation and you are very likely overwhelmed by the wealth of information and possibilities you find. How do you approach your search for ABAP guidance? Do you delve deeper into the subject and try to find out the best way to do it, or do you just look for the first snippet of code — maybe in existing programs — that seems appropriate and simply cut and paste? And if you're an experienced ABAP developer, you'll probably solve your task in the same way you have done it for years now. And in both situations — ABAP neophyte or graybeard — in the end you'll probably be satisfied if your coding somehow just works. Why should you care about alternatives or new features of ABAP if your primary task has been fulfilled? This article is intended to provide an answer to this question.

The need for an ABAP programming guide as presented here stems, for the most part, from the fact that ABAP is a vibrant, evolving language. Although you may have been able to program in ABAP for almost 30 years, the development of the language itself has always continued in parallel. New developments in ABAP continue — either through additions to the existing feature set or replacement of (and

thereby making superfluous or obsolete) current features.

The issue of superfluity and obsolescence is often especially troublesome for programmers new to the world of ABAP. For instance, why are there two ABAP programming models — object-oriented and procedural? Why are there different statements to accomplish the same task, such as `FIND` and `SEARCH`? More experienced ABAPers may not be troubled by these available choices — they are comfortable with the feature set that was available when they learned ABAP, so why should they be concerned about evolving changes to the language? "I am used to internal tables with header lines — why should I change that?" is a statement that we often hear. Unfortunately for the ABAP language (but lucky for your existing programs!) SAP's practice of maintaining strict downward compatibility prevents the total extinction of obsolete features from the language. It is an objective of this article to show you a way through that jungle of possibilities.

This is the first part of a three-part article series in which we present basic guidelines for the effective use of the many possibilities ABAP provides to application developers to create business applications. The aim of these ABAP programming guidelines is to support the development of ABAP programs that are:

- Correct
- Maintainable
- Well-structured
- High-performing
- Readable
- Up-to-date
- Robust

Our guidelines cover formal as well as conceptual rules for ABAP programs. It is not the aim of this article series to provide a general programming guide that is independent of the programming language — it is an ABAP-*specific* programming guide. (Nevertheless, some language-neutral aspects, such as programming style, will also be covered. Our examples will show, for instance, our firm belief in the language-independent KISS — Keep It Simple Stupid — principle, based on Occam's Razor.[1])

We are aware that a wealth of ABAP guidelines already exists — both inside and outside of SAP. Such guidelines stretch from mere naming conventions to elaborate how-to guides, and are often specialized for discrete development areas. The guidelines and best practices we present here come directly from the ABAP Language Group[2] at SAP. Although there have been many requests for the ABAP Language Group to make available such guidelines, we have heretofore refrained, because we have felt that the preparation of guidelines was simply not our task — since we're not application specialists we cannot tell developers how to develop their applications. In the meantime, however, the language has grown to the point where we now feel responsible for not only providing new language features, but also presenting our ideas on the best practices for the use of those features. We see in our daily experiences (for example, in the ABAP language hotline service) that many developers stick to their old habits, even when creating new programs. We want to present guidelines that follow a very basic approach that mirrors the basic role of the *ABAP* Language Group (as opposed to SAP and non-SAP application development groups). Therefore, this collection is personally biased — it is based on our view of the language. Moreover, we do not claim that our guidelines are complete or that they cover all questions of ABAP programming.[3] Nevertheless, it is our hope that the ideas we present here will serve as a foundation for the development of your own more specialized guidelines for development projects or will give your developers useful hints

[1] For more on this principle, see http://en.wikipedia.org/wiki/Occams_Razor.

[2] *ABAP Language Group* denotes the development teams at SAP that deal with the ABAP language itself. This group mainly consists of two teams. The *core team* develops internal stuff such as the compiler, internal tables, ABAP Objects, Shared Objects, and so on. The *TSI team* develops tools, services, and interfaces such as Open SQL, ABAP test tools, the XML interface, and so on. The ABAP Language Group is part of the organizational unit *SAP NetWeaver Application Server ABAP*, which includes additional teams such as the *ABAP Workbench* and *Connectivity* teams.

[3] For example, we do not cover general performance guidelines.

for their daily work in cases where no guidelines currently exist.

In addition to *guidelines*, program *standards* play a vital role. Your organization may have already defined product standards for software development. If so, you may be obliged (for legal reasons, perhaps) to adhere to such standards. We urge the additional adoption of the programming guidelines defined in this article series to complement and support the standards you may already have in place. For example, SAP itself has a rather voluminous set of official product standards (regarding functional correctness, performance, security, usability, accessibility, globalization, etc.) that must be adhered to in SAP's own development departments.

In summary, the guidelines in this article series form a collection of ideas and hints regarding the proper and effective use of the ABAP language. The three parts focus on the following subject areas:

- Part 1

  Fundamental rules and formal criteria

- Part 2

  Best practices

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

- Part 3

  Best practices

  - Data storage and retrieval
  - Dynamic programming
  - Error handling

  Administrative issues

  - Testing programs
  - Documenting programs
  - Using packages

  In this first article, we'll lay some groundwork by first introducing fundamental rules for modern ABAP programming. These rules will serve as a general framework for the rest of this article (and the two articles to follow in upcoming issues of *SAP Professional Journal*). We'll follow that discussion by providing formal criteria for writing ABAP programs that support the aims listed in the introduction.

In the sections that follow, we present each topic by first setting the context in which the guideline might apply, followed by our specific recommendation. It is our belief that all ABAP programmers will benefit from following these guidelines in new and ongoing ABAP development projects.[4]

Note!

This article 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.

Fundamental rules

The guidelines presented in the following sections describe the kind of programming preferred when starting a new project or redesigning an existing one, including:

- Using the object-oriented vs. procedural programming model
- Program execution and invocation
- GUI programming

[4] In order to facilitate the application of these (and perhaps other) programming conventions, a new SAP check tool is planned for future development; such a tool will assist in ensuring adherence to guidelines and standards.

Note that some guidelines later in this article assume that you are following our first recommendation, presented in the following section.

Using the object-oriented vs. procedural programming model

ABAP supports a hybrid programming model. You can use an object-oriented (OO) programming model based on classes and interfaces, and you can use the more classic procedural and event-driven programming model based on function modules, subroutines, dialog modules, and event blocks (more on this in the upcoming guidelines for processing blocks). Both models can be used in parallel. You can use classes inside classic processing blocks or you can call classic procedures from methods.

In ABAP Objects, SAP has implemented a cleanup of the ABAP language. Within the scope of this language cleanup, stricter syntax checks are performed in classes that restrict the use of obsolete language elements. The stricter syntax checks usually result in a syntax that should also be used outside of ABAP Objects (in procedural programming) but for compatibility reasons older versions of the syntax are not forbidden. The stricter syntax checks comprise:

- Prohibiting many obsolete statements and additions
- Requiring many implicit syntax completions to be explicit
- Detecting and preventing potentially incorrect data handling

You can find a complete list of all obsolete language elements that are forbidden in ABAP Objects in the ABAP keyword documentation.

