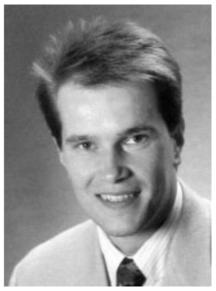
Introducing the next generation of ABAP debugging — the New ABAP Debugger

by Boris Gebhardt and Christoph Stöck

Boris Gebhardt
Senior Developer,
NetWeaver Application
Server ABAP,
SAP AG



Christoph Stöck
Development Architect,
NetWeaver Application
Server ABAP,
SAP AG

(Full bios appear on page 98.)

Some people think that using a debugger (in principle) is evidence of badly designed programs. But, as you know, nothing is that simple. Developing software is no longer just about writing standalone programs using well-known algorithms. Nowadays more and more time is spent integrating custom code into existing code — as in service-based framework architectures, for example. In such cases, it is essential to have proper interaction not only between custom plug-ins and the framework, but also between the custom plug-ins and all other participating plug-ins. When it comes to ensuring the integrity of these complex scenarios through debugging, developers run the risk of spending the majority of their time just preparing for the debugging session, sometimes just trying to locate the code of interest — time that would be much better spent on the debugging itself.

That's why debugging in ABAP can be so easy, even in complex scenarios. In contrast to other platforms, ABAP developers do not have to deal with questions such as: "Where is my corresponding source code?" or "Is the executable I want to analyze currently loaded in its debug version?" or "Why is remote debugging so complicated?" In the ABAP development environment, all you have to do is connect to your system to have access to the ABAP Debugger. That's it! Once you're logged on, all ABAP sources are available, and the ABAP virtual machine is ready to run the debugger immediately. Finally, remote debugging is nothing special in the ABAP development environment, because the ABAP Workbench, which integrates the debugger, is instantly accessible when an SAP GUI is up.

But the framework to bring the debugger up is one thing, the debugging tools themselves another. The ABAP Debugger that you know, now called the "Classic ABAP Debugger," has been in use for more than 10 years and continues to do its job quite well. But it does have its limitations — notably, there are some types of ABAP programs it cannot debug, and its user interface is rather rudimentary. To address these limitations, SAP introduced the "New ABAP Debugger" with SAP NetWeaver '04, which includes a new architecture that enables analysis of all ABAP programs and a state-of-the-art user interface. With SAP NetWeaver 2004s, the New ABAP Debugger has even more enhancements: integration of the new ABAP Front-End Editor into the new source code display, complete with syntax highlighting and a Data Quick Info pop-up; the Diff Tool, which guides you through the differences between, for example, two nested structures or internal tables; and the Data Explorer, which explains even complicated objects graphs at a glance.

In this article, we explain the architecture of the New ABAP Debugger, which has been developed from scratch to provide for a more flexible and intuitive user interface. We then guide you through the new world of ABAP debugging in SAP NetWeaver 2004s, describing all the new functionality, and with the help of a real-life debugging scenario, we explain how the New ABAP Debugger helps you to solve problems faster.

Although the focus of this article is on the New ABAP Debugger released with SAP NetWeaver 2004s, the majority of the features and functionality presented here are available in SAP NetWeaver '04, too. Moreover, we've added some helpful hints concerning the Classic ABAP Debugger. So even if you still have to wait for your development system to be upgraded to the latest SAP NetWeaver release, this article will provide you with useful tips and a deeper insight in debugging your ABAP programs.

Before we dive into the details, let's take a closer look at why SAP decided to develop a new debugger in the first place.

Why a new debugger?

To help you understand the rationale behind the development of the New ABAP Debugger, let's first look at the underlying architecture of the Classic ABAP Debugger.

The Classic ABAP Debugger is a "one process" debugger, meaning that the debugger and the debuggee[1] run in the same context — i.e., the same internal session. From a technical point of view, this means that the debugger shares its context data with the application being debugged and vice versa. From the user's perspective, this means that the debugger appears in the same SAP GUI window as the application being debugged.

The one-process architecture of the Classic ABAP Debugger has some significant drawbacks:

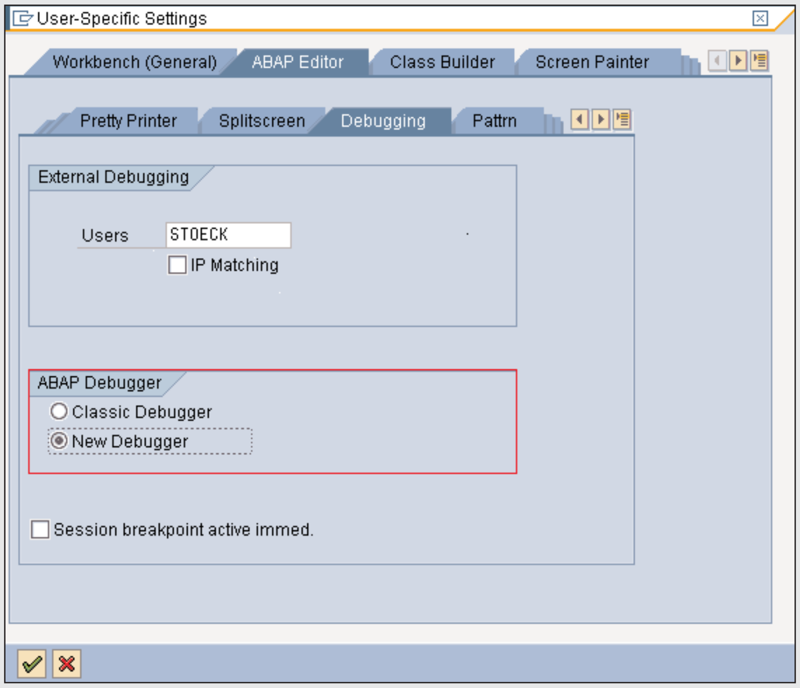
- **Not all ABAP code can be debugged:** It is not possible to debug conversion exits and field exits with the Classic ABAP Debugger. For example, inside the ABAP statement `WRITE`, a conversion exit can be executed by implicitly calling a user-defined function module that is specified in the domain of the statement's source field. The Classic ABAP Debugger is not able to debug conversion exits, because, during conversion exits, all dialog operations are forbidden, such as the ABAP statements `CALL DIALOG` and `CALL SCREEN`, therefore making it also impossible to send a debugger screen.

- **No state-of-the-art user interface:** Because the context of the debugger resides in the same internal session as the context of the debuggee, in principle, every functionality that is running for the debugger can potentially affect the debuggee's context. To avoid this risk, the Classic ABAP Debugger does not use ABAP for its own functionality, and therefore most of its coding is implemented inside the kernel. With this restriction, technologies for enhancing or creating a more appealing debugger user interface, such as the SAP Control Framework (CFW), which requires ABAP, cannot be used.

These drawbacks, especially the complaints about the "rustic" interface of the Classic ABAP Debugger, have resulted in the emergence of a completely new debugger built upon a completely different architecture.

For those who use the Classic ABAP Debugger, the good news is that it is not going anywhere (and neither are the many other available SAP

[1] The process that is being debugged.



External Debugging

Users STOECK

☐ IP Matching

ABAP Debugger

○ Classic Debugger

● New Debugger

☐ Session breakpoint active immed.

Figure 1 Selecting the New ABAP Debugger as the default debugger tool

troubleshooting tools).[2] In fact, you can switch between the Classic ABAP Debugger and the New ABAP Debugger at any time.[3]

Switching between the old and the new

Switching between the Classic ABAP Debugger and the New ABAP Debugger is easy — in the Object Navigator (transaction SE80) or in the ABAP Editor (transaction SE38), follow the menu path Utilities → Settings. In the User-Specific Settings dialog, select either the Classic Debugger or the New Debugger option, as shown **Figure 1**. The selected option then becomes your default debugger tool until you change it (in SAP NetWeaver ’04, the Classic ABAP Debugger is the initial default; in SAP NetWeaver 2004s, the New ABAP Debugger is the initial default).

Switching from the Classic ABAP Debugger to the New ABAP Debugger means that you will see — in addition to a completely new user interface — two other differences that reflect the contrasting architectures:

- The New ABAP Debugger opens in a separate SAP GUI window.
- The debuggee context — i.e., the subject being debugged — is the external session (roughly speaking, the SAP GUI window the debuggee is running in), rather than the internal session as in the case of the Classic ABAP Debugger.

