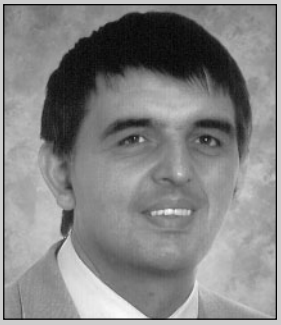
Evaluating the Quality of Your ABAP Programs and Other Repository Objects with the Code Inspector

Randolf Eilenberger and Andreas Simon Schmitt

Randolf Eilenberger, Performance and Benchmark Group, SAP AG



Andreas Simon Schmitt, Business Programming Languages Group, SAP AG

(complete bios appear on page 30)

Software that fails to meet defined standards of quality can have untold consequences for users and developers alike. Suppose you build a wonderful application, but nobody can install it because the documentation is incomplete, and the process is error-prone and poorly tested. Or consider a program that just crawls along on the production system, although it always performed well in the development environment. Understandably, users will flatly refuse to work with the application for well-grounded productivity reasons. Of course, you can then call in the experts to run some performance tests, examine the trace files, and try to identify the bottlenecks. But correcting defects in this late phase — on the production system — is significantly more expensive than doing so early on in the planning, design, or development phase.

To reach and sustain a high level of quality, developers and quality managers *must* know precisely how closely their programs adhere to the relevant standards. Some standards, such as functional correctness, are relevant for every application or product, while others apply only to a subset. Clearly, usability is important only for programs that have a user interface. Accessibility, security, and performance also come to mind, as well as — among many other characteristics — the existence of an adequate user manual, support for upgrades, or the ability to perform data archiving.

A new tool called the “Code Inspector” can help you easily identify some of these types of shortfalls in your ABAP programs. The Code Inspector scans programs for potential problems that are closely related to the static ABAP code, particularly in the areas of performance and security. Best of all, you can now obtain this information while the

Leveraging Dynamic and Static Tests

To perform dynamic tests, you create test cases and test data, execute the tests, and evaluate the results from the tracing tools. The advantage of dynamic tests is that nearly all detected problems will be relevant. But you can only locate a defect if the critical code path containing the defect is part of a test case and will be executed. You can approach, although never reach, complete coverage of all possible code paths by extending the number of test cases — thereby increasing the cost, however. Even if you have abundant test cases, the occurrence of a defect may also depend on a data constellation that you never considered.

You can compensate for some of these drawbacks by using static analysis during the development process. As static tests are often easier to automate than dynamic tests, they can also decrease your testing costs. Think of static checks as expert systems that extend the normal ABAP Syntax Check. The knowledge that you gain from program analysis leads to the development of static checks. For example, consider the following Open SQL statement:

```
SELECT SINGLE * FROM DBTAB WHERE NAME = AUTHOR.
```

If the NAME field is not the first field in a database index (or the second field, when the table is client-dependent and the first field of the index contains the client), this statement will typically lead to a full table scan. For a database table DBTAB that contains many entries, the result will be a long runtime. Therefore, the corresponding expert static check searches for SELECT statements in a program and compares the WHERE clauses with the definitions of the accessed database tables and their indexes. The check raises a message if NAME is not the first (or second) field of a database index of DBTAB.

program is being developed, so you can correct any issues *before* they reach the production system. To introduce you to the Code Inspector, this article will:

- Present the Code Inspector framework.
- Explain how to work with this new tool.
- Review some important checks that are built-in and ready to use.

Options for Determining Program Quality

Aside from design and code reviews, the most efficient technique for controlling program quality is testing. While not always possible, it is desirable to use automated tests to evaluate adherence to standards. You further need to distinguish between dynamic and static tests (see the sidebar above). In a *dynamic* test, you execute the program or scenario to be tested and process some test data. In a *static* test, you evaluate the quality of an object from its static definition. For example, you might try to determine program quality by analyzing the static code.

Good candidates for static checks include naming conventions, layout style guides, and other simple programming standards that are tightly associated with the static code. To some extent, you can also use static checks to look for the use of statements related to system stability and security, such as database hints or expressions that manipulate programs.

In contrast, program performance strongly depends on the dynamically executed code and the processed data. Performance is not normally

Obviously static checks have some shortcomings of their own. In many cases, static checks only refer to the *possibility* of problems that a statement might pose. These problems may never occur because:

- The statement will never be executed. Or, it will only be executed very infrequently or under extraordinary circumstances, so it will never be a problem on a production system.
- The statement will never be executed with critical data. In the example we just discussed, if the database table contains only a few entries, even in the production system, a full table scan would have no impact.

A static check can rarely judge the relevancy of a piece of code. Nor can the check foresee what data a program will process on a production system. Clearly, the success of a static check relies upon a sapient developer reviewing the check results to decide whether a potential problem could evolve into a real one.

We recommend a combination of static and dynamic tests for best results. For example, the SAP product has a defined set of performance standards. To monitor compliance with these standards, we would use a combination of the following tests:

- Static performance checks with the Code Inspector (transaction SCI) for all objects.
- Dynamic checks with Runtime Analysis (transaction SE30) and Performance Trace (transaction ST05) for the most important scenarios.

Since these tools are a fundamental part of SAP Web Application Server (Web AS) technology, be sure to use them in your own testing, too.

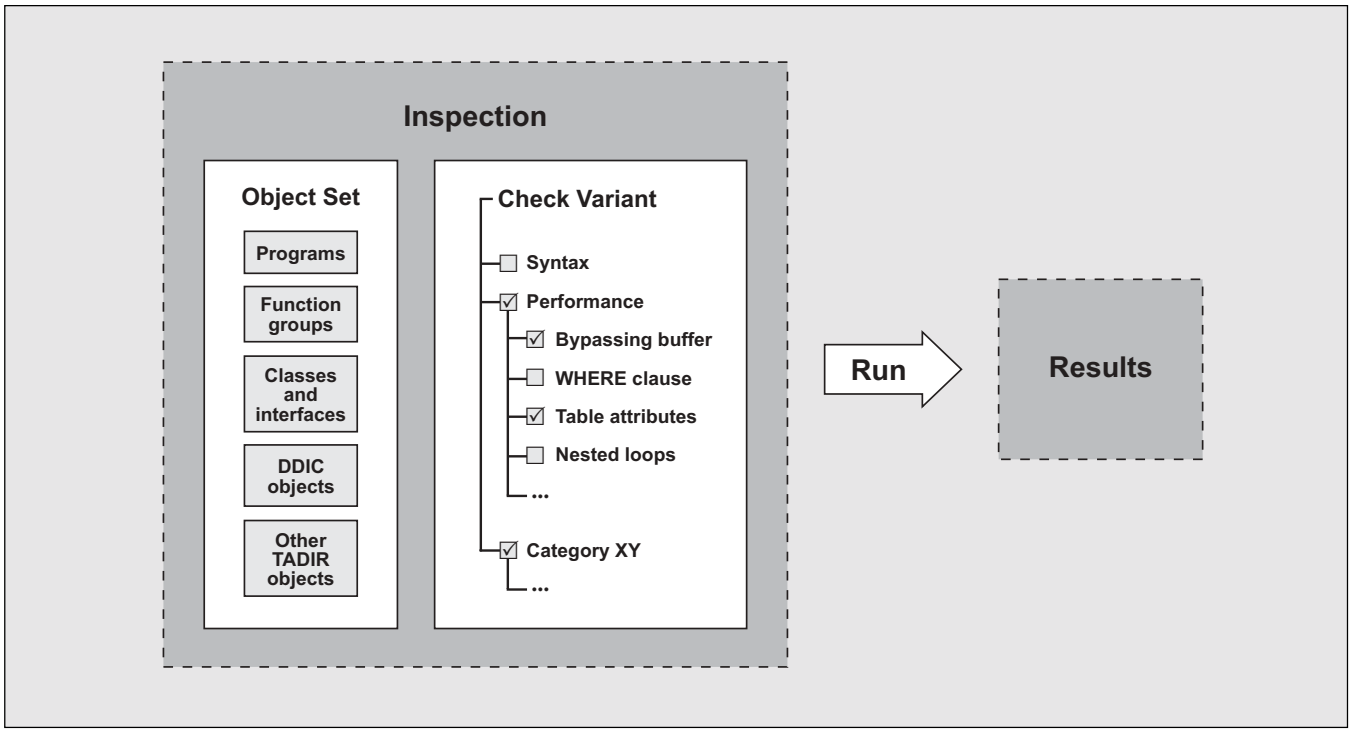
a problem on the development system, where adequate test data rarely exists, but it can quickly become one in the production system. Fortunately, you can make some assumptions about program performance based on static analysis. For example, by analyzing both the source code and the data dictionary definition, you can determine whether an Open SQL statement might become a performance bottleneck because of a badly coded `WHERE` clause or missing database index.

To perform static quality checks, you really need a tool for scanning a lot of source code and other object definitions efficiently. Starting with SAP Web Application Server (Web AS) Release 6.10, you now have such a tool with the new SAP Code Inspector. It checks programs, function groups, classes, and other repository objects. As an ABAP developer, you can easily select all the objects for which you are responsible. As a quality manager, you might want to select all objects for a number of packages, combine this set of objects with a compilation of individual checks, and then run a test that executes the combination. The Code Inspector architecture supports efficient data sharing of individual checks, as well as the option to run the test in parallel in up to 12 tasks. Consequently, you receive test results within minutes, even for very large object sets.

The Code Inspector Framework

Let's start by examining how the Code Inspector supports the static analysis of ABAP programs and other development objects. With this powerful new tool — or perhaps better to say "framework," since you can easily extend it with new individual check modules — you can:

Figure 1 Elements of the Code Inspector



- Check single objects and object sets.
- Combine individual checks easily into a set (referred to as a *check variant*).
- Save and reuse object sets, as well as check variants (which can also be transported through the system landscape).
- Set parameters to control check behavior.
- Achieve fast check execution as a result of check data sharing and optional parallel processing.
- Access online documentation for checks and result messages.
- View all check results consistently in a hierarchical tree format.
- Navigate directly to the object that raises a message.