Recommendation

The object-oriented programming model provided by ABAP Objects is superior to the procedural programming model, since it offers better means for:

- Data encapsulation
- Explicit object instantiation
- Improved code reuse via inheritance
- Standalone interfaces
- Explicit event raising and handling

In addition to these five general reasons for using ABAP Objects, three more specific reasons apply.[5] ABAP Objects is:

- Simpler to learn and use than procedural and event-driven ABAP
- Has a stricter syntax check that prevents the use of obsolete language elements
- Provides access to new ABAP technology

Therefore, we recommend as our first and most fundamental guideline the use of ABAP Objects wherever possible. This means that all coding should be implemented in global or local classes and that global or local interfaces should be used when appropriate. Exceptions to this rule include the reuse of services that are implemented in existing function modules or the implementation of new function modules where technically necessary (you still must create function modules for remote invocations via RFC, and we recommend function modules for handling classic Dynpros defined in function groups, as discussed in the upcoming guidelines for GUI programming). Nevertheless, all operational statements[6] relevant for an application should be implemented in methods. As a blanket rule, subroutines (`FORM` … `ENDFORM`) should never be used for implementing new operational statements. Instead, use local classes for internal services. This holds especially for the internal modularization of

[5] You can find a detailed description of these eight reasons and a comparison of ABAP Objects to procedural ABAP in the article "Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Programmer Should Give It a Second Look" (*SAP Professional Journal*, September/October 2004). As stated in that article, we believe "…that programming with ABAP Objects means programming in a modern style that exploits the benefits of a paradigm that was invented to solve the problems of complex software projects — object orientation."

[6] With "operational statements" we denote all coding that is not declarative. Operational statements comprise all statements that can be executed. In ABAP, an operational statement is always part of the implementation of a processing block.

class pools,[7] where methods of local classes are preferable to private methods of the global class itself (more on this in Part 2 of this article series when we discuss ABAP Objects, modularization, and program flow).

Program execution and invocation

Program execution/invocation in ABAP means the loading of a program into memory and the execution of one, some, or all of its processing blocks. Programs can be executed as follows:

- **Standalone program execution:** In standalone program execution, an ABAP program is loaded as the main program into its own internal session[8] and the internal session exists for as long as the main program is being executed. Standalone program execution is carried out either by calling a transaction (ABAP statements `CALL TRANSACTION` or `LEAVE TO TRANSACTION`) or by submitting an executable program (ABAP statement `SUBMIT`).
  - A *transaction* is defined by connecting a transaction code to a program. You can either define dialog transactions by connecting transaction codes to the Dynpros of a program, or define object-oriented transactions ("OO transactions") by connecting transaction codes to the methods of a program. In the first case, program flow is controlled by the Dynpro flow logic; in the second case, it is controlled by the method's implementation.
  - An *executable program* is a special kind of program that is controlled by the reporting process of the ABAP runtime environment. It contains event blocks as handlers for reporting events that are triggered from the reporting process. Reporting events are `START-OF-SELECTION`, `GET`, and `END-OF-SELECTION`.
- **Invoked program execution:** In a dependent program execution, a program that is already loaded into an internal session invokes a procedure of another program that has not yet been loaded. Before the procedure is executed, its program is loaded in to the caller's internal session. External procedure calls are possible for methods of global classes, function modules of function groups, and subroutines of any ABAP program.

Recommendation

According to our fundamental guideline in the previous section (i.e., use ABAP Objects wherever possible), you work mainly in an object-oriented environment. There, program execution/invocation should consist essentially of calling a method.

- For standalone program execution, use object-oriented transactions — i.e., transaction codes that are connected to a "main" method of a global or local class. You might also start an executable program where a main method is called in the `START-OF-SELECTION` event block (an example of this is shown in the upcoming guidelines for processing blocks). But the latter should be restricted to programs that must be able to execute in background jobs.
- For invoked program execution, call methods of global classes. You can also call existing services that are available as function modules from your methods, but you should never call subroutines externally (more on this in the guidelines for processing blocks).

GUI programming

ABAP offers various possibilities for GUI programming. Classic GUI techniques that are embedded in the language are:

- **Classic Dynpros:** A classic Dynpro is a component of an ABAP program that is created using the

[7] A *class pool* is an ABAP program that contains the definition of a single global class and can contain an arbitrary number of local classes. The class pool of a global class is generated automatically from the Class Builder tool of the ABAP Workbench when the class is created. The counterpart of class pools for global interfaces are *interface pools*.

[8] An internal session is a memory area in which the data and objects of an ABAP program are stored during its execution.

Screen Painter of the ABAP Workbench. It is called either via a transaction code (dialog transaction) or `CALL SCREEN`. Calling a Dynpro always starts a Dynpro sequence.[9]

- **Selection screens:** Selection screens are special forms of Dynpro generated from a set of respective ABAP statements. They either are called automatically when executing executable programs, or can be specifically invoked via `CALL SELECTION-SCREEN`.

- **Classic lists:** Classic lists provide the means for structured and formatted output of data. The formatted data is stored via special ABAP statements (e.g., `WRITE` and `FORMAT`) in a list buffer and is displayed in a dedicated Dynpro screen when it is called. A list is called either automatically when executing an executable program or via `LEAVE TO LIST-PROCESSING`.

In addition to these classic GUI techniques, there are newer techniques now available to the ABAP developer — these are handled via object-oriented interfaces:

- **GUI controls:** GUI controls are components of the presentation server that are addressed from ABAP via ABAP Objects proxy classes of the Control Framework (CFW). GUI controls are still bound to classic Dynpros.[10]

- **Business Server Pages (BSPs):** BSPs are the ABAP counterpart of Java Server Pages (JSPs).[11] A BSP page is an HTML page with partly dynamic content. The dynamic content is provided by embedding server-side scripting written in ABAP. On the application server, a script appears as a generated ABAP Objects class.

- **Web Dynpro ABAP**: Web Dynpro[12] is a technology used to create platform- and language-independent Web-based interfaces. The architecture of Web Dynpros is based on the Model-View-Controller (MVC) approach.[13] The Web Dynpro technology is available in SAP NW AS Java as well as in SAP NW AS ABAP (as of SAP NetWeaver 2004s).[14] ABAP Web Dynpros are created with the Web Dynpro Explorer of the ABAP Workbench. Compared to classic Dynpros, an ABAP Web Dynpro is not a component of an ABAP program but is used via class-based proxy objects.[15]

Recommendation

The basic rule for programming user interfaces is to separate presentation logic from application logic. Do not mix coding that manages screen flow or screen appearance (as well as event handlers for screen and control events) with screen-independent application or business coding. Only application logic that is independent from the presentation logic allows:

- Access to business logic using different UIs
- Access to business logic using different automated clients (i.e., non-human users)

[9] A *Dynpro sequence* is started via a transaction code or the statement `CALL SCREEN`. The sequence in which Dynpros are processed is determined by the "Next Screen" specified for each Dynpro in the sequence. The Next Screen is part of a Dynpro's static properties and can be overwritten in the program. A Dynpro sequence ends when the Next Screen of a Dynpro has the number 0.

[10] See "SAP Controls Programming Essentials — What Every ABAP Developer Now Needs to Know" (*SAP Professional Journal*, November/December 2001).

[11] For further information on this subject, see "A Developer's Guide to Creating Powerful and Flexible Web Applications with the New Web Application Builder" (*SAP Professional Journal*, January/February 2002).

[12] For further information on Web Dynpro ABAP, see http://help.sap.com → Documentation → SAP NetWeaver → SAP NetWeaver 2004s → SAP NetWeaver Library → SAP NetWeaver by Key Capability → Application Platform by Key Capability → ABAP Technology → UI Technology → Web Dynpro for ABAP.

[13] The MVC triad of classes is typically used to build user interfaces, and consists of three objects: the *Model* is the application object, the *View* is its output representation, and the *Controller* defines the way the interface reacts to user input.

[14] SAP NW AS is the successor to SAP Web AS 6.40, and serves as the central application platform of SAP NetWeaver 2004s. It comprises both SAP NW AS ABAP and SAP NW AS Java. While SAP NW AS ABAP is evolved from the former SAP Basis, SAP NW AS Java is based on J2EE.