[2] See the *SAP Professional Journal* articles “An Integrated Approach to Troubleshooting Your ABAP Programs: Using Standard SAP ‘Check’ Tools During Development and Testing” (March/April 2004); “An Integrated Approach to Troubleshooting Your ABAP Programs: Using Standard SAP Investigative Tools for Production Problems” (May/June 2004); and “An Integrated Approach to Troubleshooting Your ABAP Programs: Expert Tips for Making the Most of the ABAP Debugger” (July/August 2004).

[3] Just remember that not all ABAP coding can be debugged with the Classic ABAP Debugger. So it will, of course, not be possible to switch from the New to the Classic ABAP Debugger while debugging a conversion exit, for example.

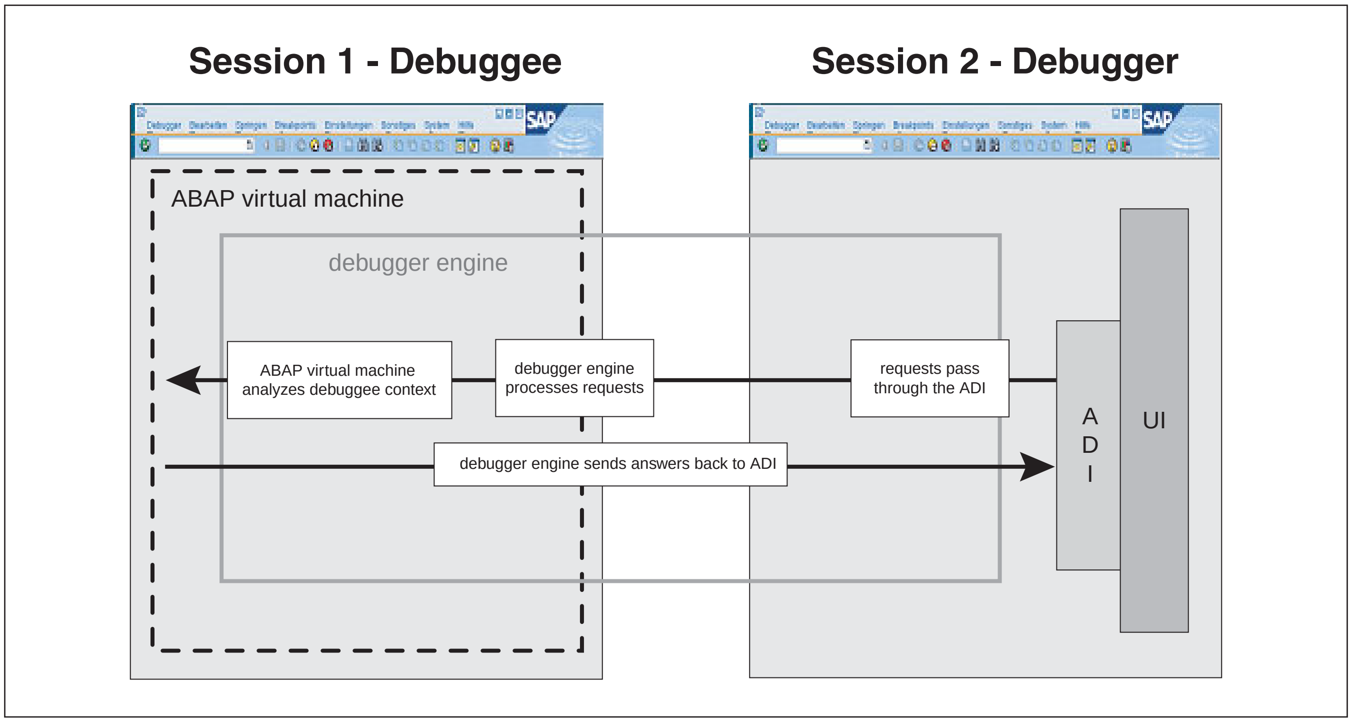

Session 1 - Debuggee
Session 2 - Debugger
ABAP virtual machine
debugger engine
ABAP virtual machine analyzes debuggee context
debugger engine processes requests
requests pass through the ADI
debugger engine sends answers back to ADI
ADI
UI

Figure 2 The two-process architecture of the New ABAP Debugger

Let's take a closer look at these differences in terms of the "two-process architecture" of the New ABAP Debugger.

The "two-process architecture" of the New ABAP Debugger

The basics of the two-process architecture of the New ABAP Debugger are illustrated in **Figure 2**.

The foundation of this architecture is its *two* external sessions, each of which appear in its own SAP GUI window: one session for the debuggee and another separate session for the debugger. The debuggee is controlled by the debugger via the debugger engine, which works deep inside the ABAP virtual machine. All requests coming from the debugger user interface have to pass through the ABAP Debugger Interface (ADI), where they are directed to the debuggee session via the debugger engine. The debugger engine then processes the requests by analyzing the debuggee's context (with the help of the debuggee's virtual machine) and sends the results back to the ADI of the debugger.

With this new architecture, the debugger user interface lives in its own internal session, integrated into a second, well-separated external session (SAP GUI window).

Separate SAP GUI windows for the debugger and the debuggee

The two-process architecture of the New ABAP Debugger means that you are working with two SAP GUI windows.

To simplify the handling of the two windows, the GUI focus automatically follows the active window. For example, when you enter "/h" in the command field of any application, the debugger window is created after the next action and it automatically gets the GUI focus. Clicking on the Continue button (or pressing F8) in the debugger window then deactivates the debugger window (unless the application stops at a breakpoint first) and activates the debuggee window.

Note!

Activating the debuggee window while the debugger is connected raises an important question — if no breakpoints are pending, how do you stop the debugger, since the debugger window is now inactive? The answer is actually quite simple. You can disconnect the debugger from the debuggee and close the debugger's window at any time by entering "/hx" in the command field of the debuggee window.[4]

This two-window approach, in which the debugger lives in its own external session, has significant advantages. In the Classic ABAP Debugger, which runs in the same internal session as the debuggee, running through specific statements such as `SUBMIT` or `CALL TRANSACTION` opens a new internal session, resulting in the loss of the current debugger instance and the start of another instance. As a consequence, there is a loss of all debugger breakpoints[5] currently set, all debugger settings, and all arrangements and fillings of the debugger tools (e.g., if you are displaying an internal table with a customized arrangement of rows in the Table View, the information about that internal table and its table row specifications are lost). In the New ABAP Debugger, since the debugger is in a separate external session, opening another internal session on the debuggee side does not result in the loss of the current instance — therefore the current breakpoints, settings, and the arrangements and fillings of the debugger tools are not lost.

[4] If you want the New ABAP Debugger to behave exactly like the Classic ABAP Debugger (e.g., terminate at the end of the internal session, or the roll area), you can modify the settings to enable options such as "Close debugger after continue (F8) and roll area end."

[5] This holds true only for the SAP NetWeaver '04 release. In SAP NetWeaver 2004s, the scope of the debugger breakpoints was expanded for the Classic ABAP Debugger so it is now the same as for the New ABAP Debugger, which is described further in this article.

The two-window approach also means that the coding of the debugger poses no threat to the integrity of the application being debugged. Because of this, in contrast to the Classic ABAP Debugger, complex ABAP coding can be used to present a state-of-the-art user interface.