Technically, the Code Inspector framework consists of two functional areas:

- **The test driver:** The test driver defines and stores the test tasks (referred to as *inspections*), executes them, and stores and displays the results.
- **An extensible set of checks:** The Code Inspector comes with many built-in checks (and more will be added in future releases), but you can also define your own. These additional checks can share the prepared data of the objects to be examined. In other words, the Code Inspector only performs data preparation tasks (such as dissecting a program into its individual ABAP tokens) once per object.

To use the Code Inspector, you work with elements that define and control its operation. **Figure 1** illustrates these elements and the relationships between them. An *inspection* (the test you run) consists of the combination of an *object set* (the objects to scan) and a *check variant* (the set of checks to perform). Executing an inspection then produces the *results* as output. Next we'll examine each element in a little more detail.

Check Variant

The Code Inspector comes with a variety of individual checks that are organized into functional categories such as syntax, performance, and security. Some checks also have parameters that allow you to further control the concrete scope of the check and its behavior. Therefore, the complete specification of a check is the reference to the individual check, extended by its parameters. A check variant is a compilation of one or more individual checks. In order to apply a check to an object, you must first add the check to a check variant. You can name and save check variants for later reuse, as well as transport them through the system landscape.

Object Set

Checks operate on objects. An object can be any development object that exists in an SAP system and has an entry in the catalog of repository objects (table TADIR). In other words, you can choose programs (reports, function groups, classes, interfaces), screens, user interfaces, database definitions, global types, and so on. An object set is the specification of the list of objects you want to scan. As of Web AS 6.20, the Code Inspector offers four different ways to create object sets. You can select objects from:

- The catalog of repository objects (table TADIR)
- A transport request
- The results of an intersection or union of two other object sets
- The results of a check execution (that is, an inspection)

You can also save and reuse object sets, which are identified by a name and a version number.

Inspection

An inspection is the specification of the test task, which consists of two components:

- The checks to be performed, according to the specified check variant
- The object set on which to execute the checks

An inspection can be either *anonymous* or *named*. Anonymous inspections are temporary and not persisted; named inspections are saved with a name and a version number. A named inspection can be run serially by one task, or in parallel by several tasks for faster execution.[1] After completion of an inspection, a named inspection contains an additional component — the results. Named inspections are designed to handle large object sets (say, more than 20 objects) and are necessary if you need the results to be persisted. Use anonymous inspections for performing small, ad-hoc queries on single objects or small object sets, and when persistence is not required.

Working with the Code Inspector

The Code Inspector is an automated tool that allows you to check the static definitions of repository objects based on criteria such as performance or security. As a result of the check, you see a hierarchical list with messages that refer to objects or statements in objects that appear to be problematic.

These messages are classified as “Information,” “Warning,” or “Error.” Especially for performance checks, many of these messages only indicate *potential* problems. You must apply your knowledge and common sense to decide whether the Code Inspector hit the mark or was simply being hypersensitive. On the other hand, the lack of any performance messages does not guarantee that a program will be free of performance problems.

We’ll examine how to interpret the results later. First let’s look at how you use the Code Inspector.

[1] For more on parallel processing, see the article “Speed Up High-Throughput Business Transactions with Parallel Processing — No Programming Required!” in the January/February 2002 issue of this publication.

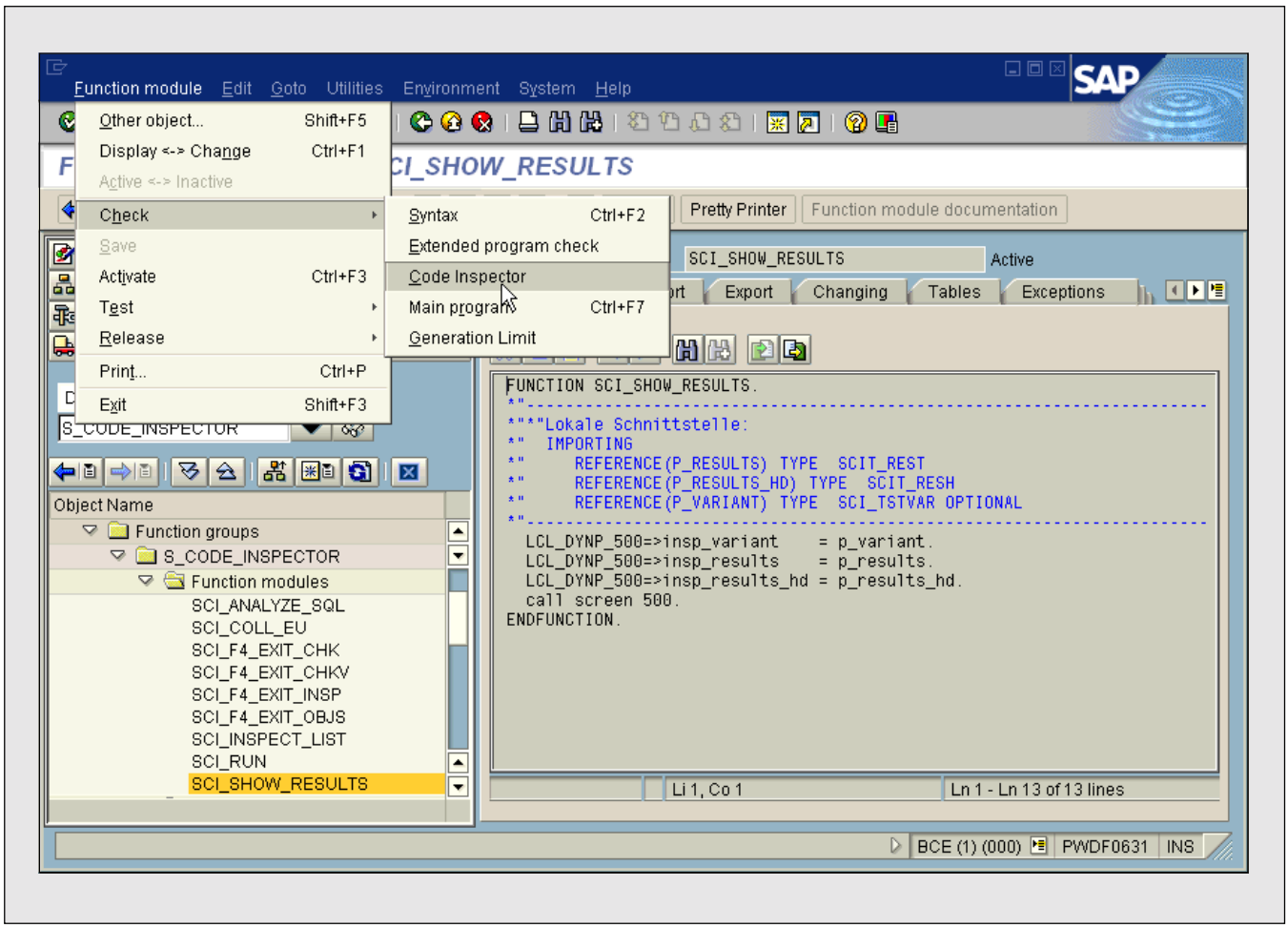
Calling the Code Inspector for a Single Object

When you are working on a program, function module, or class, you typically call the ABAP Syntax Check directly from the editor. But wouldn't it also be nice to have the performance or security checks of the Code Inspector at your disposal? As of Web AS 6.10, you can call the Code Inspector for the object you are working on from the menu of the ABAP Workbench (SE80), the ABAP Editor (SE38), the Function Builder (SE37), or the Class Builder (SE24). Simply select <*Object*> → *Check* → *Code Inspector* (see **Figure 2**), where <*Object*> represents "Program," "Function module," or "Class."

> ✓ ***Function Module Checks***
>
> *For a function module, a check examines the entire function group to which a module belongs, since function modules are not self-contained objects from the perspective of a syntax check.*

When called from these workbench transactions, the Code Inspector automatically applies the DEFAULT check variant (see the sidebar on the next page) to the examined objects. The inspection results are not persisted.

Figure 2 Calling the Code Inspector from the ABAP Workbench Menu



Check Variants That Come with the Code Inspector

The Code Inspector package S_CODE_INSPECTOR comes with a variety of individual checks that you can combine into check variants. You can transport global check variants (see the sidebar below) through the system landscape like any other transportable SAP object. The Code Inspector comes with a few global check variants, the two most important of which are:

- **DEFAULT:** Every inspection started from the ABAP Workbench uses the DEFAULT check variant, which is extensive. This check variant contains the normal and extended ABAP Syntax Checks, as well as additional performance checks and security checks. (We'll examine these checks in more detail in the section "Standard Checks That Come with the Code Inspector.") You can also check workbench objects with a different combination of checks by defining your own local DEFAULT check variant, which will then override the global one. You create a local DEFAULT check variant either with transaction SCI or from the result screen of a workbench inspection.
- **PERFORMANCE_CHECKLIST:** This check variant includes a basic set of checks for performance analysis, such as "Analysis of the WHERE clause for SELECT, UPDATE, and DELETE." It comprises the most important performance checks to help you identify badly coded WHERE clauses or statements that bypass the SAP table buffers already in the implementation phase. For a detailed description of the performance checks, see the section "Standard Checks That Come with the Code Inspector."