[15] When programming via (network) connections, a proxy object is an object on the client side that acts on behalf of another object on the server side. Working with proxy objects allows the developer to deal with remote objects in the current programming language as if they were available in the current memory.

This "separation of concerns"[16] is a basic programming model for supporting stability and maintainability because it allows:

- Programming of reusable services
- Testing of business logic by using isolated unit tests (more on this in Part 3 of this article series when we discuss testing programs)

Let's look at the recommendations for each of the available techniques in turn.

Web Dynpro

Since the Web Dynpro technology is based on the MVC model, separation of presentation logic from application logic is automatically achieved. Therefore, as of SAP NetWeaver 2004s, Web Dynpro ABAP is clearly the state-of-the-art technology for business application programming in ABAP. If available, it should be used for all new projects.

Business Server Pages (BSPs)

If Web Dynpro ABAP is not yet available in your system, or if you must create tailor-made UIs, you can use BSPs for Web-based applications. Compared to the Web Dynpro technology, the BSP technology is much more fundamental — you have to take care of the separation of presentation logic from application logic yourself. The Web Application Builder of the ABAP Workbench supports you in building BSP pages that are based on the MVC model[17] and it is highly recommended that you do so. But even when creating a BSP application based on MVC, you still have a lot of choices for implementing the model. Therefore, as of SAP NetWeaver 2004s, BSPs can be considered predecessor technology for general Web-based applications. For new applications that contain more than one screen, consider using Web Dynpro ABAP, where a special realization of the MVC model has already been prepared for you. The use of BSPs should be restricted to special cases where your Web application is based on single BSP pages and where scripting is needed because the functionality cannot be implemented via Web Dynpro.

Classic Dynpros

Unlike Web Dynpro and BSPs, there is no direct technical support for the separation of the presentation logic from application logic for classic Dynpros in the ABAP Workbench or via a framework such as the MVC model. Due to the lack of a framework that supports a "separation of concerns," the implementations of many Dynpro-based applications — classic dialog transactions — are constituted from ABAP programs (module pools) where presentation logic and application logic are hopelessly intermingled. Therefore, Web Dynpro ABAP should clearly be seen as the successor technology not only to BSPs, but to classic Dynpros as well — even for desktop applications that aren't necessarily Web-based.

Since the programming model of Web Dynpro is predefined, there may be special cases where screen programming with Web Dynpro is not sufficient to meet the needs of the application. Examples for such applications can mostly be found when working in a more technical environment — when programming an infrastructure, for example. The ABAP Workbench itself is a prominent example. For programming IDEs such as the ABAP Workbench, and when an application is accessed via the classic SAP GUI on a presentation server only, classic Dynpros (including selection screens) can still be used.

If you must program using classic Dynpros, it is your responsibility to separate presentation logic from application logic to every extent possible. You can do so by following these recommendations:

- Include no operational statements in dialog modules. Use dialog modules for screen handling only. You might even reduce the implementation

[16] The concept of "separation of concerns" (originally from Edsger W. Dijkstra, "On the role of scientific thought") provides a fundamental technique for coping with complexity. A *concern* is any coherent issue in the problem domain. It is important to identify and encapsulate those parts of software that are relevant to a particular purpose, such that each addresses only one or more concerns. Software can be then decomposed and organized into more manageable parts, resulting in greater comprehensibility, reusability, scalability, and maintainability.

[17] See the *SAP Professional Journal* articles "Build More Powerful Web Applications in Less Time with BSP Extensions and the MVC Model" (March/April 2003) and "Develop More Extensible and Maintainable Web Applications with the Model-View-Controller (MVC) Design Pattern" (January/February 2004).

User interface accessibility considerations

User interfaces must be accessible to every person. At SAP, accessibility is part of the set of defined product standards — the screen layout must be designed in such a way that all necessary information is accessible to all users (including, for example, blind users) with the aid of assistive technologies (screen readers). The most important points of SAP's accessibility standard are the following:

- All fields must have meaningful labels assigned to them.
- All table columns must have a column header.
- All icons must have an associated quick info pop-up.
- Color must *not* be used as the only way of representing information.
- Use frames with meaningful titles in order to group fields on a screen.

Accessibility is another strong reason for using modern GUI techniques versus more classic ones. For example, an ALV list is in most cases automatically accessible and therefore superior to a classic list, where you must code all the relevant formatting yourself. The same holds for Web Dynpro ABAP versus classic Dynpro — with Web Dynpro ABAP, the Web Dynpro Explorer of the ABAP Workbench generates accessible Web pages automatically; with classic Dynpro (and also BSPs), you must take care of each relevant screen element yourself.

of dialog modules to an absolute minimum by programming the screen handling in methods of additional local classes.

- Write all coding necessary for classic Dynpros in dedicated framework programs that are separated from the application programs. Since Dynpros are not supported in global classes, the ABAP program type that is most appropriate to encapsulate classic Dynpros are function groups, since they can contain Dynpros as components, and they offer function modules as a defined public interface.[18]
- Classic Dynpro fields require global data objects in the framework program (function group); reduce the number of necessary global variables in the function group to the absolute minimum. In order to fulfill the separation of presentation logic from application logic on the level of data types and data objects, for each Dynpro declare a dedicated structure in the ABAP Dictionary that represents the fields and semantic properties for that Dynpro, instead of using the structure definitions of database tables directly (more on this in Part 2 of this article series when we discuss type definitions and data declarations). The only global variables declared in the function group should be the structures for the Dynpros encapsulated by the function group.
- Take advantage of the object-oriented services provided for functionality of the presentation server. Make use of wrapper classes, such as those of the CFW[19] for working with SAP GUI controls, or those of SAP Desktop Office Integration (DOI)[20]

[18] For an example, see the appendix to the article "Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Programmer Should Give It a Second Look" (*SAP Professional Journal*, September/October 2004).

[19] See "SAP Controls Programming Essentials — What Every ABAP Developer Now Needs to Know" (*SAP Professional Journal*, November/December 2001).

[20] For a complete discussion of SAP DOI, see "SAP Desktop Office Integration (SAP DOI) — An Easier Way for ABAP Programmers to Integrate Desktop Applications with R/3" (*SAP Professional Journal*, March/April 2000).

for calling Microsoft Office applications instead of using the statements for OLE2 directly.

Classic lists

There's really no longer any need to use classic list programming (and to live with any of its limitations, such as a lack of user flexibility in output formats). Instead, always use SAP List Viewer (ALV),[21] text controls, or even tree controls — whatever is appropriate.

Formal Criteria

The remaining sections in this article provide an overview and recommendations on general program settings and layout, including:

- Program attributes
- Processing blocks
- Source code sequence
- Source code organization
- Coding style
- Using modern ABAP features
- Writing correct ABAP
- Internationalization

As stated earlier, some of the guidelines in these sections are predicated on the assumption that you are following our first and foremost recommendation — to use ABAP Objects wherever possible.

Program attributes

Each ABAP program has attributes that influence the program's execution behavior:

- **Type:** Defines which processing blocks a program can contain and how the program is executed by the ABAP runtime environment.
- **Logical database:** Links an executable program to a logical database. A logical database is a special program that provides the executable program with data from the nodes of a hierarchical tree structure. It consists of subroutines that are called by the ABAP runtime environment during execution of the executable program.
- **Unicode checks active:** Defines a Unicode program in which tightened syntax checks are executed and in which certain statements show different semantics compared to those that apply in non-Unicode programs. A program of this kind usually returns the same results in a Unicode system as in a non-Unicode system.[22]
- **Fixed point arithmetic:** Determines whether the decimal point is considered in operations for numbers of type p.