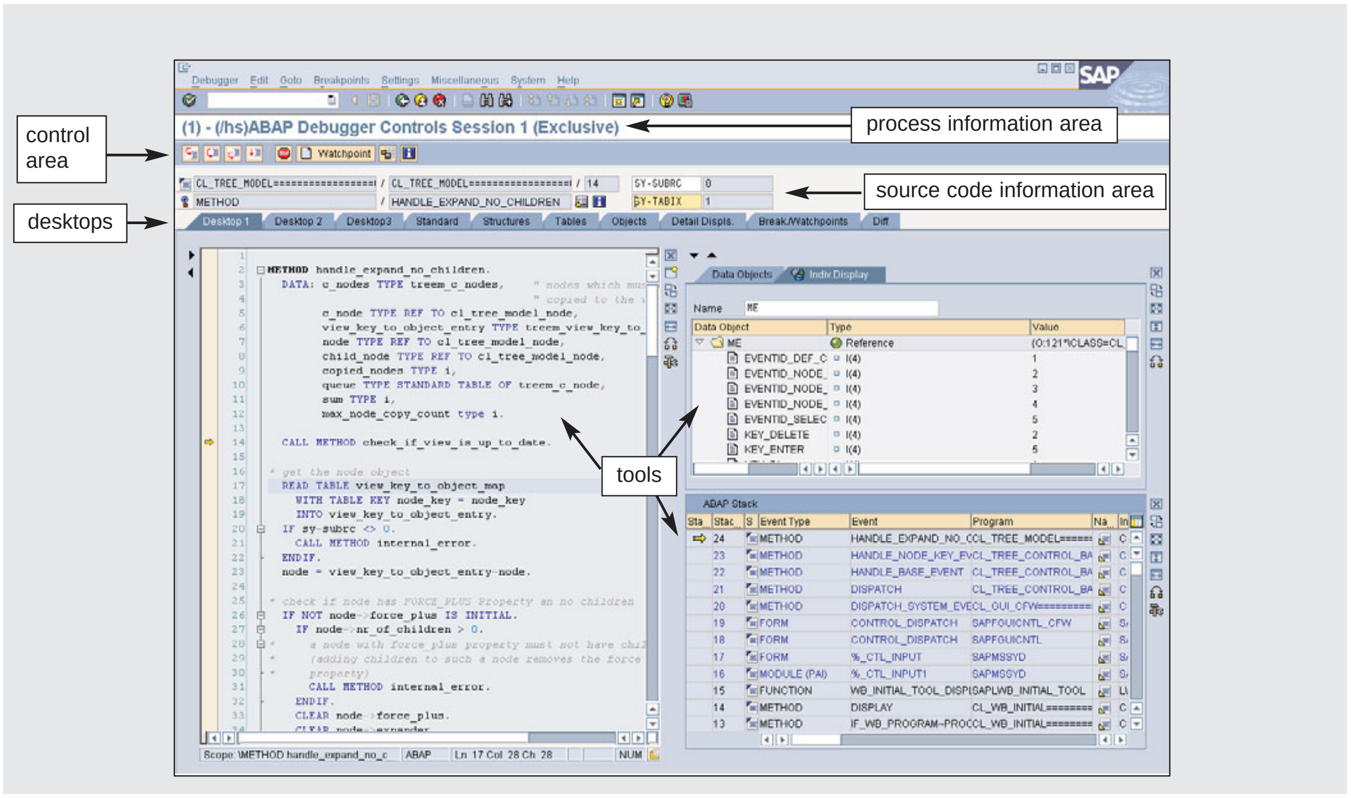
The New ABAP Debugger user interface

The New ABAP Debugger provides a state-of-the-art user interface that can be customized to your needs. The intent of this new user interface is to improve the productivity of debugging sessions in ABAP support and development.

Before exploring how the New ABAP Debugger user interface can achieve greater productivity, let's focus on the following basic questions about it:

- What are the main parts of the New ABAP Debugger user interface?
- How can I customize the user interface to my needs?
- Which debugger tools are available through the user interface, and how do I work with them?

After answering these questions, we'll explore the new world of debugging possibilities provided by the New ABAP Debugger. We'll look at two of the standard debugging features of the New ABAP Debugger. We'll first look at the new source code display tool, which leverages the new ABAP Front-End Editor. This tool provides syntax coloring and allows you to display variable values in a data tooltip just by pointing the cursor at the variable's symbol directly in the source code. We'll then turn our attention to the Variable Fast Display, which shows all global variables and all local variables (including all parameters) at a glance. We will also describe in detail how the New ABAP Debugger supports exploring ABAP data structures with specialized detail views (e.g., a Table View for internal tables, an Object View for classes or objects, etc.), and how you can use the Data Explorer to display complex and nested data types in a tree-like structure.



```
CL_TREE_MODEL================= / CL_TREE_MODEL================= / 14     SY-SUBRC  0
METHOD                         / HANDLE_EXPAND_NO_CHILDREN             SY-TABIX  1
```

```
METHOD handle_expand_no_children.
  DATA: c_nodes TYPE treem_c_nodes,        " nodes which mus
                                           " copied to the v
        c_node TYPE REF TO cl_tree_model_node,
        view_key_to_object_entry TYPE treem_view_key_to_
        node TYPE REF TO cl_tree_model_node,
        child_node TYPE REF TO cl_tree_model_node,
        copied_nodes TYPE i,
        queue TYPE STANDARD TABLE OF treem_c_node,
        sum TYPE i,
        max_node_copy_count type i.

  CALL METHOD check_if_view_is_up_to_date.

* get the node object
  READ TABLE view_key_to_object_map
    WITH TABLE KEY node_key = node_key
    INTO view_key_to_object_entry.
  IF sy-subrc <> 0.
    CALL METHOD internal_error.
  ENDIF.
  node = view_key_to_object_entry-node.

* check if node has FORCE_PLUS Property an no children
  IF NOT node->force_plus IS INITIAL.
    IF node->nr_of_children > 0.
*     a node with force_plus property must not have chil
*     (adding children to such a node removes the force
*     property)
      CALL METHOD internal_error.
    ENDIF.
    CLEAR node->force_plus.
```

```
Scope \METHOD handle_expand_no_c   ABAP   Ln 17 Col 28 Ch 28   NUM
```

Name: ME

Data Object	Type	Value
ME	Reference	{O:121*\CLASS=CL_
EVENTID_DEF_C	I(4)	1
EVENTID_NODE_	I(4)	2
EVENTID_NODE_	I(4)	3
EVENTID_NODE_	I(4)	4
EVENTID_SELEC	I(4)	5
KEY_DELETE	I(4)	2
KEY_ENTER	I(4)	5

Sta...	Stac...	S	Event Type	Event	Program	Na...	In
⇨	24		METHOD	HANDLE_EXPAND_NO_C	CL_TREE_MODEL======		C
	23		METHOD	HANDLE_NODE_KEY_EV	CL_TREE_CONTROL_BA		C
	22		METHOD	HANDLE_BASE_EVENT	CL_TREE_CONTROL_BA		C
	21		METHOD	DISPATCH	CL_TREE_CONTROL_BA		C
	20		METHOD	DISPATCH_SYSTEM_EVE	CL_GUI_CFW=========		C
	19		FORM	CONTROL_DISPATCH	SAPFGUICNTL_CFW		S
	18		FORM	CONTROL_DISPATCH	SAPFGUICNTL		S
	17		FORM	%_CTL_INPUT	SAPMSSYD		S
	16		MODULE (PAI)	%_CTL_INPUT1	SAPMSSYD		S
	15		FUNCTION	WB_INITIAL_TOOL_DISP	SAPLWB_INITIAL_TOOL		L
	14		METHOD	DISPLAY	CL_WB_INITIAL========		C
	13		METHOD	IF_WB_PROGRAM~PROC	CL_WB_INITIAL========		C

Figure 3 Main components of the New ABAP Debugger user interface

To help explain some of the more advanced features of the new debugger, such as the new DIFF tool, which allows you to compare arbitrary ABAP variables (e.g., two big internal tables) within a second one, we'll guide you through a real-world debugging session scenario. This will illustrate the impact of many of these new features on your daily work as an ABAP developer.

First things first, though — let's review the main components of the user interface of the New ABAP Debugger.

The New ABAP Debugger user interface

As shown in **Figure 3**, there are five main components of the New ABAP Debugger user interface:

- Process information area
- Control area
- Source code information area
- Desktops
- Tools

Let's take a moment to explore these components:

- **Process information area:** The process information (or title) area of the user interface provides information about the status of the debugger or debuggee. The following information is provided:
  - **Session number:** Because you can debug several applications in parallel, you need to know which debuggee session the debugger user interface is connected to.[6]
  - **Debug setting/session type:** Next in the

[6] Right-click on the SAP logon icon in the Microsoft Windows taskbar (at the lower right edge of your Windows desktop) to display all of your current sessions with the corresponding session numbers. You can also navigate to a particular session by selecting it from this display.

process information area is information[7] about the debugger/debuggee session type or status:

(/hs) indicates that system debugging is active.

-HTTP- indicates HTTP debugging.

-RFC->destination indicates debugging of an RFC module at the specified destination.

-UPDATE- indicates debugging of the asynchronous update functionality of a transaction.

-ATTACHED- indicates that the debugger is attached to a process (you can attach the debugger to a running process via transaction SM50 by following the menu path Process → Debugging).

- **Exclusive/Non-Exclusive** indicates whether the debugger session runs as an *exclusive* session or a *non-exclusive* session. In an exclusive debugging session, the work process is exclusively locked for your currently running debugging session; in a non-exclusive debugging session, the work process is not locked, and any debugger action may involve an implicit database commit during roll-in/roll-out of the debuggee context. The handling of exclusive and non-exclusive debugging sessions is almost identical for the New and Classic ABAP Debugger. See the sidebar below for more information on exclusive and non-exclusive sessions.