Defining Global and Local Elements

Every instance of a Code Inspector element (inspection, object set, check variant) can be either *global* () or *local* (). Global elements are visible to, and usable by, every user in the system. To omit redundant definitions, define elements as global when they are frequently or widely used — for example, check variants that all developers in a department or the entire system should use (such as the DEFAULT check variant for inspections started from the workbench), or large and important object sets. On the other hand, you want to minimize the number of entries that individual users see in the F4 help. Declare elements as local when they are of interest only to a specific user. For example, users should not create global object sets that contain only the objects for which they are responsible.

> ✓ ***Note!***
>
> *For a global inspection, the Code Inspector only accepts the combination of a global object set and a global check variant.*

Calling the Code Inspector for Several Objects

To check objects other than programs, function groups, or classes, or to check many objects at the same time, you use the Code Inspector to create arbitrary object sets, add individual checks to check variants, and combine an object set and a check variant to an inspection. Each instance of these elements can be either global or local (see the sidebar above).

Creating and Tailoring Object Sets

In order to check more than one object, you define an object set.

For example, suppose you want to check all ABAP object classes for which a user named "MORIARTY" is responsible. You would follow these steps:

1. Call the Code Inspector (transaction SCI).

2. On the initial screen (see **Figure 3**), enter a name for the object set ("SUSPECT" in the example) and click the "Create" button (), which takes you to the screen shown in **Figure 4**.

3. In the "Object Assignment" area of the "Select Object Set" tab, enter the SAP user name "MORIARTY" in the "Person responsible" field — not to be confused with the "Person Responsible" field at the top of the screen, which is the name of the person responsible for the object set ("HOLMES" in the example here).

4. In the "Object Selection" area in the lower part of Figure 4, enter an asterisk ("*") in the "Class" field to limit the object set to classes only. Without the asterisk, the object set would also include other types of objects (function groups, programs,

Figure 3 Creating the Object Set

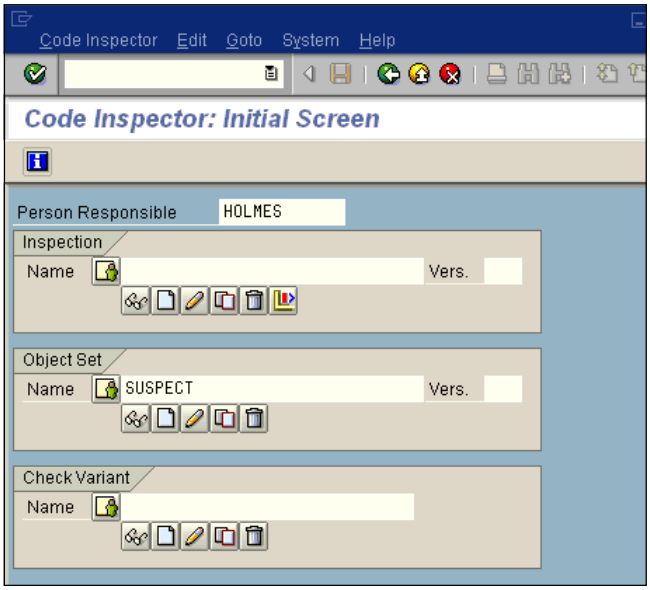
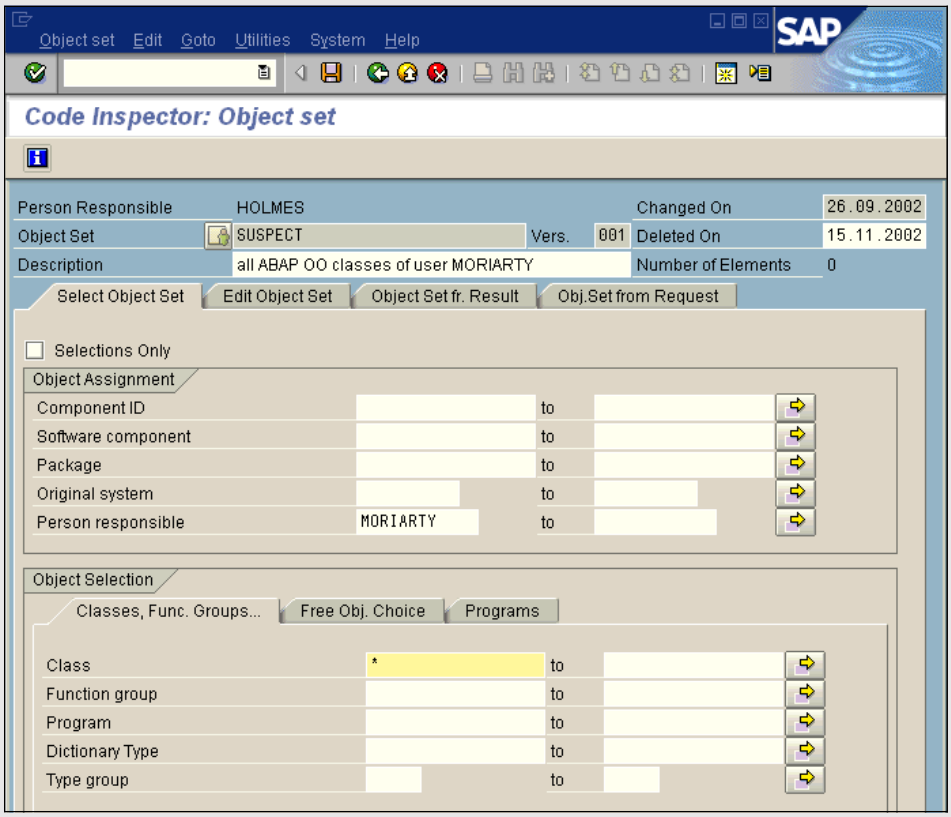


Figure 4 Selecting Objects to Include in the Object Set



etc.). The asterisk also ensures that *all* classes assigned to the user "MORIARTY" will be included. If you enter "CL_A*" instead, only classes that start with "CL_A" are selected.

5. Save the object set by clicking the "Save" button () in the standard toolbar.

6. To review the object list, click "Display Objects" (). If you are not satisfied, change your selections and save the object set again.

Once you have created and saved an object set, you can then tailor your object selections further. In the "Object Assignment" area (see step 3 on the previous page), you can also select objects that are assigned to a particular:

- SAP application component ID
- Software component
- Package
- Original system

Use these options to include or exclude single packages or components. Note that these options are linked by AND operations. If you choose entries that define a set without an intersection, the resulting object set will be empty.

In the "Object Selection" area (refer back to step 4 on the previous page), using the tabs you can select many different types of single objects or ranges of objects. **Figure 5** describes when to use each tab, and which contains the appropriate options for selecting an object type.

Figure 5 Using the Tabs in the "Object Selection" Area

Tab	What You Can Select	When to Use It
"Classes, Func. Groups..."	Specific classes, interfaces, function groups, programs or reports, dictionary types (data element, structure/table, view, physical pool/cluster, table type), and type pools.	To include these predefined object types in your object set.
"Free Obj. Choice"	Any TADIR object type (such as TABL for tables).	To include an object type that is not defined on the previous tab.*
"Programs"	A program based on characteristics (such as program type or status) taken from the table of report sources REPOSRC.	Only if you are familiar with the SAP concept of programs and includes. If you simply want to add a program or report to your object set, use the tab "Classes, Func. Groups...". Since mistakes (such as not specifying a program name) on this tab can lead to very long response times, you must check the "Activate Selections" box on this tab in order to activate the selection.

* If you select an object type on this tab, make sure that the Code Inspector has a check to handle the object type. Otherwise, the object will be part of an object set, but no check will occur. As of Web AS 6.20, the checks delivered by SAP can examine program-like objects (classes, programs, and function groups) and database table definitions. Of course, you can also add a new check to examine objects of another type, and then select the objects on this tab.

Other Methods of Creating an Object Set

Creating an object set from scratch is typically the first method you learn. Once you become familiar with using the Code Inspector, other methods of creating object sets offer added power and flexibility. You access these additional methods via the following tabs on the object set screen (refer back to the screen shown in Figure 4):

- **Edit Object Set:** You can unite or intersect object sets and filter them based on the attributes "Object Type," "Object Name," "Package," and "Responsible." Use this option when you want to create an object set that consists of some combination of two other object sets. Note that this option does not work with virtual object sets.
- **Object Set from Result:** You can create a new object set containing objects that triggered messages in an inspection. Again, you can filter the set based on the attributes "Object Type," "Object Name," "Package," and "Responsible," and also by a single check, its message codes, and message priority ("Error," "Warning," or "Information"). Use this option if you want to determine your result set in several steps — in other words, use any objects that raise issues in a first inspection as the object set for a subsequent inspection.
- **Object Set from Request:** You can also extract object sets from transport requests/tasks and filter them based on the attributes "Object Type," "Object Name," "Package," and "Responsible." If there are program-like objects in the request, the new object set will contain only their associated main programs (programs, classes, or function groups instead of includes, methods, or function modules). Use this option when you want to check the objects in a transport request. For example, you might want to ensure that only objects adhering to your security rules are transported into your production system.

Finally, if you check the "Selections Only" box at the top of the "Select Object Set" tab (see Figure 4), your selection criteria will be saved, but no object list will be generated (referred to as a *virtual object set*). When you use a virtual object set in an inspection run, the Code Inspector will generate an up-to-date object list at runtime to ensure that the inspection does not examine an outdated object set. Use virtual object sets when objects are commonly created or deleted in the selected range of your object set (such as a package). In those cases, using a fixed object list would apply an outdated state of the object set.

✓ *Deletion Date*

A daily background job prevents the Code Inspector from accumulating too much data. Inspections and object sets have an automatically generated deletion date of 50 days ahead, which means that the inspection or object set will be deleted after 50 days. Inspections that have not yet been executed, as well as object sets included in those inspections, are excluded. To keep your object sets or inspections for a longer time, simply change the deletion date. You can change the deletion date individually for individual inspections and object sets.