Recommendation

Follow these guidelines when working with program attributes:

- **Type:**
  - With ABAP Objects (remember our first fundamental rule), you work mostly with *class pools* and *interface pools*. The appropriate program type is set automatically by the Class Builder.
  - When working with classic Dynpros (refer back to the guidelines for GUI programming), you work mostly with *function pools*. The program type is set automatically by the Function

[21] For more on ALV, see the *SAP Professional Journal* articles "A Developer's Guide to the New ALV Grid Control" (November/December 2000) and "Take a Fresh Look at the Redesigned SAP List Viewer in SAP NetWeaver '04: Write Programs to Present Tabular Data in Less Time and with Fewer Lines of Code" (November/December 2005).

[22] 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. The system code page for SAP NW AS ABAP is currently UTF-16 with a platform-dependent byte order. All ABAP programs on such an application server must be Unicode programs (i.e., the Unicode check must be active). Nevertheless, Unicode programs can also be executed in non-Unicode systems.

Builder. The same holds if you want to provide functionality to users outside of SAP NW AS ABAP via RFC.

- If you want to program functionality in local classes, where a method is to be invoked via a transaction code, you create the respective program directly in the ABAP Editor or the Object Navigator. We recommend that you use the program type *subroutine pool* for this purpose since a subroutine pool is simply a program that has no additional implicit properties. Compared to module pools, a subroutine pool does not support Dynpros (for which you use function groups) and, therefore, cannot be started via a dialog transaction. Compared to executable programs, a subroutine pool does not support reporting events and cannot be started via `SUBMIT`, which is connected to the reporting process. The only procedures that are syntactically allowed in a subroutine pool are subroutines and methods of local classes, and according to our guidelines (refer to the upcoming guidelines for using processing blocks), you should use only methods and refrain from using subroutines. Such a subroutine pool, containing only methods, can be directly compared to a Java program that can be executed by calling its main method.
- Use *executable programs* only where technically necessary — for running in background jobs, for example. In such cases, the executable program should contain no operational statements outside local classes. The event block `START-OF-SELECTION` should be implemented as a mere entry point, where a method of a local class is invoked (again, refer to the upcoming guidelines for using processing blocks).

• **Logical database:** Do *not* use a logical database as a program attribute. Logical databases are based on subroutines instead of methods, and the behavior of an executable program connected to a logical database is governed by processes of the ABAP runtime environment that involve data sharing between programs, implicit subroutine calls, and the implicit triggering of events. These concepts are in contradiction to the benefits of the object-oriented programming model — data encapsulation, use of methods, and explicit triggering of events. Instead of creating new logical databases, consider offering the respective services in an appropriate global class.[23] If you want to use an existing logical database, you might instead call it explicitly via the function module `LDB_PROCESS` from within a method.

• **Unicode checks active:** This attribute should always be set, regardless of whether the program will be executed in a Unicode system. The rules for Unicode largely enhance the quality and maintainability of programs in the following ways:[24]

- Static checks are stricter.
- Byte and character strings are processed separately.
- Structures are handled appropriately according to their layout.
- Uncontrolled memory manipulation is not permitted.

> ***Note!***
>
> For the remainder of this article we assume that you are working with Unicode programs only — obsolete or dangerous language elements that are forbidden in Unicode programs will not be discussed further.

• **Fixed point arithmetic:** This attribute should *always* be set. Without it, the position of the

[23] For an example, see Figure 18 in the article "Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Programmer Should Give It a Second Look" (*SAP Professional Journal*, September/October 2004).

[24] For a complete list of syntax and semantics changes in Unicode programs, see the respective table in "Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Programmer Should Give It a Second Look" (*SAP Professional Journal*, September/October 2004).

decimal point of packed numbers is not taken into account, except for formatting output for Dynpros or with the statement `WRITE TO`. If you want to work with packed numbers without a fractional portion, you can easily do so by declaring them with the addition `DECIMALS 0`.

Processing blocks

Every ABAP program consists of *processing blocks*. Each non-declarative statement of an ABAP program is part of some processing block. Possible processing blocks are:

- **Procedures:** Procedures have a parameter interface, can have local data, and are called via ABAP statements.
  - *Methods* are defined by `METHOD` and `ENDMETHOD` and implement the functionality of classes.
  - *Function modules* are defined by `FUNCTION` and `ENDFUNCTION` and represent reusable coding entities.
  - *Subroutines* are defined by `FORM` and `ENDFORM` and were previously used to modularize ABAP programs.
- **Dialog modules:** Dialog modules defined by `MODULE` and `ENDMODULE` have no parameter interface and cannot have local data. Dialog modules are the functional interface between classic Dynpros and ABAP programs.
- **Event blocks:** Event blocks have no parameter interface and cannot have local data. Event blocks are triggered from the ABAP runtime environment.
  - The *program constructor* defined by `LOAD-OF-PROGRAM` is triggered once when a program is loaded into an internal session.
  - *Selection screen events* defined by `AT SELECTION-SCREEN` are triggered during the PBO and PAI of a selection screen.
  - *List events* defined by `AT LINE-SELECTION`, `AT USER-COMMAND`, etc., are triggered during classic list creation and processing.
  - *Reporting events* defined by `START-OF-SELECTION`, `GET`, etc., are triggered during program execution after `SUBMIT`.

Recommendation

We recommend the following when working with processing blocks:

- **Procedures:** With ABAP Objects (remember our first recommendation), you work mostly with methods. Function modules in function groups may be created where absolutely necessary for technical reasons (RFC, classic screens), but new function modules should be written such that they contain no operational statements. Function modules should serve merely as wrappers for method invocations. (You can, of course, continue to use existing function modules.) In this vein, create no new subroutines and make every attempt to avoid calling subroutines of other programs (with the exception of subroutines of a subroutine pool created via `GENERATE SUBROUTINE POOL` that serve as wrappers for local methods; see the example below for an example of using a subroutine pool). Although subroutines could technically be conceived of as the "public methods" of a program, in general they are intended for internal (i.e., private) use only. Instead of subroutines, for internal modularization you can use methods of local classes, which can be restricted to their intended usage.

  A (generated) subroutine pool should contain maximally one subroutine as an entry point. The following example illustrates such a subroutine. It serves the same purpose as the `START-OF-SELECTION` event block in an executable program:

```
FORM form.
  cl_application=>main( ).
ENDFORM.
```

> ***Note!***
>
> For the remainder of this article, we'll assume that you implement your operational statements in methods only, and therefore obsolete or dangerous language elements forbidden in classes will not be discussed further. (For a complete list of such elements, refer to the ABAP keyword documentation.)

- **Dialog modules:** Dialog modules are only necessary for complex classic Dynpros. For example, you may want to react in some way to a specific user input or to control the data transport from Dynpro to ABAP program via `FIELD` statements (perhaps in combination with a message-based error dialog). When working with Web Dynpro or with controls on classic Dynpros, you normally do not need dialog modules at all — you can program the necessary PBO and PAI before and after `CALL SCREEN`. When you have no other choice than using dialog modules, the same holds as for function modules: do not program operational statements inside dialog modules; invoke methods instead.
- **Event blocks:** Most event blocks have become obsolete, though there are some cases in which they can still be useful.
  - You might use the `LOAD-OF-PROGRAM` in function groups: a function group can be seen as a public class with only public static methods, with the program constructor playing the role of the static constructor.
  - Selection screen events play the same role for selection screens as dialog modules for general Dynpros. Therefore, for the implementation of the respective `AT SELECTION-SCREEN` event blocks, the same holds as for dialog modules.
  - List events are no longer needed (in new coding), since classic lists themselves should no longer be used (refer back to the rules for GUI programming).
  - From the available reporting events, use only `START-OF-SELECTION`. As with dialog modules, the corresponding event block should contain no operational statements, but a call to the main method of the main local class of the executable program. Strictly speaking, you need `START-OF-SELECTION` only if your program must be executable via `SUBMIT` (refer back to the guidelines for program execution and invocation).