- **Control area:** Standard features for execution control (step into, step over, return, continue) in the New ABAP Debugger are similar to those in the Classic ABAP Debugger. The new debugger also provides shortcuts to create breakpoints and watchpoints. We discuss each of these features separately later in the article.

[7] For more information on debugger settings and system areas, see the article "An Integrated Approach to Troubleshooting Your ABAP Programs: Expert Tips for Making the Most of the ABAP Debugger" in the July/August 2004 issue of *SAP Professional Journal*, and also visit the "Download Files" page at www.SAPpro.com.

Exclusive/non-exclusive debugging sessions

Using the Classic ABAP Debugger, you probably have come across the "COMMIT WORK executed by system" message, which is displayed after each debug step. Or, when debugging a SELECT/ENDSELECT block, you have probably experienced an application crash with a DBIF_RSQL_INVALID_CURSOR runtime error, which announces that the database cursor has been lost. These are just some of the inconvenient things that can happen if you run what is referred to as a non-exclusive debugging session.

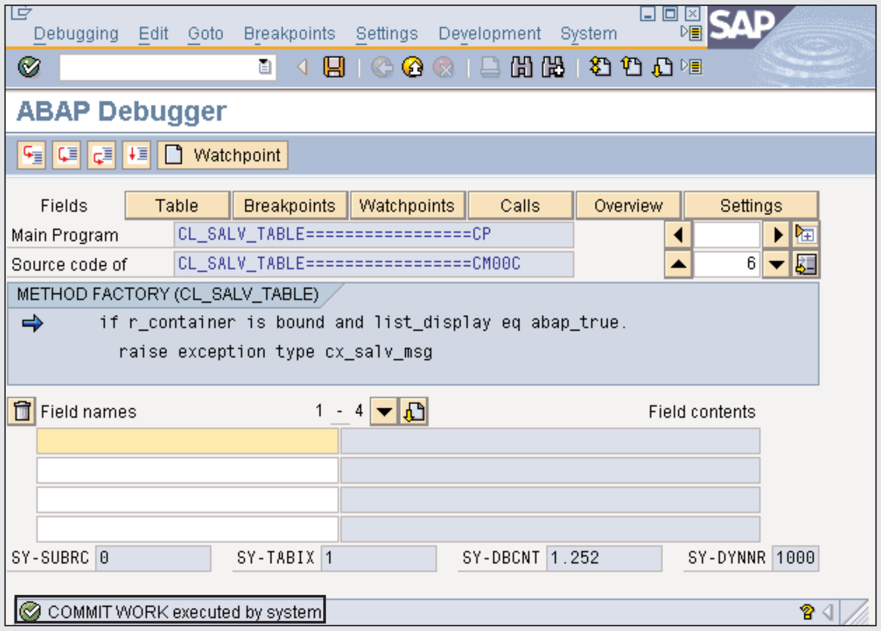
What is a non-exclusive debugging session?

If you run a normal transaction, your user context is rolled out of a work process if a screen is displayed, and rolled into the next free work process after the next user action. Because the database connection is linked to the work process, and a roll-in/roll-out may direct you to another work process, the database connection is closed during roll-out and is reopened during roll-in. Therefore, in order to preserve a consistent state, a standard roll-out is always connected to a database commit before the database connection is closed.

If the debugger screen is active, then the debuggee is rolled out and a database commit is executed. This is indicated in the Classic ABAP Debugger by the "COMMIT WORK executed by system" message, as shown in the screenshot on the next page.

Continues on next page

Continued from previous page



```
Main Program      CL_SALV_TABLE================CP
Source code of    CL_SALV_TABLE================CM00C
METHOD FACTORY (CL_SALV_TABLE)
      if r_container is bound and list_display eq abap_true.
        raise exception type cx_salv_msg
SY-SUBRC 0    SY-TABIX 1    SY-DBCNT 1.252    SY-DYNNR 1000
COMMIT WORK executed by system
```

If you debug a SELECT/ENDSELECT block, this commit introduced by the debugger closes the database cursor, and you receive the runtime error DBIF_RSQL_INVALID_CURSOR during the next fetch from the database. Especially when you are debugging the update functionality of a transaction, these database commits can lead to inconsistent application data, because a subsequent rollback in the application cannot roll back the database changes that were committed by the debugger.

For these reasons, there needs to be a better and safer way to debug productive code. This is where exclusive debugging sessions come into play.

What is an exclusive debugging session?

If you run an exclusive debugging session, the work process is *locked* for your debugging session. This means that no other user can use this work process, and as a consequence, there is no need for the system to close the database connection after a debugger roll-out because the roll-in will reenter the same work process. Therefore, in an exclusive debugging session, there are no debugger-introduced database commits, making it safe for you to debug updates and SELECT/ENDSELECT blocks.

When do you use which?

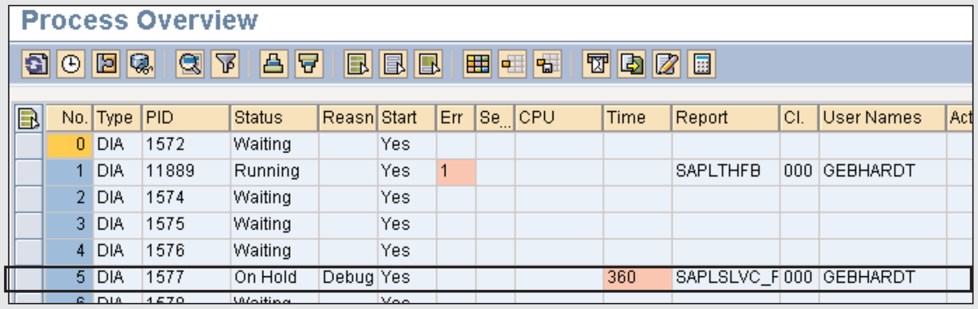
So if exclusive debugging sessions are better and safer, then why do we need non-exclusive debugging sessions?

Imagine you work on an application server with ten dialog work processes. Ten of your colleagues run debugging sessions in parallel in an exclusive session mode. The result is that all dialog work processes are locked. No other user can log on because there are no free dialog work processes. To address this situation, there needs to be a maximum number of exclusive debugging sessions, which you can set with the rdisp/wpdbug_max_no profile parameter. We recommend allowing a maximum of half of the available dialog processes be converted into exclusive debugging sessions.

The New ABAP Debugger also allows you to switch explicitly from a non-exclusive to an exclusive session via the main menu path Debugger → Exclusive Debugging mode switch on. If you run an exclusive debug session in the Classic ABAP Debugger, you can only check indirectly by using transaction SM50 (Process Overview). For an exclusive debugging session, you see the process status as "On Hold" and the reason "Debug" as shown in the screenshot on the right.

Note!

In a productive client, only exclusive debugging sessions are allowed in order to prevent any data corruption due to unintended database commits. If there is no free exclusive debugging session, you receive the DEBUGGING_IMPOSSIBLE runtime error.


Process Overview

No.	Type	PID	Status	Reasn	Start	Err	Se..	CPU	Time	Report	Cl.	User Names	Act
0	DIA	1572	Waiting		Yes								
1	DIA	11889	Running		Yes	1				SAPLTHFB	000	GEBHARDT	
2	DIA	1574	Waiting		Yes								
3	DIA	1575	Waiting		Yes								
4	DIA	1576	Waiting		Yes								
5	DIA	1577	On Hold	Debug	Yes				360	SAPLSLVC_F	000	GEBHARDT	
6	DIA	1578	Waiting		Yes								