Creating a Check Variant

After creating an object set (see the sidebar to the left for additional methods), the next step is to decide which checks you want to execute on those objects. The Code Inspector always uses check variants, which are compilations of individual checks, instead of single checks only. (We'll examine the standard checks that come with the Code Inspector in more detail later in this article.) The standard check variants that come with the Code Inspector may be sufficient at first, but sooner or later you will want to create your own check variants for more flexibility.

Creating a check variant is a simple process. You select checks from a tree that contains all active

check categories with their respective active individual checks.

✓ *Check Administration*

When you create a new check variant, the Code Inspector displays the current check variant tree. To add or remove a check (for example, suppose you implemented a new check), select GoTo → Test Administration. You will see a list where you can activate or deactivate individual checks and check categories by checking or unchecking the box in front of the check class.

Follow these steps to create a new check variant:

1. In transaction SCI, enter a suitable name for the check variant and click the "Create" button ( ).

2. To add a check to your check variant, select the checkbox next to the check (as shown in **Figure 6**).

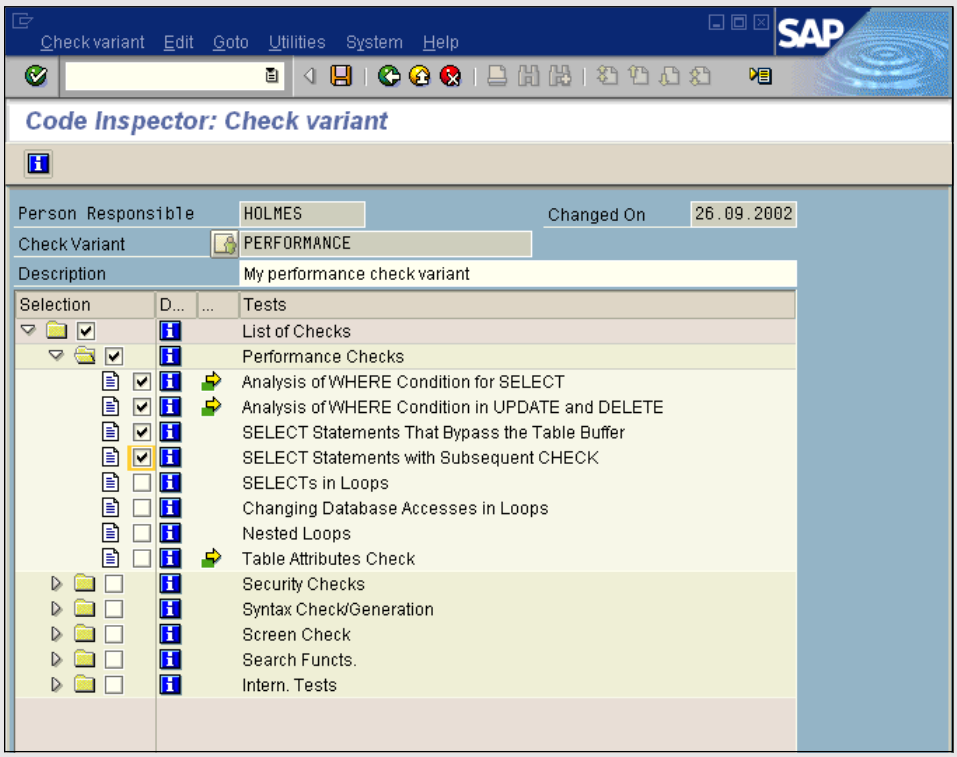
3. To see information about a check, click the "Information" button ( ) next to the checkbox.

4. Do the following for checks with user parameters:

   - An arrow with a box below it ( ) next to the check name indicates that you can set one or more parameters to further define the check attributes. These attributes can be input parameters or can activate a sub-check (a component of a single check that can generate a message). Click the arrow to display a pop-up where you can set the parameters.

   - After setting the parameters, click "Execute"

Figure 6 *Adding Checks to a Check Variant*



() on the resulting pop-up. Some checks will only be executed if the parameters are set appropriately.

For example, the search functions can only be performed if you specified a search word, and that search word must have at least three characters. If a check can be executed with the current parameter settings, the box below the arrow is green. If not, the box is grey. If a check is selected but its parameter settings are not valid, the Code Inspector refuses to save the check variant.

5. To save the check variant, click the "Save" button ().

Creating and Executing an Inspection

You now understand how to create an object set and a check variant. Next, you want to apply the individual checks of the check variant to the objects in the object set. This process happens during the inspection run. So you will need to combine a single object or an object set with a check variant to define the test task that performs the inspection.

Remember that inspections can be anonymous or named. Anonymous inspections are useful for ad-hoc queries with only a few objects where you do not need the results to be persisted (see the sidebar below). You use named inspections when you are dealing with large object sets and/or want to persist the results.

A named inspection is stored persistently with its results. It can handle object sets containing any number of objects and runs either on the local server or in parallel tasks on the servers of a server group. The server group automatically handles load balancing between the different servers, making parallel processing more efficient.[2] Since executing an inspection with many objects puts a heavy load on the SAP system, you can also run a named inspection outside working hours by scheduling it in a background job.

Follow these steps to create and run a named inspection:

1. In the Code Inspector (transaction SCI), enter a

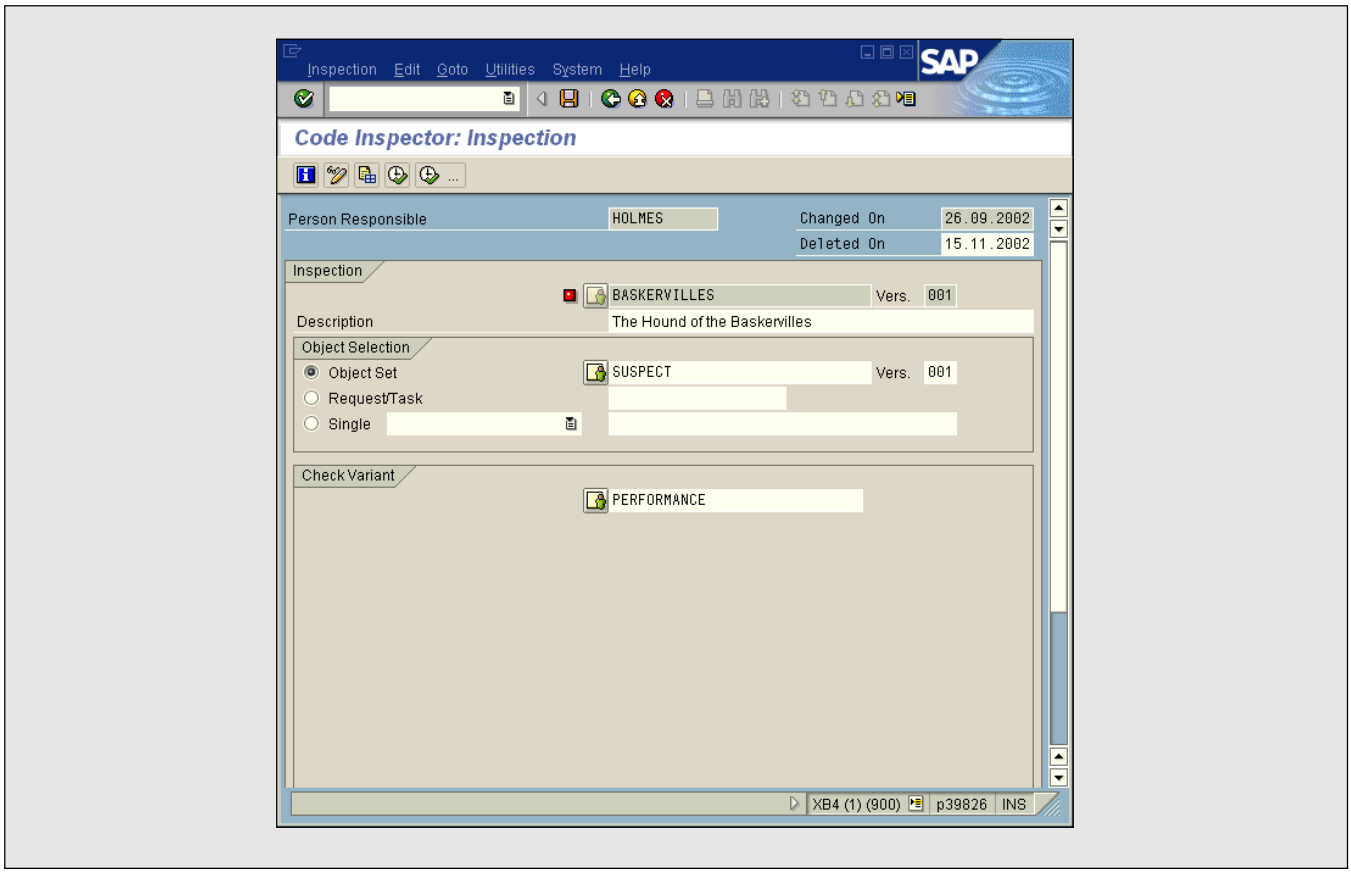
[2] To create a server group, choose *RFC → RFC Groups* from the menu of transaction SM59.

Anonymous Inspections

Anonymous inspections and their results are not stored persistently. The object sets to be checked are restricted to a maximum of 50 objects; larger sets generate an error message. Anonymous inspections are thought to be used for ad-hoc analysis. If you need to check only a few objects and do not need persistent results (such as when you want to test whether a single check works as you expect — it's always a good idea to first check a single object only), press the "Create" button () without entering a name for the inspection. You can then combine a small object set or a single object with an existing check variant ("Predefined") or one picked spontaneously from the tree of checks ("Temporary Definition"). When you press the "Execute" button (), the inspection will be executed immediately on the local server.