    The following example illustrates the allowed use of `START-OF-SELECTION` in an executable program:

```
REPORT application.

CLASS cl_application DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS main.
    ...
ENDCLASS.

CLASS cl_application IMPLEMENTATION.
  METHOD main.
    ...
  ENDMETHOD.
  ...
ENDCLASS.

START-OF-SELECTION.
  cl_application=>main( ).
```

Source code sequence

The ABAP Compiler gives you a great deal of leeway regarding the order in which code can be entered:

- The order of processing blocks is arbitrary.
- It is not necessary to define the `START-OF-SELECTION` event block explicitly.
- With the exception of `AT SELECTION-SCREEN` and `GET`, event blocks (especially `START-OF-SELECTION`) can occur multiple times inside a program. When an event is triggered, the respective event blocks are executed sequentially as single units.


```
REPORT demo.

CLASS demo DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS main.
ENDCLASS.

CLASS demo IMPLEMENTATION.
  METHOD main.
    FIELD-SYMBOLS <fs> TYPE ANY.
    ...
*   ASSIGN dobj     TO <fs>. "Syntax error
    ASSIGN ('DOBJ') TO <fs>. "Dynamic assign
    IF <fs> IS ASSIGNED.
      MESSAGE 'Assigned' TYPE 'I'.
    ENDIF.
    ...
    DATA dobj TYPE i.
    ...
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  demo=>main( ).
```

Figure 1 Static visibility vs. dynamic validity

- The position of declarative statements (for types, data, classes, or interfaces) is not limited to the head of a program or a procedure. You can write declarative statements following processing blocks, and the ABAP Compiler even allows you to declare types or data in event blocks or dialog modules (which cannot contain local data) or in control structures (which do not have a local context). Such in-place declarations declare objects that are statically visible in all downstream statements (those that follow the declaration) of the same context (program or procedure).[25] In spite of the downstream-only *visibility*, those objects are nevertheless dynamically *valid* throughout their entire context (program or procedure). In particular, declarations within event blocks or dialog modules are valid for the entire program; declarations in control structures are valid for the surrounding procedure.

The example program in **Figure 1** demonstrates the confusing difference between static visibility and dynamic validity of a data object `dobj` that is declared in-place (i.e., not at the head of a method). The static use of `dobj` in the first `ASSIGN` statement is not possible because `dobj` is not *visible* at that point. The dynamic use of `dobj` as shown in the second `ASSIGN` statement is possible, because `dobj` is *valid* for the entire method `main` at runtime.

[25] The compiler will not recognize as valid any static references to that variable "before" (i.e., "above") its declaration.

Recommendation

The key areas to be aware of regarding code sequence are:

- Declarations vs. implementations
- Sequence of program parts
- Sequence of declarations
- Sequence of statements in procedures
- Definition of event blocks

Let's look at each in turn.

Declarations vs. implementations

You should always distinguish in your program clearly between a global declaration part at the top of the source code and an implementation part below.

- **All global declarations should be written into the global declaration part.** When programming with ABAP Objects (our fundamental rule), the global declaration part contains all class and interface definitions of a program as well as the global data declarations that are absolutely necessary for technical reasons — for connecting classic Dynpro fields to the program, for example.
- **All implementations should follow the global declarations.** When programming with ABAP Objects, the implementations consist of the implementation parts of the classes defined in the global declaration part. Other kinds of processing blocks appear only due to technical necessity (refer back to the guidelines for processing blocks).

The reason for this rule is that all declarations should be statically visible to all implementations. When you organize your program using INCLUDE programs (more on this in the upcoming guidelines for source code organization) — which is recommended for large programs — the ABAP Workbench supports you with the well-known Top INCLUDE. The Top INCLUDE should always be the first INCLUDE of a program. It should contain all global declarations of a program, and it should contain no implementations (operational statements). For programs that are generated by the ABAP Workbench (class pools, function pools), the rule is fulfilled automatically. When working with such programs, take care not to place your own declarations and implementations in the wrong INCLUDE programs.

Sequence of program parts

In addition to a general division between declarations and implementations, keep your programs readable by arranging the declarations and applications neatly. Decide which arrangement fits your programs best and be faithful to that scheme.

For example, two possible schemes are bottom-up and top-down.

- In a **bottom-up** scheme, you define things before you need them. In such a scheme, helper classes are defined in front of the main application classes, or helper variables are defined in front of the main variables. The advantage of a strictly bottom-up scheme is that you can always refer to preceding declarations and therefore minimize the need to defer the declaration of classes and interfaces with the `DEFERRED` addition. On the other hand, such programs might become less readable, since you might place large amounts of coding at the top of the program that does not express its main relevance. In such cases, it is a good idea to put all helper components into INCLUDE programs (more on this in the upcoming guidelines for source code organization) that do not pollute the source code.
- In a **top-down** scheme, you start with the main components (interfaces, classes, other declarations), followed by helper components. In such a scheme, you start with the "important" things, which might make a program more easily readable. On the other hand, this scheme may create a need for multiple deferred declarations. Again, INCLUDE programs can be used to support this scheme.

It's a question of taste as to whether you prefer the bottom-up or the top-down scheme, but for purposes of consistency, you should stick strictly to one style or the other within any given program (or INCLUDE).

Inside the implementation part of a class, method implementations should follow the sequence of their declarations in the declaration part.

Sequence of declarations

For the sequence of component declarations in classes, or local type/data declarations in procedures, there are three possible schemes:

1. Base the sequence of declarations on the *generality* of the declared components. This scheme is supported by chained statements defined with the colon-comma syntax.
   - Types
   - Constants
   - Static components
   - Instance components
   - Field symbols
2. Base the sequence of declarations on the *type* of the declared components. This scheme is also supported by chained statements.
   - Types
   - Constants
   - Data objects
   - Methods
   - Events
   - Field symbols
3. Base the sequence of declarations on the components' *meaning* in the program. Here, you group the components by their semantic proximity. For example, first all components that are used for handling one kind of object, and then all components for another kind of object.

The first two schemes are based on *technical properties* (such as visibility or type) of the components, while the last scheme depends on the *semantics* of your program. The sequence of declarations should follow a hierarchy where the top levels are governed by the technical schemes and the bottom levels follow the semantic scheme, or vice versa. A common scheme places the type of the declared components at the top level, followed by data objects, methods, and events subdivided into static and instance components, and finally components grouped according to their semantics at the bottom level. Note that in the case of local classes, a pure semantic sequence at the top level is hindered by the syntactical division into visibility sections anyway.

> ***Note!***
>
> We are speaking of the source code sequence of an ABAP program here, where all INCLUDE programs are resolved. We are not speaking about the source code sequence within single INCLUDE programs (more on this in the upcoming guidelines for source code organization).

Sequence of statements in procedures

Inside procedures and processing blocks, we recommend the following rules:

- **Inside procedures**, all local declarations (types, variables, constants, field symbols) should be declared at the beginning of the procedure — i.e., directly after the introducing statement and before the first functional statement. Thus, we are repeating the general scheme of dividing declarations from operational statements.
- **Inside processing blocks** that have no local data (dialog modules and event blocks), declarations should not be used at all.