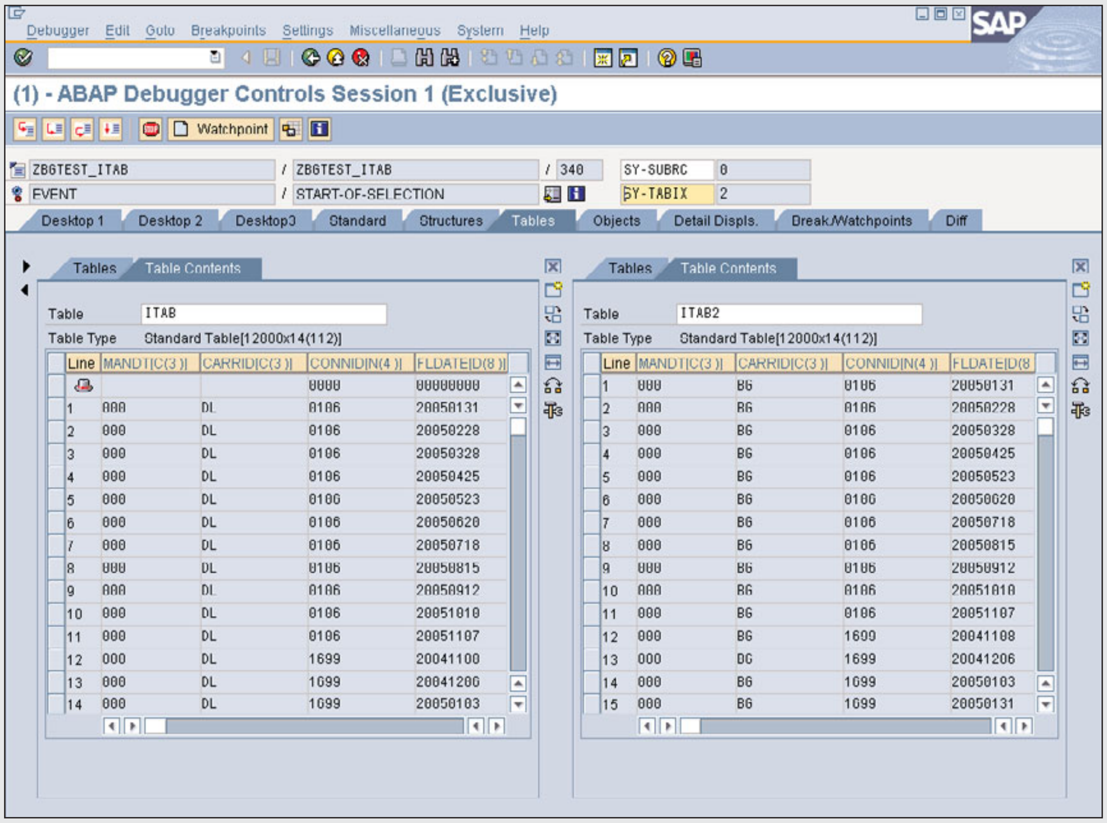
One thing to keep in mind when running an exclusive debugging session is that, as mentioned earlier, there are no implicit debugger database commits. While this makes debugging safer, it also means that there is an inherent risk of a database lock-wait situation.

Caution: database locks in exclusive debugging sessions

Imagine the following scenario: A very popular program is changed (e.g., SAPMV45A, which is used in transaction VA01, Create Order), and directly after this change, you start debugging transaction VA01. The system generates program SAPMV45A automatically and sets a database lock on the relevant database table line (D010SINF). Now you go for lunch. The debugger is still open, and the database lock remains during your break. Unfortunately, users running VA01 are now hung up on this lock because they cannot generate SAPMV45A. This lock-wait situation will not be resolved until you finish debugging or the system cancels your session with a TIME_OUT runtime error.

This problem is less likely if you run a system with an SAP Web AS kernel release greater than 6.10, which includes remote generation. Remote generation runs the generation and automatically executes the database commit at the end in a parallel work process.

While the New ABAP Debugger provides no implicit control, it does allow you to send explicit database commits and rollbacks to the debuggee in order to release any database locks (follow the menu path Edit → Database → Commit/Rollback). This is also possible with the Classic ABAP Debugger (follow the menu path Debugging → Database → Commit/Rollback).



(1) - ABAP Debugger Controls Session 1 (Exclusive)

ZBGTEST_ITAB	/ ZBGTEST_ITAB	/ 340	SY-SUBRC 0
EVENT	/ START-OF-SELECTION		SY-TABIX 2

Table ITAB

Table Type Standard Table[12000x14(112)]

Line	MANDT[C(3)]	CARRID[C(3)]	CONNID[N(4)]	FLDATE[D(8)]
			0000	00000000
1	000	DL	0106	20050131
2	000	DL	0106	20050228
3	000	DL	0106	20050328
4	000	DL	0106	20050425
5	000	DL	0106	20050523
6	000	DL	0106	20050620
7	000	DL	0106	20050718
8	000	DL	0106	20050815
9	000	DL	0106	20050912
10	000	DL	0106	20051010
11	000	DL	0106	20051107
12	000	DL	1699	20041108
13	000	DL	1699	20041206
14	000	DL	1699	20050103

Table ITAB2

Table Type Standard Table[12000x14(112)]

Line	MANDT[C(3)]	CARRID[C(3)]	CONNID[N(4)]	FLDATE[D(8
1	000	BG	0106	20050131
2	000	BG	0106	20050228
3	000	BG	0106	20050328
4	000	BG	0106	20050425
5	000	BG	0106	20050523
6	000	BG	0106	20050620
7	000	BG	0106	20050718
8	000	BG	0106	20050815
9	000	BG	0106	20050912
10	000	BG	0106	20051010
11	000	BG	0106	20051107
12	000	BG	1699	20041108
13	000	DG	1699	20041206
14	000	BG	1699	20050103
15	000	BG	1699	20050131

Figure 4 Using the Tables desktop to compare two tables

- **Source code information area:** The New ABAP Debugger displays the full information about the current source position. The displayed information depends on the current code type — in ABAP code you see "main program," "include," and "source line" information; in screen flow code, you see "main program," "screen number," and "source line" information; and in the code for a Business Server Pages (BSP) application, you see all the information about the current "BSP page" and "source line."

- **Desktops and tools:** Finally, we reach the most important part of the New ABAP Debugger user interface — the desktops and tools. The desktops are your work areas, and all available tools (e.g., the Source Code display tool and the Variable Fast Display, which are discussed in the section "The New ABAP Debugger tools") can be arranged on the desktops. You can configure the desktops to your needs and switch to specialized desktops for special debugging tasks (such as comparing two variables or analyzing an internal table in detail). **Figure 4** shows a comparison of two tables using the Tables desktop.

As you can see, there are three user-specific desktops (Desktop 1, Desktop 2, and Desktop 3) and seven standard desktops (Standard, Structures, Tables, Objects, etc.). You can customize which tools appear on any desktop (with a maximum of four tools on a desktop), along with the position and size of the tools on the desktop, but only the configuration of the three user-specific desktops can be stored permanently in a debugger variant. (For more on debugger variants, see the sidebar on the next page.) The seven standard desktops are preconfigured by the system and cover all essential debugging scenarios, so there is no need to customize them permanently.

Debugger variants

A debugger variant can consist of the following:

- Options (e.g., customizing the navigation to the Detail Views or special options for the different tools) and debugger settings (e.g., turn the system debugging on or off)
- Breakpoints
- User interface customizations (e.g., which tools are located on which desktop)

Debugger variants are stored on the database. To access them, you need the name of user who created the variants and the variant name. You can also download a debugger variant to a local file.

When can I use variants?

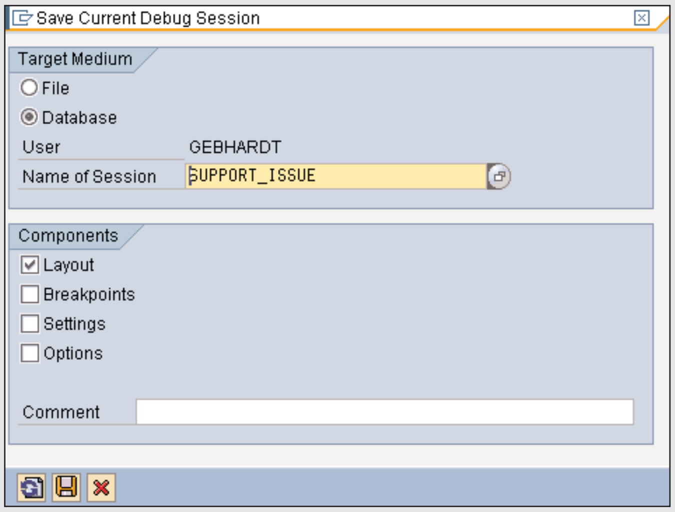
Imagine the following scenario: You are using the New ABAP Debugger to troubleshoot an ABAP problem. After some time, you find a specific function module has returned a wrong value. You want to pass this problem to a colleague, who has expert knowledge about this function module.

To ensure that your colleague can reproduce the problem, you will need to compose a very detailed description that identifies the source lines at which to set breakpoints, the debug settings that must be active, and so on. This can take a great deal of time, and there is a chance that you might forget critical information.