After running an anonymous inspection, press the "Results" button () to see the results. Pressing the "Repeat" button () enables you to change the input parameters for and restart an anonymous inspection. Remember that you will lose the results of the previous inspection when you press the "Repeat" button or leave the "Inspection" screen.

Figure 7 Creating a Named Inspection



name for the inspection and click the “Create” button ( ).

2. Combine an object set, a single object, or objects from a request with an existing check variant (see **Figure 7**).

3. You can run the named inspection in one of two ways:

   - Choose the “Execute” ( ) button to start the inspection immediately on the local server in a single task.

   - Click the ... button to display the “Execution Options” pop-up screen (see **Figure 8**). Here you can further control how and when the inspection is run.

     This screen is where you can select a server

Figure 8 Setting the Execution Options for an Inspection

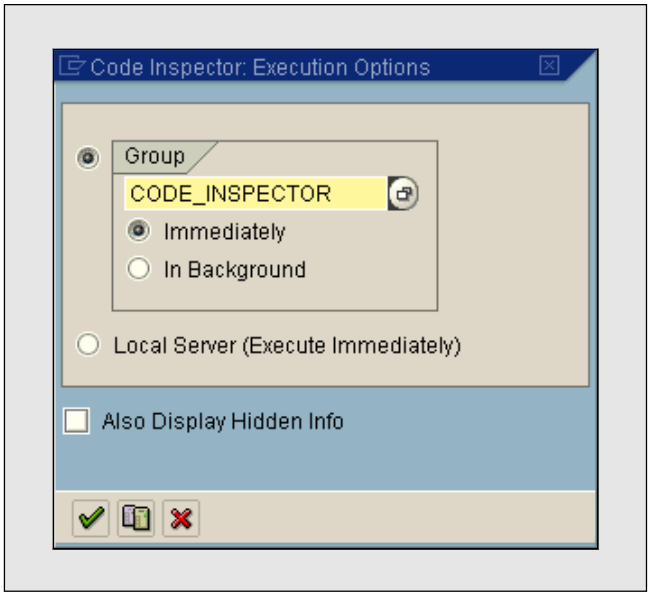
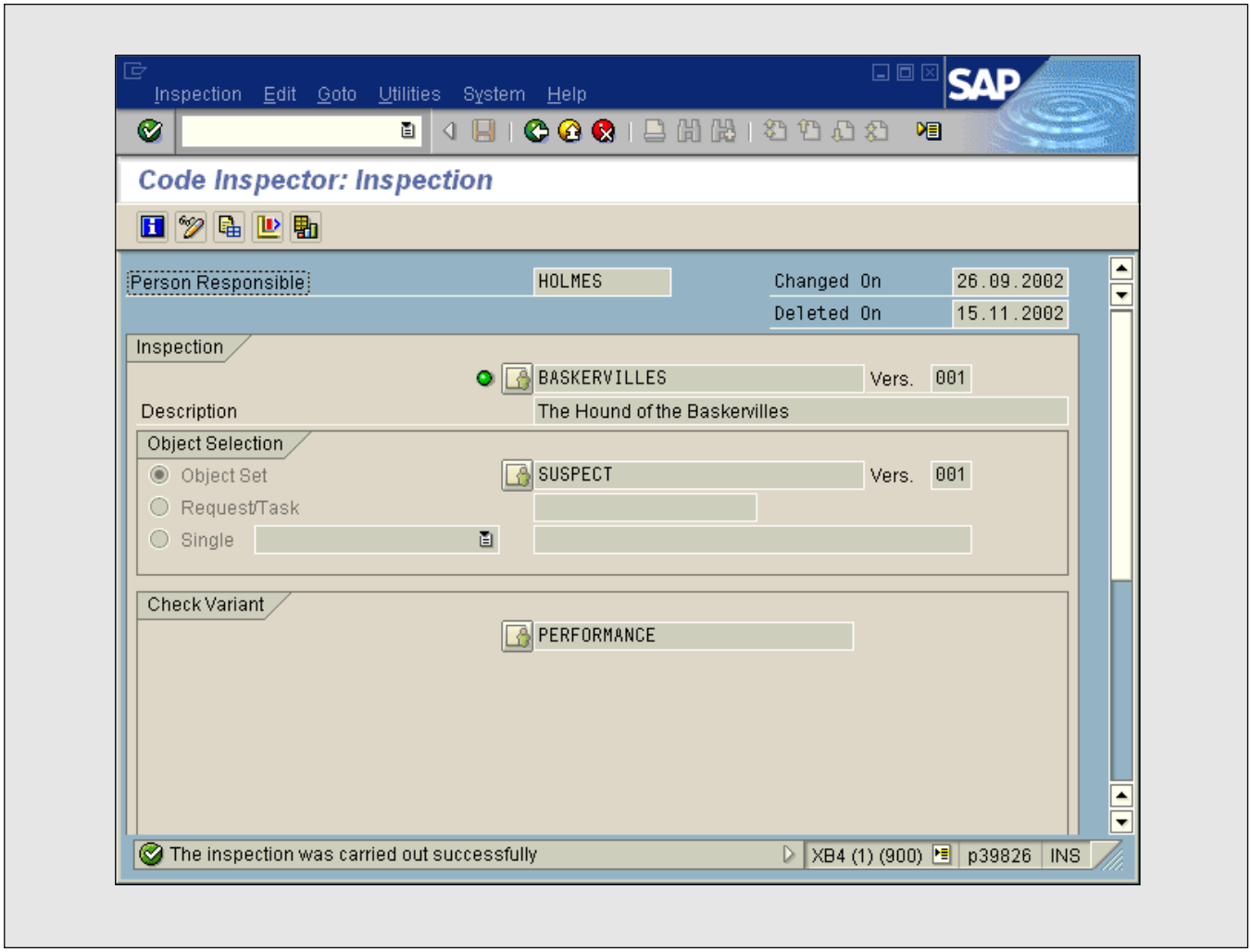


Figure 9 The Inspection Screen After Completion of an Inspection Run



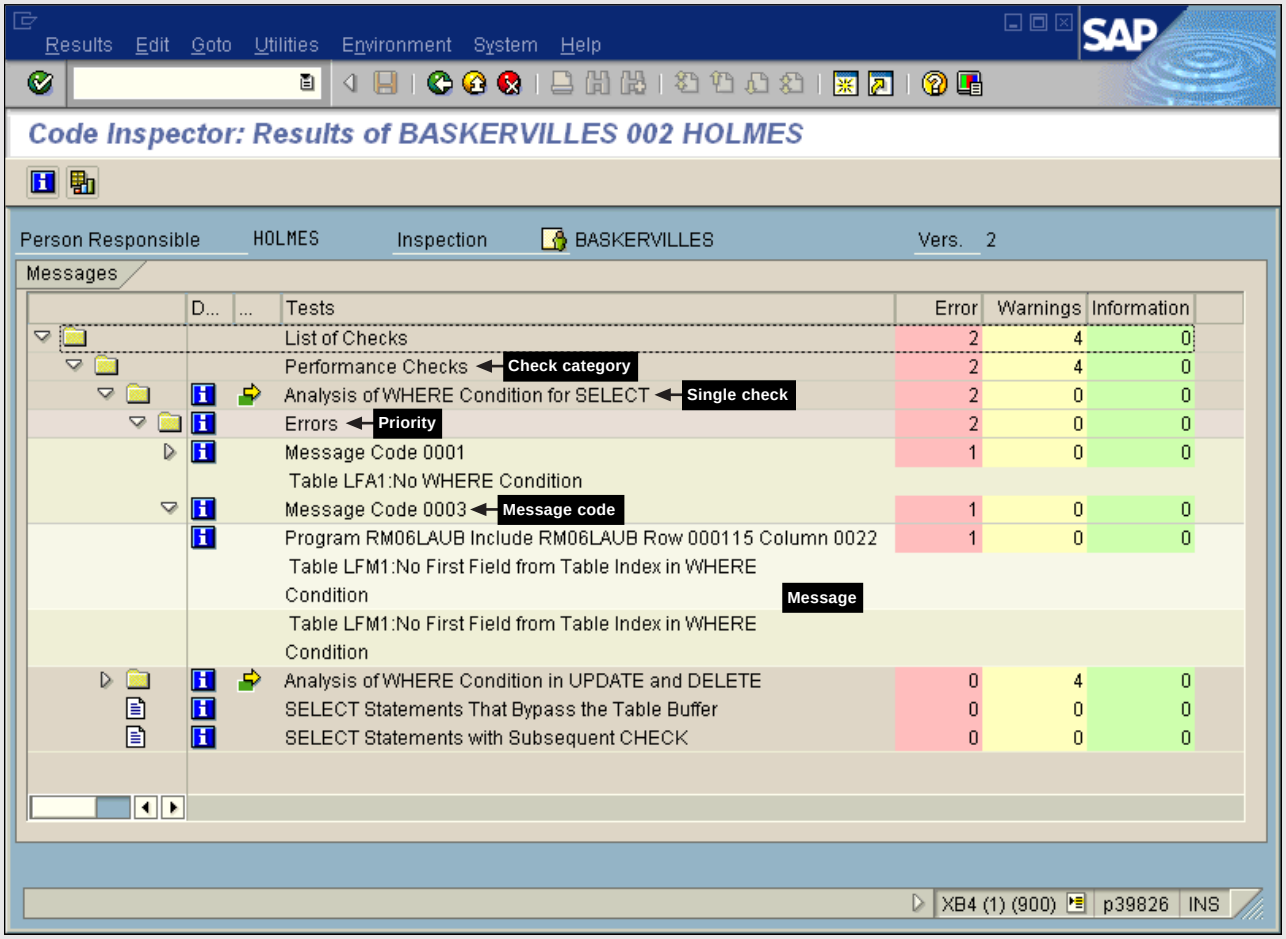
group on which to run the inspection. Make sure that the group contains only active servers that have sufficient resources. The "Maintain Server Group" button () takes you directly into the server group maintenance screen (SM59), where you can create a server group and define the resource parameters for the servers in the group. To schedule the inspection in a background job, click the "In Background" radio button.