These rules reflect the fact that in-place data declarations do not open a new data context in ABAP. Placing declarative statements (for example, `DATA`) at arbitrary positions, such as in loops, makes it appear that the data is opening a local context (it appears to be declared within the loop only), which it does *not* — it's being globally defined in the surrounding

context![26] You should follow these rules in order to avoid any confusion between the visibility and the validity of declarations (see the example in Figure 1). When you follow the rules, the scopes of visibility and validity are the same.

Definition of event blocks

Two practices should be employed when coding event blocks within an executable program:

- Even though optional, for consistency and clarity, the event block `START-OF-SELECTION` should be explicitly defined.
- Each event block must be unique inside a program; that is, you should not define any event block more than once.

The following sequence is an example of using a top-down scheme in an ABAP program. The program is divided into the essential two parts — a global declaration part and an implementation part — and the declarations of the declaration part are strictly ordered according to their generality and their importance:

- The first part is the **global declaration part** of the program, where *all* global declarations are listed in the following sequence:
  1. Deferred declarations of *local classes* and *interfaces* that are used in following declarations
  2. Declaration of *data types* global to the program
  3. Declaration of *local interfaces* of the program
  4. Declaration of *local classes* of the program
  5. Declaration of *data objects* global to the program
- The second part is the **implementation part** of the program, where all processing blocks are listed in the following sequence:
  1. Implementation parts of all classes containing *method implementations* in the sequence of their declarations
  2. Other helper processing blocks (*function modules*, *dialog modules*, if necessary)

Such a top-down scheme is common for ABAP programs.

Source code organization

Source code of ABAP programs can be organized using INCLUDE programs or macros.

- An **INCLUDE program** is an ABAP program that is not generated independently, but can be integrated into one or more other ABAP programs using the `INCLUDE` statement.
- A **macro** is a code sequence defined between `DEFINE` and `END-OF-DEFINITION` for reuse in the same program.

Recommendation

INCLUDE programs provide an effective tool for organizing one and only one ABAP program.

- INCLUDE programs should be created to provide for the source code modularization of exactly one ABAP program only — an INCLUDE program should not be included in several programs. Especially, do not use INCLUDE programs for reusing:
  - Type definitions
  - Data declarations
  - Local classes
  - Procedure implementations

  Use global service classes or interfaces instead. The reuse of INCLUDE programs in several programs severely constrains the maintainability of the programs as well as the maintainability of the INCLUDE program itself. Changes in an INCLUDE program that is reused in several programs can make programs syntactically

[26] For a detailed discussion of in-place declarations, see pitfalls #7 and #8 in "Steering Clear of the Top 10 Pitfalls Associated with ABAP Fundamental Operations and Data Types" (*SAP Professional Journal*, July/August 2001).

incorrect. This can especially be a problem if INCLUDE programs are reused in systems other than their original system.

- The naming of an INCLUDE program should follow the naming conventions of the ABAP Workbench. (These are the names proposed when creating INCLUDE programs via forward navigation.) The naming conventions of the ABAP Workbench support the use of an INCLUDE program in exactly one program.
- You should never distribute functional units — i.e., processing blocks — over several INCLUDE programs.
- For large programs, the use of INCLUDE programs is strongly recommended in order to preserve program readability and maintainability. For a single program, INCLUDE programs can be used effectively to organize programs that are arranged following either the bottom-up *or* the top-down scheme (refer back to the guidelines for source code sequence). **Figure 2** is an example of how INCLUDE programs can be used in a program where the declaration part is arranged in bottom-up style. Resolving the INCLUDE programs results in a program where helper classes are declared before they are needed. Nevertheless, the `main` class remains prominently visible in the program.

```
*&---------------------------------------------------------------------*
*& Program         MAIN_PROG
*&---------------------------------------------------------------------*

PROGRAM  MAIN_PROG.

INCLUDE main_prog_top.

INCLUDE main_prog_impl.

*&---------------------------------------------------------------------*
*&  Include        MAIN_PROG_TOP
*&---------------------------------------------------------------------*

INCLUDE main_prog_decl1.

CLASS main DEFINITION.
  ...
ENDCLASS.

*&---------------------------------------------------------------------*
*& Include         MAIN_PROG_IMPL
*&---------------------------------------------------------------------*

CLASS main IMPLEMENTATION.
  ...
ENDCLASS.

INCLUDE main_prog_impl1.

*&---------------------------------------------------------------------*
```

Figure 2 Using INCLUDE programs in a program with a bottom-up declaration *Continues on next page*

Figure 2 *continued*

```
*&  Include       MAIN_PROG_DECL1
*&---------------------------------------------------------------------*

INCLUDE main_prog_decl2.

CLASS helper1 DEFINITION.
  ...
ENDCLASS.

*&---------------------------------------------------------------------*
Include           MAIN_PROG_IMPL1
*&---------------------------------------------------------------------*

CLASS helper1 IMPLEMENTATION.
  ...
ENDCLASS.

INCLUDE main_prog_impl2.

*&---------------------------------------------------------------------*
Include           MAIN_PROG_DECl2
*&---------------------------------------------------------------------*

CLASS helper2 DEFINITION.
  ...
ENDCLASS.

*&---------------------------------------------------------------------*
Include           MAIN_PROG_IMPL2
*&---------------------------------------------------------------------*


CLASS helper2 IMPLEMENTATION.
  ...
ENDCLASS.
```

- The division between declarations and implementations is supported by the Top INCLUDE, which, as a naming convention, has the postfix TOP and can be generated automatically when creating a program in the ABAP Workbench. If you organize your source code with INCLUDEs, you should always organize all statements that belong to the global declaration part within the Top INCLUDE (which might include other INCLUDEs). Never split the global declaration part into INCLUDEs that are not included in the Top INCLUDE. The reason is that the compiler always includes the Top INCLUDE when compiling other INCLUDEs of the program (of course, the same holds for the syntax checker). This allows you to maintain and deliver parts of your program independently from other parts. A well-known example is the activation and transport of methods or function modules independently from their respective classes or function pools. This is possible because the ABAP Workbench organizes each procedure in its own INCLUDE. For the same reason, you should refrain from placing operational coding in the Top INCLUDE. Otherwise, the compiler (or syntax checker) loads this coding unnecessarily when the Top INCLUDE is needed for compiling other implementation parts of the program.

- Granularity of INCLUDEs:
  - For the declaration part, the granularity depends on the number of declarations. If your declaration part comprises only a couple of interfaces/classes with not too many components, you can write all declarations into the Top INCLUDE directly. Otherwise, create individual INCLUDEs for interfaces/classes. At a minimum, an INCLUDE should not comprise less than one interface or class declaration.
  - For the implementation part, the granularity depends on the size of the implementations. We propose that one INCLUDE comprise maximally the implementation part of one class. Depending on size, you can write the method implementations directly into such an INCLUDE or include individual INCLUDEs for the implementations. An INCLUDE should minimally comprise the implementation of one method.

Although macros can also be used to organize programs, their use is discouraged for the following reasons:

- They increase the difficulty of reading code.
- Their implementation in the ABAP language provides only a bare minimum of functionality.
- You cannot debug *into* macro-generated code.

For source code organization, use INCLUDE programs instead of macros. For functional modularization use procedures (methods) instead of macros.

Coding style

As with most other programming languages, ABAP provides for a large variety of *programming styles*. By programming style, we mean formal practices that do not influence the semantics of a program. Some examples are:

- Naming conventions
- Indentation
- Comments

Note!