A smarter and more effective way to communicate the necessary details of the situation is to save the current debugger settings and breakpoints in a debugger variant. Then your colleague can just load the saved variant after starting a debugging session, and all settings and breakpoints will be active at once.

In the Save Current Debug Session dialog (shown in the screenshot below), you can specify which parts (layout, breakpoints, settings, and options) to store in the variant. As mentioned previously, you can save the variant as a local file, which you can then email to your colleague. Alternatively, you can leave the variant in its default location on the database, in which case your colleague simply uses the Load debugger variant dialog to load your variant from the database.

To load a debugger variant, you simply start the New ABAP Debugger for the application and then follow the menu path Debugger → Debugger Session → Load. In the components frame of the dialog, you can decide which parts of the variant you want to load in the debugger session. In the example, after loading the breakpoints and settings in the debugger, your colleague can directly start to analyze the error.



File [✓ Database]

User GEBHARDT
Name of Session SUPPORT_ISSUE

[✓ Layout] Breakpoints Settings Options

Comment

Continues on next page

Continued from previous page

The START_UP variant

There is one special debugger variant named START_UP that is always loaded when you start the New ABAP Debugger. You can customize your default debugging environment by modifying the START_UP variant. For example, if you customize the debugger user interface (to be more precise, your user desktops), and then save your changes by clicking on the Save layout button (), those changes are stored in the START_UP variant and will always be loaded when you start the debugger. You must explicitly load all other debugger variants by using menu path Debugger → Debugger Session → Load.

Note!

In the Classic ABAP Debugger, starting with SAP Web Application Server 6.20, a subset of this functionality (you can only save the current breakpoints and some debugger settings, such as the system debugging on/off option, in a session on the database) is available using the menu path Debugging → Sessions.

Let's take a closer look at how you can change the look of the debugger user interface to suit your needs and help you work more efficiently.

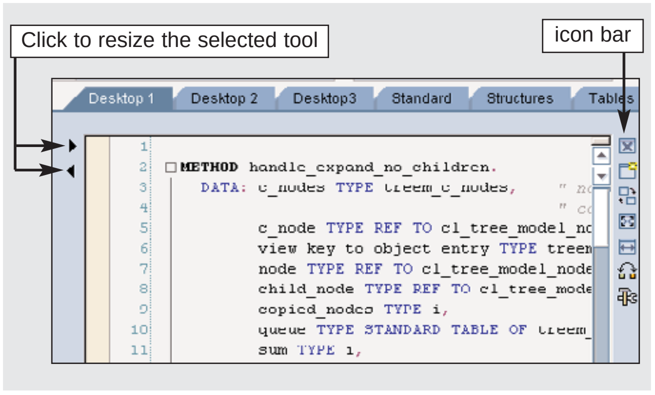
Customizing the debugger user interface

In **Figure 5**, you see a debugger tool (the Source Code display tool) residing on a desktop. It is automatically accompanied on the left by arrow symbols, which you can use to resize the tool, decreasing or increasing the size of the area it covers in the window. On the right side is an icon bar, which provides the following functionalities:

- Close (or remove) the tool.
- Choose a tool from the pop-up dialog, and add it to the current desktop.
- Exchange the tool with another tool selected from the pop-up dialog.
- Set the size of the tool to full-screen mode (obscuring all other tools).
- Maximize the tool horizontally/vertically (while rearranging the other tools accordingly).
- Swap the location of one tool with that of another tool.
- Invoke a tool-specific services dialog.

When you select the last functionality (invoke a tool-specific services dialog), a services dialog that is specific to the tool in use opens, like the one shown in **Figure 6**, which is specific to the Tables tool (we'll walk through the debugger tools in detail in a moment). The services dialog consists of two sections:

- **Standard:** Standard services are available for almost all tools. You can download the current tool content — for example, you can download the content of an internal table to a local file in order to analyze it by sorting the data, for example, in



```

METHOD handle_expand_no_children.
  DATA: c_nodes TYPE treem_c_nodes,      " no
                                         " co
          c_node TYPE REF TO cl_tree_model_no
          view key to object entry TYPE treem
          node TYPE REF TO cl_tree_model_node
          child_node TYPE REF TO cl_tree_mode
          copied_nodes TYPE i,
          queue TYPE STANDARD TABLE OF treem_
          sum TYPE i,
```

Figure 5 Customizing the layout of a tool

Microsoft Excel. You can also perform searches in the displayed content of the tool (press Ctrl-F to start a search and Ctrl-G to continue a search) — you could search for a special attribute of an object in the Object View, for example.

- **Tool-Specific:** This portion of the dialog presents all the specific features of the selected tool. As shown in the table-specific services dialog in Figure 6, there are tools for customizing the columns and for modifying the content of an internal table (e.g., changing or inserting a row). You even have the option to delete the table itself.

Now you know how easy it is to customize your debugger desktops by adding (or removing) any of the tools, and how easy it is to arrange them on the desktops. You can have up to four (even identical) tools on one desktop. For example, you can display four internal tables in parallel on one desktop using four instances of the Table View.

Moreover, there is a history of all your customizing changes available, so you can undo your changes if necessary. Let's say you opened four Table Views to compare four internal tables. Now you want to display one of these internal tables in detail. You switch to full screen for one Table View, which closes the other three automatically. It would be a nuisance if you had to re-create the three tools in order to get back to your four-table view after finishing your detail analysis. In the New ABAP Debugger, you can use the standard "Back" (F3) button to retrieve the last 10 debugger user interface configurations, along with all tool properties, such as the four previously displayed internal tables.

Now let's take a closer look at the tools available in the New ABAP Debugger.

New ABAP Debugger tools

The tools of the New ABAP Debugger are optimized working utilities for the different debugging scenarios you may encounter. Some of the tools you will recognize from the Classic ABAP Debugger; others are new.

If you add a new tool to the current desktop, you will get the pop-up window "New Tool" shown in **Figure 7**, where you can choose the appropriate tool. The tools are categorized into three sections:

- **Standard Tools:**
  - **Source Code:** Display current source code extract in a back-end editor.

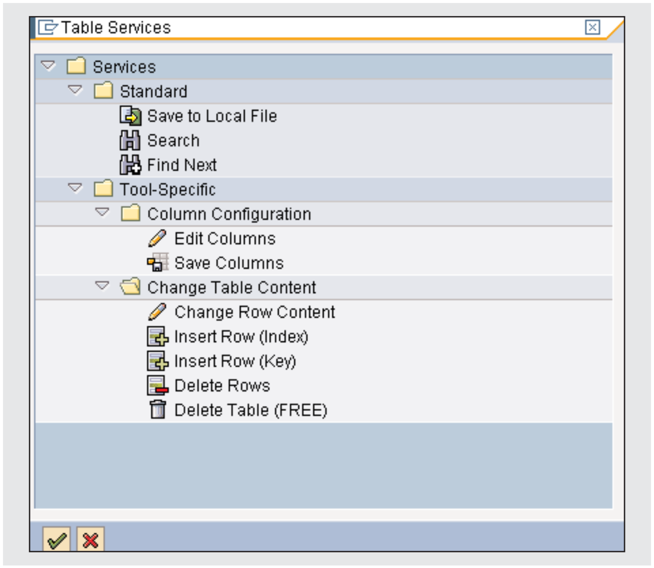


Table Services
- Services
  - Standard
    - Save to Local File
    - Search
    - Find Next
  - Tool-Specific
    - Column Configuration
      - Edit Columns
      - Save Columns
    - Change Table Content
      - Change Row Content
      - Insert Row (Index)
      - Insert Row (Key)
      - Delete Rows
      - Delete Table (FREE)

Figure 6 Table-specific services dialog

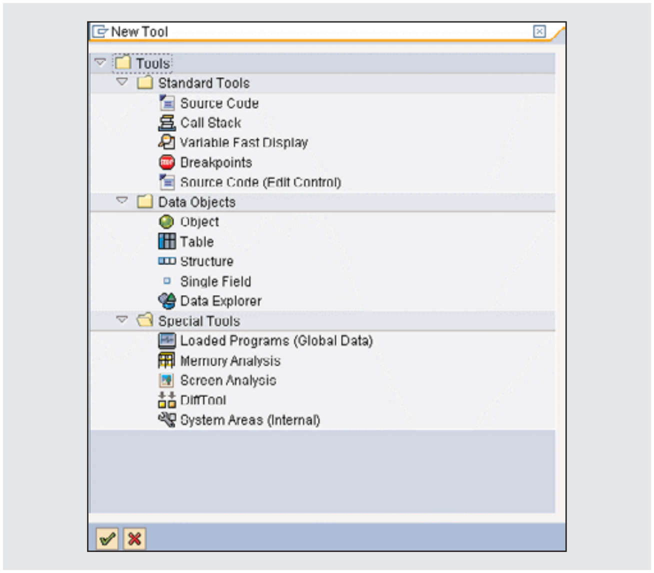

New Tool
- Tools
  - Standard Tools
    - Source Code
    - Call Stack
    - Variable Fast Display
    - Breakpoints
    - Source Code (Edit Control)
  - Data Objects
    - Object
    - Table
    - Structure
    - Single Field
    - Data Explorer
  - Special Tools
    - Loaded Programs (Global Data)
    - Memory Analysis
    - Screen Analysis
    - DiffTool
    - System Areas (Internal)

Figure 7 The New ABAP Debugger tools

- **Call Stack:** Display the current ABAP and screen stacks.
- **Variable Fast Display:** Display value and type of variables.
- **Breakpoints:** Maintain breakpoints, watchpoints, and checkpoints.
- **Source Code (Edit Control):** Display current source code extract in the new ABAP Front-End Editor.