Press "Continue" () to start the inspection. If you selected the "In Background" option, you first see a screen where you can schedule a job for your inspection.

✓ *For Large Object Sets*

Run the inspection on a server group when the object set is large (for example, more than 50 objects), so that it is performed in parallel in up to 12 tasks on all servers of the server group (if they have free resources). The load will automatically be split into packages of 10 to 50 objects.

Figure 10 *Results of an Inspection Run*



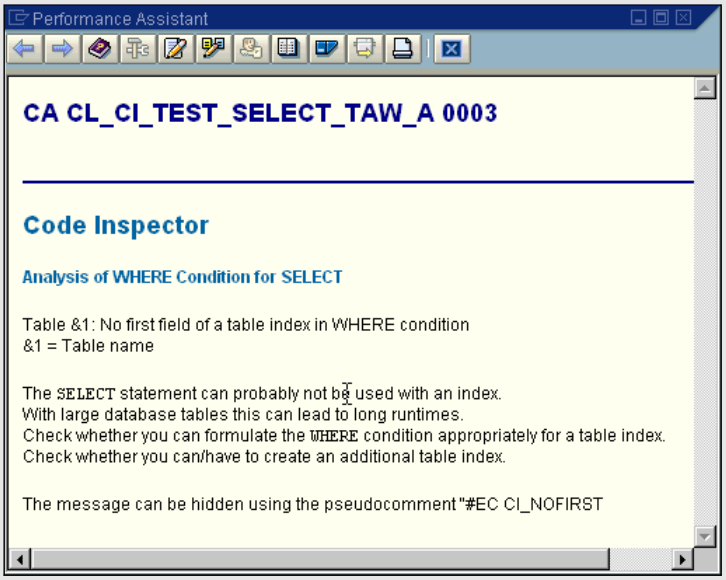
When an inspection run completes successfully, the inspection screen displays a status message across the bottom of the screen, as shown in **Figure 9**. Press the "Results" button ( ) on the "Inspection" screen to display the results.

The Code Inspector displays the results of the inspection in an easy-to-read hierarchical tree format. **Figure 10** shows an example display of the results of an inspection run.

✓ *Priority Color Codes*

- *An entry in the red column in Figure 10 (the left column) indicates an error message.*
- *An entry in the yellow column (the middle column) indicates a warning message.*
- *An entry in the green column (the right column) indicates an information message.*

Figure 11 Information Provided for a Check Message



Each message contains the name of the examined object and a short explanation of why the message was raised. If the examined object is program-like, you also see the source code position of the statement that caused the message. Double-clicking the message text takes you to the ABAP Workbench, where you can display and edit the object. For further explanation, press the "Information" button (🛈) next to the message. You will then see an explanation of the problem (see **Figure 11**) and further links within

Suppressing Inspection Messages with Pseudo-Comments

Many Code Inspector checks provide advice or notes rather than error messages. Even something that the result tree indicates as an "Error" can nevertheless be correct under specific circumstances. For this reason, you may want to explicitly skip these messages by inserting pseudo-comments directly into the code that caused a message during the inspection.

For example, a performance check of the Code Inspector might indicate that "a database table is accessed without specifying a WHERE clause." Press the "Information" button (🛈) in front of the check message to display the online documentation for the check (like the documentation shown in Figure 11). It will tell you that the statement will lead to a full table scan when executed (or, if the table is client-dependent, to a scan of all entries of the current client).

If you really need to read all the table entries during inspection (since there is no better way to access the information), you can insert the pseudo-comment "#EC CI_NOWHERE, as the online documentation will tell you, into the statement (or the line following the statement) in your ABAP program. The Code Inspector will then suppress the message in further inspection runs. You will find the appropriate pseudo-comment at the end of the online documentation that is provided for every check.

the Code Inspector online documentation. For some checks, the Code Inspector also suggests a pseudo-comment that you can insert in the code to suppress that message (see the sidebar below).

> ✓ ***Inspection Statistics***
>
> *Every named inspection generates statistical data about the inspection run. This information is especially useful in situations where a run has been interrupted due to a server shutdown or an error during execution. After the run has completed or aborted, click the "Statistics" button () on the "Inspection" screen. You can see the start and end time/date of the run, cumulated runtime, names of involved servers, number of tasks and objects, and the execution state of the objects. Any error conditions are also noted.*

The Top-Down Approach to the Code Inspector

You have now learned how to create a new inspection from the bottom up — first you create an object set, then a check variant, and finally the inspection. For novice users, we recommend this approach in order to become familiar with the Code Inspector. Later, you might find it quicker to use the top-down approach instead:

1. Create a new inspection.

2. Choose an existing object set using the F4 help. Alternatively, enter the name of a new object set in the "Object Set" field. Double-click the field and answer the "The Element Does Not Exist. Create it?" pop-up with "Yes." Create and save the new object set, and return to the "Inspection" screen.

3. Choose an existing check variant using the F4 help. Alternatively, enter the name of a new check variant in the "Check Variant" field. Double-click the field and answer the "The Element Does Not Exist. Create it?" pop-up with "Yes." Create and save the new check variant, and return to the "Inspection" screen.

4. Run the inspection.

We chose to implement this method to define an exception for the Code Inspector, instead of one based on entries in exception tables. We prefer pseudo-comments for the following reasons:

- Developers must touch the code in order to omit a message. Hopefully they will first think carefully about doing it right, and only then consider inserting a pseudo-comment.

- With pseudo-comments, the object that raises the message and the proper exception are kept together.

- Pseudo-comments serve as a form of documentation. Developers must make it clear that they recognize the message and the possible problem, but that they decided to suppress the message anyway.

You can also choose to run an inspection that ignores all pseudo-comments. This method can be useful in conjunction with a search for pseudo-comments (which can also be accomplished with the Code Inspector) to review whether pseudo-comments are being used appropriately.

Standard Checks That Come with the Code Inspector

The Code Inspector comes with a set of built-in checks that are organized into functional categories, such as:

- Syntax checks and program generation
- Security checks
- Performance checks
- Search operations

To help you get the most out of the Code Inspector right away, we want to introduce you to the most important checks in these categories. Since SAP is continually extending the Code Inspector framework with new check modules, you may find additional checks in your system when you read this article. When creating a new check variant, remember that you can display the online documentation for an individual check to learn more about it.

Syntax Checks and Program Generation

This category contains checks that are associated with the ABAP Syntax Check. All checks are standard tests that are also available individually from the ABAP Workbench. The advantage of applying the checks in the Code Inspector is that you can combine them with other checks and perform them on a set of programs at the same time. The checks in this category are:

- Normal ABAP Syntax Check
- Extended Program Check
- Program Generation

You can add one, two, or all three of these checks to a check variant. Since the ABAP Syntax Check is also a part of Program Generation, you do not need to combine them.

Normal ABAP Syntax Check

This check performs the ABAP Syntax Check, which is also available from most workbench transactions. You control its behavior by setting the following parameters:

- **One or Several Errors:** Normally the syntax check terminates after the first error is detected, but you can also choose to proceed. If a fatal error is detected or the maximum number of 50 errors is exceeded, the check is terminated.
- **With or Without Warnings:** In addition to error messages, the syntax check also returns warnings for minor syntax problems, such as missing implementations of interface methods. You can turn these warnings on or off.

Extended Program Check

This check uses the same set of checks that is available with the workbench transaction SLIN for single objects. It processes static checks that are too complicated or too time-consuming for the normal syntax check. The first operation of this check is a normal syntax check. If any error is detected, no further checks are performed. The extended checks are classified into the following categories:

- **References to program external units:** Verifies that the external program units exist and interfaces are used correctly. Program external units are:
  - *Transactions*, which are called with `CALL TRANSACTION 'tcod'`
  - *Dialog modules*, which are called by `CALL DIALOG 'dialog'`
  - *Reports*, which are called with `SUBMIT 'report'`

  - *Function units*, which are called by `CALL FUNCTION 'func'`
  - *External forms*, which are called by `PERFORM form IN PROGRAM program`
  - *Screens*, which are called by `CALL SCREEN nnnn`
  - *Global messages*, which are referred by `MESSAGE Ennn`
  - *User interfaces*, which are referred by `SET PF-STATUS` and `SET TITLE-BAR`
  - *Authority objects*, which are referred by `AUTHORITY-CHECK`

- **Multi-language enabling:** Searches for constructs that hamper the use of a program in different languages — for example, text literals without text IDs. Text literals appear in the language in which they were typed, and are not processed by translation services.
- **Package check:** Detects the illegal use of objects from other packages.
- **EBCDIC/ASCII portability:** Detects whether a program behaves differently in EBCDIC and ASCII (e.g., the comparison of character fields).
- **Generation limits:** Determines if generation limits, such as the maximum number of data objects, are close to being reached.
- **Statements in wrong context:** Scans for statements that are used in an inappropriate language context. For example, the `COMMIT WORK` statement within a `SELECT ... ENDSELECT` loop leads to the loss of the database cursor.
- **Unnecessary items:** Searches for form subroutines that are not used in a program, or fields that do not have read access.

Program Generation

Since every program generation starts with a syntax check, this check is very similar to the others in this category. The relevant difference is the fact that this check also generates the load format (byte code) of a program. You might want to use it to precompile programs to avoid compilation when a user wants to execute a program (which results in delayed system response times). Program generation is also useful for removing inconsistencies that sometimes appear in an SAP system during a system upgrade.