As we mentioned in the introduction, the tenor of this article is ABAP-specific. We will therefore not delve deeply into general recommendations regarding programming style. This section is therefore rather brief, giving only some basic, ABAP-oriented recommendations.

Recommendation

We are not suggesting far-reaching restrictions applicable to programming style. In particular, we are providing no strict rules for naming conventions, indentation, use of upper or lower case, etc. Instead, our recommendation is that you define a style for yourself or for your development team. Once you have defined such a style (or your team has decided to follow a given style), however, you should make every attempt to adhere to that style (at least as it pertains to a given development project). Our minimal recommendations that can serve as a basis for all style definitions are as follows:

- Use the **Pretty Printer** in the ABAP Editor, and use it frequently. It's good practice to use the Pretty Printer each time one of your own programs is brought into the ABAP Editor and/or each time it's saved. Choose the Pretty Printer settings that suit you best. A common setting (commonly used with the old front-end editor) is to use indentation and to distinguish between keywords and operands using upper and lower case. With the new front-end editor[27] (available as of SAP NetWeaver 2004s), which supports syntax highlighting,

[27] By the term "old front-end editor," we mean the standard editor up to SAP NetWeaver '04 that is implemented using a relatively simple text editor control. As of SAP NetWeaver 2004s, a new front-end editor is available that is specialized for use when editing source code. The new front-end editor supports syntax highlighting and provides many other advanced features. A code completion feature is under development for the next release. Use of the new front-end editor is highly recommended if available.

another common setting is to use indentation but not to use upper/lower case conversion, since the syntax highlighting provides excellent differentiation between statement elements. (Note, however, that designators are still shown in upper case in tools such as the ABAP Debugger or in the short dumps of runtime errors.) Furthermore, when using literals as dynamic tokens (more on this in Part 3 of this article series when we discuss dynamic token specification), in most cases such literals must be entered in upper case. Therefore, a setting where keywords are shown in lower case and operands are shown in upper case may be appropriate for your particular programming context. On the other hand, also note that in ABAP documentation and in related publications, keywords are traditionally shown in upper case. In the end, therefore, it often resolves simply to a question of taste.

- Program code *must* be readable and understandable by everyone who knows ABAP. If programs or parts of programs are not understandable by reading the source code itself and its documentation, you must use **comments**. Comments should be in English and describe what a program, procedure, or part of a program is doing. If necessary, you should also comment on how the code produces its results — e.g., by including citations to literature if you implement general algorithms.

- Aside from the **naming rules** prescribed by the syntax in Unicode-enabled classes (start a name with a letter followed by a sequence of letters, digits, and underscores), we suggest no strict naming conventions for internal names. Nevertheless, we strongly recommend that names be meaningful to English-speaking readers and that you use naming conventions where appropriate to make coding clearer (more on this in Part 2 of this article series when we provide some hints on modularization).

- Since the mixed use of **upper and lower case** characters inside a name is problematic when the Pretty Printer is used with upper/lower case conversion, it is recommended that parts of names are delineated using underscores (e.g., `get_value` instead of `getValue`). However, if you use the new front-end editor with syntax highlighting, you can switch off the upper/lower case conversion of the Pretty Printer and perhaps use mixed-case designators such as `getValue`.

- Since there are different **name spaces** for data *types* and data *objects*, it is theoretically possible to declare data objects with the names of data types. However, we strongly advise against this, in order to eliminate confusion on the part of the reader of your code. If you must declare data objects with the names of data types, however, restrict the use of this feature to cases where your meaning will be absolutely clear. For example, a helper variable for an index might be declared via `DATA i TYPE i`, but never declare data objects with names of data types if they are not of that type — a declaration such as `DATA i TYPE f` should never occur.

- Respect the **style of your colleagues**. If you must change or maintain coding written by other developers, you must respect the given coding style.

- For **external names** (repository objects), a small set of special naming conventions is valid that should not be used for other repository objects, including (among others):

  - `CL_` for global classes
  - `IF_` for global interfaces
  - `CX_` for exception classes
  - `CL_OS_`, `IF_OS_`, `CX_OS_` for interfaces and classes of Object Services
  - `CL_BADI_`, `IF_BADI_`, `CX_BADI_` for interfaces and classes for BAdIs

  In order to prevent hiding of external objects, do not use those prefixes for internal names in programs, either. Hiding can generally be a problem in the following cases:

  - When naming internal program objects, you

always must make sure that you do not hide a repository object that you want to use inside your program.

- In a parallel vein, you should take care not to use a global definition accidentally where you want to use a local one.

- Avoid hiding of built-in functions with functional methods.[28]

A naming convention — such as a prefix `l_` for local data declarations or `lcl_` for local classes, for example — can help here, but is not required.

> ***Note!***
>
> For repository objects created and maintained in the ABAP Workbench — such as INCLUDE programs, for example — always choose the name proposed by the workbench in order to support navigation and activation.

Using modern ABAP features

Since ABAP is an evolving language, features and concepts are added to the language from release to release — such features and concepts sometimes have a functional overlap with already existing language elements. The current syntax checker gives you the liberty to decide which language element to use for a given purpose.

Recommendation

Always use the most appropriate language elements. When new language elements have been introduced that better serve the same purpose as existing language elements, adjust the respective statements. Your goal should be to use identical language facilities to achieve identical results in different statements.

Some common examples of where to avoid outdated language elements include the following:

- Use only the relational operators (`=`, `<>`, `<`, `>`, `<=`, `>=`), which are more readable than their respective character forms (`EQ`, `NE`, `LT`, `GT`, `LE`, `GE`).

- If available, use the addition `NOT` inside predicates of logical expressions[29] (e.g., `dobj IS NOT INITIAL`) instead of the Boolean operator `NOT` in front of the logical expression (e.g., `NOT dobj IS INITIAL`). The reason is that you naturally do the same for comparisons (`dobj1 <> dobj2` instead of `NOT dobj1 = dobj2`).

- Use only the short form `meth( )` for a method call. Use `CALL METHOD meth` for dynamic invocation only. With the short form you avoid the pollution of your source code with syntactical noise. This is especially important in object-oriented programming, where method invocations are more frequently used than, for example, function calls in procedural programming. Furthermore, you then use the same syntax for normal method calls as for functional method calls in operand positions.

- Use the keyword `LENGTH len` in type and data declarations with `TYPES` and `DATA`, etc., instead of `(len)`. The reason is to use the same syntax in `DATA` and `CREATE DATA`.

- Use the equals sign (`=`) instead of `TO` and `FROM` when defining the parameter list in `EXPORT` and `IMPORT`. The reason is to use the same syntax for parameter lists as in all `CALL` statements.

- Although not strictly required, use the addition `SUBSTRING` in `FIND` and `REPLACE` when applicable. Why? The specific use of `SUBSTRING`

[28] Recall that a *functional method* is a method of an ABAP class that returns one return value only (*Returning parameter*) and can be used in the same operand positions as built-in functions. Examples for built-in functions are `ABS`, `STRLEN`, and `LINES`.

[29] In ABAP, a logical expression — i.e., an expression whose result is true or false — can be written either as comparisons involving relational operators (`=`, `<>`, etc.) or with predicates involving special language elements (`BETWEEN`, `IS`, etc.). Other than the language element `NOT` inside the predicates of a logical expression, the Boolean operator `NOT` works with the result of the expression.

will serve to clearly distinguish your search for substrings from a search for strings matching a regular expression (denoted by the alternative addition REGEX).

- Use the addition ACTUAL LENGTH instead of LENGTH in OPEN DATASET. The reason is to clearly distinguish it from the similar addition MAXIMUM LENGTH.