Note!

As you can see, there are two Source Code tools. You can choose either of these display tools, which provide almost the same functionality. The first Source Code display tool uses a back-end editor (i.e., an old-fashioned, gray look, etc.); the second Source Code (Edit Control) tool relies on a new front-end edit control, which extends the tool's functionality to include features such as syntax coloring.

- **Data Objects:** Specialized tools (also referred to as detail views) for the analysis of variables with a dedicated ABAP data type:
  - **Object:** Display/change objects and classes.
  - **Table:** Display/change internal tables.
  - **Structure:** Display/change structures.
  - **Single Field:** Display/change variables with simple data types, such as C, N, D, T, STRING, etc.
  - **Data Explorer:** Display complex data structures in a tree-like structure.

  Double-clicking on a variable displayed in any tool launches the detail view that corresponds to its data type: an Object View, Table View, or Structure View.
- **Special Tools:**
  - **Loaded Programs (Global Data):** Display all loaded programs and their global variables.
  - **Memory Analysis:** Display integrated Memory Inspector.[8]
  - **Screen Analysis:** Display the screen attributes and the subscreen tree.
  - **Diff Tool:** Compare variables.
  - **System Areas (Internal):** Display internal information about the ABAP Interpreter.

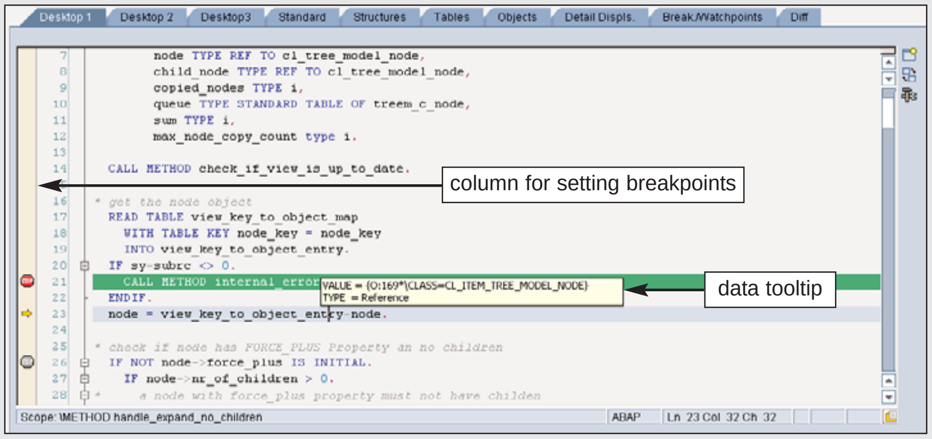
In the next sections we'll take a closer look at the most common debugging activities — displaying source code and the call stack, analyzing variables, and setting breakpoints and watchpoints. For further details on the tools that are not covered, please refer to the online documentation of the New ABAP Debugger.

Displaying source code and the call stack

The new ABAP Front-End Editor is available not only via the Source Code (Edit Control) display tool in the New ABAP Debugger, but also via the ABAP Workbench (ABAP Editor, Class Builder, and so on). The New ABAP Debugger leverages the following benefits of the new ABAP Front-End Editor (see **Figure 8**):

- ABAP code is easy to read because of real-time syntax coloring.
- Both vertical and horizontal free scrolling is possible.
- Processing block start and end (i.e., IF/ENDIF, LOOP/ENDLOOP) is highlighted in real time.
- Variable values and types appear in a Data Quick Info pop-up.
- Breakpoints are easily set using a breakpoint column.

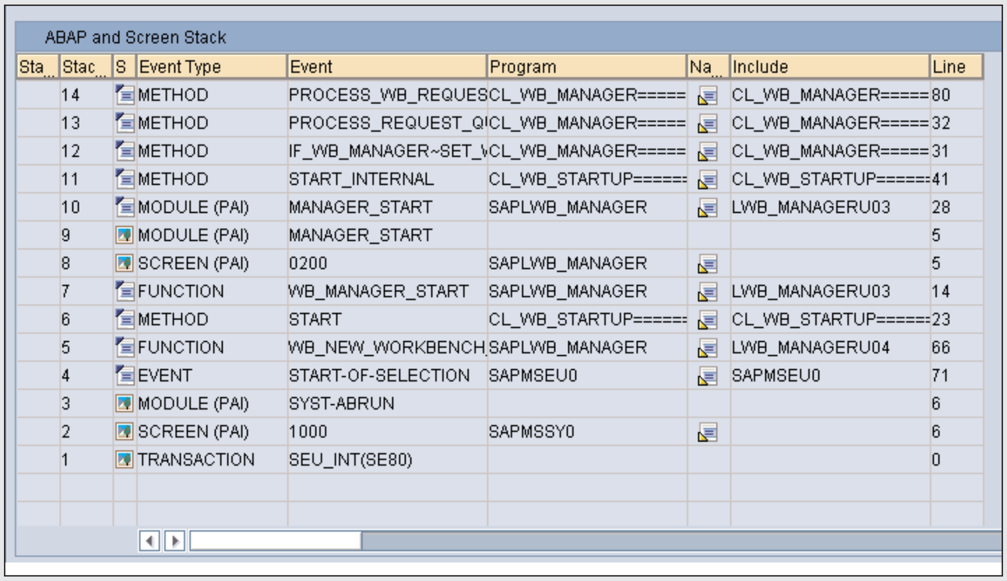
[8] The Memory Analysis tool covers almost all functionality of the stand-alone Memory Inspector (transaction S_MEMORY_INSPECTOR). For more on the Memory Inspector, see the article "Analyze Memory-Related Problems in Your ABAP Programs in Less Time and with Less Effort Using the ABAP Memory Inspector" (*SAP Professional Journal*, November/December 2004).



```
            node TYPE REF TO cl_tree_model_node,
            child_node TYPE REF TO cl_tree_model_node,
            copied_nodes TYPE i,
            queue TYPE STANDARD TABLE OF treem_c_node,
            sum TYPE i,
            max_node_copy_count type i.

   CALL METHOD check_if_view_is_up_to_date.

* get the node object
   READ TABLE view_key_to_object_map
     WITH TABLE KEY node_key = node_key
     INTO view_key_to_object_entry.
   IF sy-subrc <> 0.
     CALL METHOD internal_error
   ENDIF.
   node = view_key_to_object_entry-node.

* check if node has FORCE_PLUS Property an no children
   IF NOT node->force_plus IS INITIAL.
     IF node->nr_of_children > 0.
*       a node with force_plus property must not have childen
```

```
VALUE = {O:169*\CLASS=CL_ITEM_TREE_MODEL_NODE}
TYPE  = Reference
```