Security Checks

Some ABAP statements can endanger stability, data integrity, and overall program security when used carelessly or with bad intent. The Code Inspector security checks inform you of:

- Use of a statement that is deemed critical
- Use of a statement that infers ominous database access
- Selected statements that don't handle the system return code

Critical Statements

This check searches for any of the following types of statements that are considered critical, either for system security reasons or because they could endanger program stability:

- **Internal statements:** Some ABAP statements are provided for internal use only in SAP programs. For example, the `SYSTEM-CALL` statement interacts with the ABAP kernel. SAP may change these statements at any time without notice, leading to incompatibilities in your program. You should know if and when these statements are being used.
- **Statements where authority checks are necessary:** In the SAP system, automatic authority checks protect the invoking of transactions

and reports. For performance reasons, these automatic checks are only performed if the items are called directly by a user. If they are called internally by a program, the authority checks must be performed by the calling program. You should monitor compliance with this security requirement.

- **Database operations:** The `EXEC SQL` statement accesses database tables bypassing the SAP database interface. The SQL syntax within the statement may be specific to one database system, thereby impeding program portability. Native SQL statements also bypass the SAP table buffers, which can lead to inconsistencies when buffered tables are accessed. You should identify instances where this condition exists to review for possible impacts on data consistency.

  In addition, the `ROLLBACK WORK` statement reverts all database changes that were executed since the last database commit statement. Incorrect use of this statement can lead to serious inconsistencies in the transaction.

  A database hint (that is, a guideline for the

Figure 12 Statements Covered in the Critical Statements Check

Problem Area	Type of Statement	Examined Statements	Priority
Internal statement	Call of system functions	CALL 'cfunc'	Warning
	Call of system functionality	SYSTEM-CALL	Warning
	Generating programs and screens	GENERATE REPORT GENERATE SUBROUTINE POOL GENERATE DYNPRO	Information
Authority check	Call of transactions	CALL TRANSACTION	Information
	Call of reports	SUBMIT REPORT	Information
	Call of the editor	EDITOR-CALL	Information
Database	Use of Native SQL	EXEC ... ENDEXEC	Warning
	Use of database rollbacks	ROLLBACK WORK	Information
	Use of database hints	%_HINTS ...	Warning
Repository objects	Read a program or text pool	READ REPORT READ TEXTPOOL	Information
	Write/delete a program or text pool	INSERT/DELETE REPORT INSERT/DELETE TEXTPOOL	Warning
	Read a screen	IMPORT DYNPRO	Information
	Write/delete a screen	EXPORT/DELETE DYNPRO	Warning
	Read a global type entry	IMPORT NAMETAB	Information
	Write a global type entry	EXPORT NAMETAB	Warning

database optimizer regarding how to handle a statement) is specific to each database system. You should only use database hints as an exception, and evaluate their use carefully.

- **Reading or writing of SAP repository objects:** The main repository objects in an SAP system are programs, screens, and global types. The statements that manipulate these objects are intended for use by internal development tools only. Using these statements in application programs can potentially destroy the state of the system. Reading these objects can also be problematic because their data structures can change without any notice.

This check searches for all ABAP statements identified as critical, as described in **Figure 12**. You can select statement types individually or in any desired combination by setting the check parameters.

Access to Database Tables

This check searches for accesses to specific database tables. Some database tables contain critical information, such as personal data. Only programs that are protected with authorization objects should access these tables. This check can detect access to specific tables if the table name is used statically. You can specify a list of critical tables as a check parameter. In addition, this check can detect the following situations:

- Dynamic table accesses and dynamic `WHERE` clauses might be critical because they can hide an access to critical tables.

- The SAP system supports the concept of clients. Data of different clients should be strictly separated. In contrast with system programs, production applications should only be allowed to access the data of their own client.

Figure 13 presents the statements that you can examine with this check. (In fact, there are two checks, one for `SELECT` statements and one for changing database accesses.) Again, you can combine one or more different statement types by setting the check parameters.

Handling of the System Return Code SY-SUBRC

In the ABAP language, the system field `SY-SUBRC` returns the success of a statement. In some cases,

Figure 13 Statements Covered in the Access to Database Tables Check

Type of Statement	Examined Statements	Priority
Access to certain database tables	SELECT, INSERT, UPDATE, MODIFY, DELETE	Information
Dynamic table access	SELECT * FROM (dbtab) WHERE ... INSERT, UPDATE, MODIFY, DELETE (dbtab)	Information
Dynamic WHERE clause	SELECT * FROM dbtab WHERE (where_cond) UPDATE dbtab SET A = a WHERE (where_cond) DELETE dbtab WHERE (where_cond)	Information
Client-independent database access	SELECT, INSERT, UPDATE, MODIFY, DELETE ... CLIENT SPECIFIED	Warning

Figure 14 Statements Covered in the System Return Code Check

Type of Statement	Examined Statements	Priority
Modifying database operation	INSERT, UPDATE, MODIFY, DELETE	Warning
Reading database operation	SELECT, FETCH	Warning
Authority check	AUTHORITY-CHECK	Error
Call function modules or methods	CALL FUNCTION CALL METHOD	Information
Setting of locks (SAP enqueues)	CALL FUNCTION 'ENQUEUE_...'	Warning
Catching runtime errors	CATCH ... ENDCATCH	Information
Dynamic assigns to field symbols	ASSIGN * (f) TO <fs>	Information

not handling this return code can be suspicious. This situation is obvious in the case of the statement `AUTHORITY-CHECK`, where disregarding the return code means perform *no* authority check at all. In other statements, the return or export parameters can be undefined if `SY-SUBRC` is not 0, which can lead to application errors when the return code is ignored.

This check examines whether statements handle the system return code, as shown in **Figure 14**. Again, you can select the statement types and freely combine them by setting the check parameters. Additionally, you can specify any other statement in an input list.

Performance Checks

The Code Inspector was developed in cooperation with the SAP Performance and Benchmark Group, and one of the goals was to achieve a set of automated performance checks. Together with dynamic checks and expert performance sessions, these static checks can help evaluate adherence to the practical performance standard.

In short, these checks search for the following conditions:

- Database accesses that will lead to high runtimes because the `WHERE` clause does not use an existing database index
- `SELECT` statements that will implicitly bypass the SAP table buffers, thereby causing unnecessary database accesses
- `CHECK` statements inside `SELECT ... ENDSELECT` loops that sort out data read from the database, indicating incomplete `WHERE` clauses
- Nested loops over internal tables and nested `SELECT`s, which can be the source of non-linear runtime behavior

Analysis of the WHERE Clause for SELECT, UPDATE, and DELETE Statements

These checks identify `SELECT` statements (or `UPDATE/DELETE` statements) that cannot use any database index. They compare the database fields specified statically in the `WHERE` clause with the index definition in the ABAP dictionary. If there is no `WHERE` clause, or if the `WHERE` clause does not contain a field of a database table index, the database will perform a full table scan (or, for a client-dependent table, a scan of all entries of that client). This condition can be very harmful to performance.

Figure 15 Statements Covered in the WHERE Clause Analysis Check

Examined Statements	Priority
SELECT without a WHERE clause	Error / Warning*
In a WHERE clause, no field of a table index	
In a WHERE clause, no first (or second, for client-dependent tables) field of a table index	

* The message priority depends on the table size category (as defined in the technical settings for the table in the ABAP dictionary):
Table size category ≥ 2 → Priority = Error
Table size category < 2 → Priority = Warning

For the SELECT statement, the check considers only tables that are not buffered within the SAP table buffers. A full buffer scan — though itself a threat to program performance — is not deemed as critical as a full scan on a database table.

Web AS 6.20 supports two independent checks, one for SELECT statements and one for UPDATE and DELETE statements. **Figure 15** shows the options you can specify as check parameters; we recommend activating them all.

See the sidebar below for some pointers on what to do with the results from this check.

What to Do Next: If a WHERE Clause Cannot Use an Existing Database Table Index

Most important, *do not* create new database indexes for statements that are rarely used! Every index puts an additional load on the database, since it must be maintained whenever entries are inserted or deleted. Too many indexes also might confuse the database optimizer, increasing the risk of incorrect access plans. Obviously, only experienced developers should change the index design. But first try these suggestions for working around this situation:

- If the statement (form routine, method, program, etc.) is not used or needed anymore, just delete it.
- If the statement is only rarely used (for example, when it is part of a check program or tool that is only used exceptionally), mark the statement with the pseudo-comment "#EC CI_NOFIELD (for WHERE clauses that do not contain any index field) or "#EC CI_NOFIRST (for WHERE clauses that do not contain any first field of an index).
- If the statement is frequently executed, try to rewrite the WHERE clause so that an existing index can be used.
- If the statement is frequently executed and cannot be rewritten, either adjust the index design (be careful, changes on an index can also affect other statements) or create a new index (be careful, every additional index stresses the database).

Figure 16 Statements Covered in the Bypass Table Buffers Check

Examined Statements	Priority
SELECT on a buffered table in a JOIN	Warning
SELECT on a single record buffered table without the explicit SELECT SINGLE statement	
Generically buffered area not fully specified	
SELECT with sub-query	
SELECT with aggregate function: COUNT(), MIN(), MAX(), SUM(), AVG()	
SELECT DISTINCT ...	
SELECT ... GROUP BY ... [HAVING cond...]	
SELECT ... WHERE a IS [NOT] NULL	
SELECT ... ORDER BY, when the sort order differs from primary key	
SELECT with CLIENT SPECIFIED, without the client field in the WHERE clause	

SELECT Statements That Bypass the SAP Table Buffers

Small tables that only change rarely (especially customizing tables) should be buffered with the SAP table buffers, which notably accelerates read access. Since the SAP table buffers can only handle simple statements, omit more complex statements on buffered tables to avoid posing any risk to the performance advantage. In addition to the explicit bypassing buffers with `SELECT ... FOR UPDATE` and `SELECT ... BYPASSING BUFFER`, many statements bypass the buffer implicitly.