Writing correct ABAP

The static correctness of ABAP programs is guaranteed via static program checking. Such ABAP program checking includes the syntax check and the Extended Program Check.

- The **syntax check** checks whether the ABAP syntax is correct. The syntax check is called in the ABAP Editor via Program → Check → Syntax (and is invoked automatically when you activate a program). The syntax check provides syntax errors and syntax warnings. Only a program without syntax errors can be executed.
- The **Extended Program Check** checks an ABAP program for a number of additional statically recognizable (potential) errors. While programs with errors in the Extended Program Check can be executed, they nevertheless usually result in an exception. The Extended Program Check is called in the ABAP Editor via Program → Check → Extended Program Check or via transaction code SLIN.[30] Extended Program Checking is also provided by the Code Inspector tool (more on this in Part 3 of this article series when we discuss testing programs).

Recommendation

While it is clear that a program cannot contain syntax *errors*, there should be no syntax *warnings* either. Syntax warnings normally point out syntax constructs that are marked as obsolete and will probably become syntax errors in upcoming releases.

As a guideline, make your programs free from all errors, warnings, and messages returned by the Extended Program Check. The Extended Program Check is a powerful (but underutilized) tool in helping you to write better ABAP. If in rare cases you are convinced that you must write code in a way that leads to messages from the Extended Program Check, you can switch off non-critical checks for specific statements by using special pseudo comments — e.g., "#EC — on the statement line in question. As a general rule, statement-specific pseudo comments are preferred to general ones; do not use the statement SET EXTENDED CHECK OFF at all. If you switch off the Extended Program Check by specific pseudo comments, it signals that you have thought about it and that there are valid reasons for continuing to use the statement as written.

If your ABAP development environment utilizes a formal code review process, we recommend that part of that process incorporate a review of the Extended Program Check output for the program undergoing review.[31]

Normally, the use of text *literals* for other than technical content is to be avoided in ABAP, because they are not translated into other languages. Instead, text *symbols* should be used for non-technical texts. If the Extended Program Check warns you about a text that has purely technical meaning, you can circumvent that warning via a pseudo comment:

```
APPEND '<html>' TO html. "#EC NOTEXT
```

Internationalization

ABAP programs written for use in globally acting enterprises must be able to run with the same results on every system in every text environment

[30] For more information about the Extended Program Check, see the article "An Integrated Approach to Troubleshooting Your ABAP Programs: Using Standard SAP 'Check' Tools During Development and Testing" (*SAP Professional Journal*, March/April 2004).

[31] For more on implementing a code review process, see the article "Put Better Programs into Production in Less Time with Code Reviews: What They Are, How to Conduct Them, and Why" (*SAP Professional Journal*, July/August 2003).

worldwide.[32] To achieve this, your programs must be independent from the system code page. At SAP, globalization is part of the set of product standards.

Recommendation

Apart from the existing rules for Unicode-enabled programs, keep the following in mind:

- Use text symbols instead of text literals. Ensure that the field lengths are large enough so that translation into other languages is possible.
- Avoid using character type fields as containers for structured data. For internationalization purposes especially, structure types with correct type information for the components are mandatory in order to avoid problems arising from alignment gaps on list output or when transferring data to remote functions, files, or printers.[33]
- Don't use characters in comments, literals, and identifiers other than those that are available in 7-bit ASCII. In a worst case scenario, a program may become no longer executable when it uses a code page other than the one with which it was created. This restriction can be lifted — at least for comments and literals — in cases where programs are written for Unicode systems only.[34]
- Avoid offset/length access to texts that contain characters other than 7-bit ASCII because some languages utilize *combining characters*. Combining characters are sequences of characters that result in one grapheme. For the same reasons, avoid cutting texts during value assignments.[35] In both cases you might cut a combining character in the middle, resulting in an undefined grapheme. If necessary, you can find a correct cutting position using the static method `CL_SCP_LINEBREAK_UTILITIES=>SPLIT_STRING_AT_POSITION`.
- When working with files on the application server, open new text files for writing explicitly with code page `UTF-8` (available as of Release 6.10) if all readers can process this format. If you do not open a file with the addition `ENCODING UTF-8`, of statement `OPEN DATASET FOR OUTPUT`, the code page used for writing can depend on the text environment, and it is difficult to identify the code page from the file content during reading.

  As of SAP NetWeaver 2004s, you should add the new addition `WITH BYTE-ORDER MARK` when specifying `UTF-8` for a file opened for writing. Then a byte order mark (BOM) is inserted at the start of the file when the file is opened. A BOM is a sequence of 3 bytes that indicates that a file is encoded in UTF-8.

  Always open text files for reading with `SKIPPING BYTE-ORDER MARK` (available as of SAP NetWeaver 2004s). If there is a BOM at the start of the file, it will be ignored and the file pointer is set immediately following. Without the addition, BOM data will be handled as normal file content.
- After setting the text environment explicitly with `SET LOCALE`, don't forget to reset it.

Conclusion

In this first part of our three-part article series, we have laid the foundation for writing up-to-date ABAP programs by formulating some fundamental and formal rules. These rules form the framework of a model that underlies the two subsequent articles, which will appear in upcoming issues of *SAP Professional Journal*.

[32] The *text environment* is part of the runtime environment of an ABAP program and is made up of a language, a locale, and a system code page. All programs in an internal session have a common text environment. As a default, the text environment of an internal session is determined by the logon language and can be set by the statement `SET LOCALE`. The language of the current text environment is contained in the system field `sy-langu`.

[33] For a detailed discussion of anonymous containers, see pitfall #10 in "Steering Clear of the Top 10 Pitfalls Associated with ABAP Fundamental Operations and Data Types" (*SAP Professional Journal*, July/August 2001).

[34] Remember that writing Unicode programs (i.e., Unicode check is active) does not necessarily mean you are writing for Unicode systems only (refer back to the guidelines for program attributes).

[35] Cutting of texts occurs due to the application of conversion rules during assignments when a target field is not long enough. Texts are normally truncated on the right; numerical texts can also be truncated on the left.

Let's conclude the first part by summarizing the most important fundamental and formal rules that are necessary for writing up-to-date ABAP programs, from our point of view:

- Keep your coding structured — always use ABAP Objects. In the next article, we show you some best practices for *how* to use it.
- Separate presentation logic from application logic. The easiest way to do so is using the new Web Dynpro ABAP because it is based on the MVC model. If Web Dynpro ABAP is not yet available in your system, you must use BSPs (for Web-based applications) or classic Dynpro (for SAP GUI applications). Then, you have to take care of the separation of presentation logic from application logic yourself.
- Work only with Unicode programs, whether or not the program will be executed in a Unicode system. The rules for Unicode automatically enhance the quality and maintainability of programs.
- Implement your operational statements in methods only. Many obsolete or dangerous language elements are automatically forbidden in classes, and hence you can avoid the possibility of these creeping into your programs.
- Arrange your programs neatly, especially as such arrangement pertains to the sequence of program parts, the sequence of declarations, and the sequence of statements in procedures.
- Organize the source code of large programs with INCLUDE programs but not with macros. Do not reuse INCLUDE programs in several programs.
- Define a coding style, and stick to it.
- Keep your coding modern — keep an eye on the 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.
- Keep your coding free from syntax warnings from the syntax check, and code away all errors, warnings, and messages returned by the Extended Program Check.

We are convinced that your programs will become robust, understandable, maintainable, and, well — just better, if you follow these guidelines. The two subsequent parts of this article series will identify additional recommendations in increasing levels of detail.

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.