Figure 16 shows the most important types of `SELECT` statements that bypass the SAP table buffers in Web AS 6.20. This list could change in the future, such as if the capabilities of the SAP table buffers are extended. This check searches for all of these statements. However, the check has no parameters, so you cannot limit the search to a single statement type.

Note that this check does not cover the following two statements that also bypass the SAP table buffers:

- Use of Native SQL with `EXEC SQL ... ENDEXEC`
- Comparison between the fields of different database columns in the `WHERE` clause (for example, `SELECT ... FROM dbtab AS A WHERE KEYFIELD = L_KEY AND A~FIELD1 = A~FIELD2`)

While the second statement is very rarely used, you can detect Native SQL with the critical statements check.

See the sidebar on the next page for pointers on some next steps you can take once you have the results from this check.

SELECT ... ENDSELECT Loops That Contain a CHECK Statement

To minimize the data transfer from the database to the

What to Do Next: If a SELECT Statement Is Bypassing the SAP Table Buffers

Most important, *do not* rashly change the buffer settings of the respective table in order to buffer it more generously (that is, with fewer key fields). Keep in mind that changing buffer settings may have an impact on other statements. The SAP table buffer size is limited. Big tables that are fully buffered or buffered with a generic key area of only one or two fields may displace many smaller tables. Inserts or deletes on buffered tables lead to the invalidation of the buffered key range. If such situations happen frequently, table buffering becomes useless and simply resource-consuming. Consequently, only experienced developers should change buffer settings. But first try these suggestions for working around this situation:

- If the statement (form routine, method, program, etc.) is part of the customizing of your application, check whether you should add an explicit BYPASSING BUFFER statement to the SELECT statement. Maybe you *need* to bypass the buffer and are using the implicit bypass as a feature. But the list of statements that bypass the table buffers could change, so when you want bypassing, express this explicitly!

  If you don't need bypassing but cannot omit it, mark the SELECT statement with the appropriate pseudo-comment "#EC CI_... (in the Code Inspector results, the online documentation for the check provides the specific pseudo-comment to use).

- If the statement is frequently used in a production system, try to rewrite the WHERE clause so that the buffer can be used.

  You can replace JOINs that contain a buffered table with explicit SELECTs on each table. Or, you can replace DISTINCT and ORDER BY with ABAP functionality. (Remember that ORDER BY bypasses the buffer only if the sort order is not based on the primary key.) For example, you might use a SORT of the internal table, instead of sorting in the database with the ORDER BY statement. Of course, replacing these statements is only reasonable if the SELECT statement that fills the internal table can be satisfied by the SAP table buffer.

- Lastly, consider changing the buffer settings of the table. Look at the various table accesses in the where-used list. It may be that buffering the table is not reasonable at all.

application server, and to save database resources, only the data really needed in a program should be read from the database. To achieve this, the `WHERE` clause should restrict the size of the result set of a `SELECT` statement as much as possible.

This check searches for ABAP `CHECK` statements inside of `SELECT ... ENDSELECT` loops that sort out data after first reading them superfluously from the database. Instead of doing so, the `CHECK` condition should be incorporated into the `WHERE` clause.

A `CHECK` statement can be necessary in cases where a complex condition cannot be handled by the database interface — for example, if string operations are involved. In this case, the developer might choose to ignore the check, which again can be accomplished with a pseudo-comment.

Nested LOOPs and Nested SELECTs

The Code Inspector can also find nested `LOOP` and `SELECT` statements. Nested loops are a frequent source of non-scalable coding, because the runtime for loops over internal tables that are accessed sequentially increases linearly with the number of entries. Nested loops of this type can therefore have quadratic (or even worse) runtime behavior, meaning that doubling the number of entries will lead to four-fold (or higher) runtimes.

Normally, only a dynamic check can determine whether the code inside a loop will behave linearly. The same is true for nested `SELECT` statements; in the end, the developer must decide whether the coding is correct as is, or whether to replace a nested `SELECT` with a `JOIN` or a construct with `SELECT ... FOR ALL ENTRIES`.

Using the Code Inspector for Search Operations

The Code Inspector was designed primarily for static analysis of development objects. But the framework is also suitable for other purposes, such as testing for compliance with naming conventions, generating statistics about code key figures (such as the number of statements), and search operations. The first two checks are still under construction. However, with Web AS 6.20, the Code Inspector provides two search operations for use with ABAP programs:

- Search for single *tokens* (words)
- Search for complete *statements*

Of course, the ABAP Workbench offers search operations, too. However, Code Inspector search operations offer the following unique advantages:

- Pattern-based searching with wild-card characters "*" and "+"
- An arbitrary number of search patterns
- A parallel search mode that makes it applicable to a large number of objects
- Persisted search results

As you can imagine, Code Inspector search operations are *very* helpful when you need to reorganize many programs, such as when renaming procedures or replacing statements.

To search for single tokens or complete statements, create a check variant that only contains one or both of the search checks. You can specify check parameters of one or more search patterns with at least three characters. Then, to perform the search, simply create and run an inspection as usual. Matching patterns in your object set will appear with the priority "Information" in the standard Code Inspector result tree.

What Checks Are Coming Next?

The list of built-in checks currently includes syntax checks, performance checks, security checks, and search operations. SAP plans to continuously extend the number of built-in checks. If worthwhile and technically possible, new checks may ship with Web AS 6.10/6.20 support packages. Otherwise, they will be part of an upcoming Web AS release.

Figure 17 describes some of the checks that are currently under construction.

Helpful Hints

Here are some helpful hints to keep in mind as you set out to use the Code Inspector in your own application programs.

Figure 17 New Checks Currently Under Construction

Check Type	Description
Naming conventions	Consistent use of naming conventions for types, variables, procedures, and other entities in either a program or a package makes it much easier to understand the code and the data model. The Code Inspector will be able to check for compliance with most conceivable naming conventions.
Inefficient parameter passing	For procedure calls like PERFORM, CALL FUNCTION, and CALL METHOD, you can pass parameters in two ways: call-by-reference and call-by-value. Since the parameter value must be copied, call-by-value is always less efficient. For internal tables and large structures, the situation is admittedly worse than for simple integer values. This check searches for the usage of call-by-value and examines whether a simple switch to call-by-reference would be possible without further consideration.
Full table search in internal tables	Access to an internal table entry is fast when supported by an index. When no index can be used, the program must perform a sequential scan of all table entries internally. For example, this situation occurs if a program reads a sorted internal table with a key that differs from its unique table key.
Statement statistic	To answer the question “How much code do I have?” the most important statistic is the number of ABAP statements in your program-like objects. This check will be able to distinguish between statements belonging to different types of modularization units.

☑ If you want to display a Code Inspector element, make sure that the local/global icon is set appropriately (refer back to the sidebar on page 9). For example, if you have created a global check variant, the Code Inspector will give you an error message if you try to display it with the icon set to “local.”

☑ Remember that there is a deletion date for object sets and inspections! If you want to keep these elements longer than 50 days, simply change the deletion date.

☑ If you save an object set, the object list can be empty for different reasons. If your selections specify sets of objects that have no intersection, the resulting object set will be empty. Also, if the flag “Selections Only” is set (refer back to Figure 4), no object list will be created. This will be done at the moment of the inspection run, if your selections are not in conflict with the following rules:

- In a customer system, putting an SAP object into an object set is not supported.
- In a customer system, make sure that you are working in the development system where the object was created.

☑ To create a check variant, don’t just click on the root checkbox of the check variant tree, which selects all checks. Some checks need additional attributes to be executable, so instead select the checks individually, as you need them.

☑ Before checking a large object set with a new check, or with a check where you are unsure about the number of messages it could produce (for example, one of the search operations), you should first test the check with a single test object. The test object should be one for which you expect to raise a message.

Conclusion

You now have a basic understanding of the powerful new Code Inspector tool, which allows you to perform static checks on your choice of repository objects:

- Check a single object (such as a program or a class) by simply invoking the Code Inspector from the menu of its ABAP Workbench editor.
- Check many objects at once by creating an object set with the Code Inspector (transaction SCI).
- Combine a set of individual checks into a check variant by using the simple checkbox interface to select them and any available parameters from the check variant tree.
- To fully specify a test, combine an object set and a check variant to an inspection. After running the inspection, view the results in a hierarchical tree format.

In addition, you learned about some of the important checks that come with the Code Inspector. These ready-to-use checks make it easy for you to scan your programs for some of the biggest potential problems, including performance and security issues. Plus, we took a glimpse into the future — new checks are continuously under construction.

But suppose you have a particular check that *you* always wanted to apply to your programs, and we didn't mention it in this article. We suggest a do-it-yourself project! Seriously, we're not being lazy — just highlighting how you can make full use of the Code Inspector. You can easily implement and plug your own new checks into the framework at any time, but that's outside of the scope of this article. For more details, go to the Performance home page in the SAP Service Marketplace at **http://service.sap.com/performance**. The *Media Library → Literature* folder contains both the Code Inspector user manual and guidance on how to build your own checks.

Randolf Eilenberger received his doctorate in physics at the University of Stuttgart. He joined SAP in 1998, and since 1999 has been a member of SAP's Performance and Benchmark Group. As a co-developer of the Code Inspector, he has also implemented most of the tool's performance checks. Randolf can be reached at randolf.eilenberger@sap.com.

Andreas Simon Schmitt studied and received his doctorate in computer science and electronic engineering at the Technical University of Darmstadt, Germany. He joined SAP in 1991 as a member of the ABAP Group (now the Business Programming Languages Group), and since then has participated in the design of the ABAP language. As a development architect, he is responsible for the ABAP Compiler. He has designed and developed several internal and external tools for static analysis of programs, documentation, and other objects. He can be reached at andreas.simon.schmitt@sap.com.