Figure 8 Using the new Source Code (Edit Control) tool to display source code



ABAP and Screen Stack

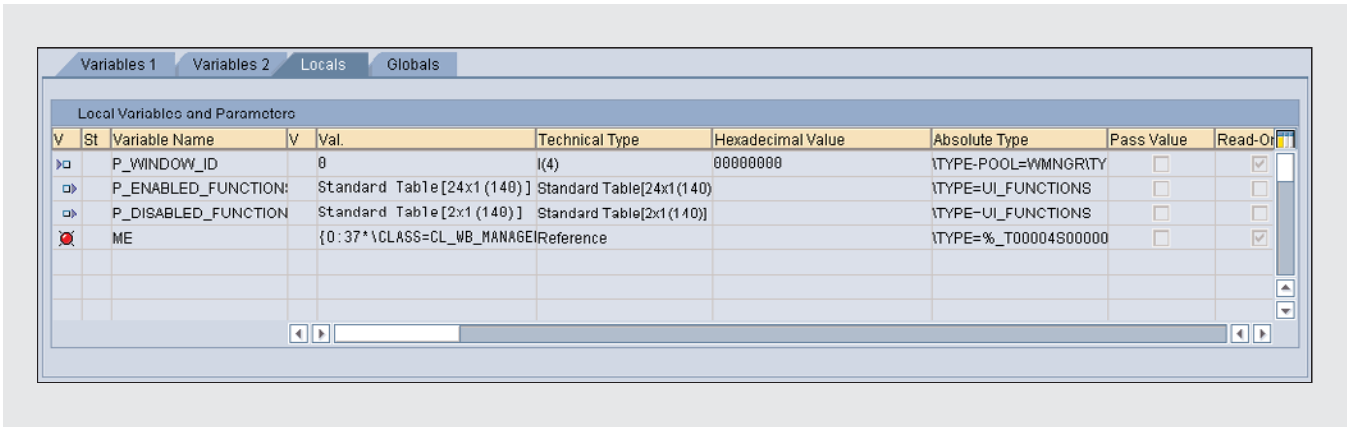
Sta	Stac	S	Event Type	Event	Program	Na	Include	Line
	14	▤	METHOD	PROCESS_WB_REQUES	CL_WB_MANAGER=====	▤	CL_WB_MANAGER=====	80
	13	▤	METHOD	PROCESS_REQUEST_Q	CL_WB_MANAGER=====	▤	CL_WB_MANAGER=====	32
	12	▤	METHOD	IF_WB_MANAGER~SET_V	CL_WB_MANAGER=====	▤	CL_WB_MANAGER=====	31
	11	▤	METHOD	START_INTERNAL	CL_WB_STARTUP======	▤	CL_WB_STARTUP======	41
	10	▤	MODULE (PAI)	MANAGER_START	SAPLWB_MANAGER	▤	LWB_MANAGERU03	28
	9	▣	MODULE (PAI)	MANAGER_START				5
	8	▣	SCREEN (PAI)	0200	SAPLWB_MANAGER	▤		5
	7	▤	FUNCTION	WB_MANAGER_START	SAPLWB_MANAGER	▤	LWB_MANAGERU03	14
	6	▤	METHOD	START	CL_WB_STARTUP======	▤	CL_WB_STARTUP======	23
	5	▤	FUNCTION	WB_NEW_WORKBENCH	SAPLWB_MANAGER	▤	LWB_MANAGERU04	66
	4	▤	EVENT	START-OF-SELECTION	SAPMSEU0	▤	SAPMSEU0	71
	3	▣	MODULE (PAI)	SYST-ABRUN				6
	2	▣	SCREEN (PAI)	1000	SAPMSSY0	▤		6
	1	▣	TRANSACTION	SEU_INT(SE80)				0

Figure 9 Using the Call Stack tool to display a combined ABAP and screen stack

Note!

The new Source Code (Edit Control) tool has an additional requirement — the SAP GUI version must be 6.40 PL 10 or higher.

The Call Stack tool displays the ABAP call stack. It allows you to switch to the context of each stack level and navigate to the source code by opening the ABAP Editor, in order to start a deeper analysis of the code in a separate window. Also new with SAP NetWeaver 2004s is the ability to display the screen (Dynpro) stack as well. Even more important, you can display a combined ABAP and screen stack to find out on which ABAP level which screen was called, and which screen invoked which ABAP module (see **Figure 9**).


Variables 1 | Variables 2 | Locals | Globals

Local Variables and Parameters

V	St	Variable Name	V	Val.	Technical Type	Hexadecimal Value	Absolute Type	Pass Value	Read-Or
		P_WINDOW_ID		0	I(4)	00000000	\TYPE-POOL=WMNGR\TY	☐	✓
		P_ENABLED_FUNCTION:		Standard Table[24x1(140)]	Standard Table[24x1(140)		\TYPE=UI_FUNCTIONS	☐	☐
		P_DISABLED_FUNCTION		Standard Table[2x1(140)]	Standard Table[2x1(140)]		\TYPE-UI_FUNCTIONS	☐	☐
●		ME		{0:37*\CLASS=CL_WB_MANAGEI	Reference		\TYPE=%_T00004S00000	☐	✓

Figure 10 Using the Variable Fast Display to show all local variables

Helpful hints for displaying source code and the call stack

- You can move the execution pointer in the new ABAP Front-End Editor by pointing at the target source line and using the context menu entry "Jump to statement."
- You can navigate from each stack line to the corresponding source line in the editor by clicking on the icon in the stack type column.

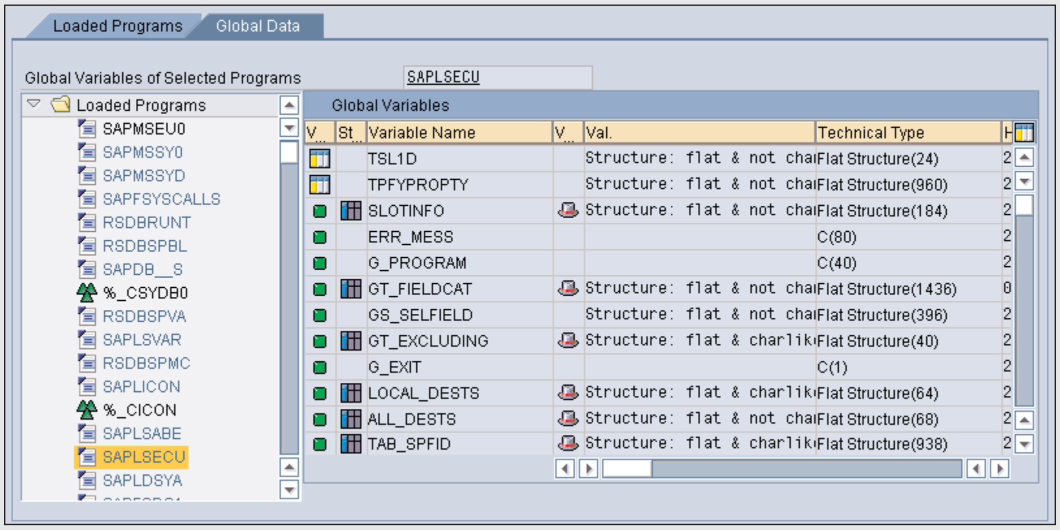
Analyzing variables

In addition to a state-of-the-art source code display, a convenient variable display and easy navigation method between linked variables are crucial components. Therefore, one of the new features of the New ABAP Debugger is the Variable Fast Display tool, which enables you to view basic information about a variable, such as the variable type, value, hexadecimal value, etc. To display this information, you select the tool and then type in the variable name, or you can double-click on the variable name (i.e., the symbol) in the source code to transfer the symbol to the Variable Fast Display. Even more convenient is the ability to display all global variables of the current program or all local variables (including the interface parameters) of the current procedure (see **Figure 10**).

If you are interested in a special global variable of another program that is part of the application, you can use the Loaded Programs (Global Data) tool, which allows you to browse all global variables of all loaded programs (see **Figure 11**), not just the current program. This tool also allows you to check out which programs have been used (i.e., loaded) in the current internal session. The information displayed indicates which programs compose the program group. (Please refer to the ABAP keyword documentation, using the keyword "program group," for details.)

One of the most significant benefits of the New ABAP Debugger is the ability to provide sophisticated and tailored detail views for all ABAP data types. The detail views are part of the New ABAP Debugger's powerful navigation system. Regardless of the tool that is active, you can navigate to a detail view by simply double-clicking on the variable in any tool (with the exception of the Source Code tool), and the variable automatically appears in the appropriate detail view. For example, double-clicking on line 2 of an internal table `ITAB` displays the line `ITAB[2]` in the Structure View. Or if you double-click on a cell in an internal table (`ITAB[2]-OREF`) that contains an object reference, this object is displayed in the Object View.

And for very complicated data structures, such as a complex nested object, it is more convenient to start an analysis in a tree-like display that allows you to dig deeper and deeper into the object graph without losing information about the levels above. The Data Explorer provides this functionality, as shown in **Figure 12**.



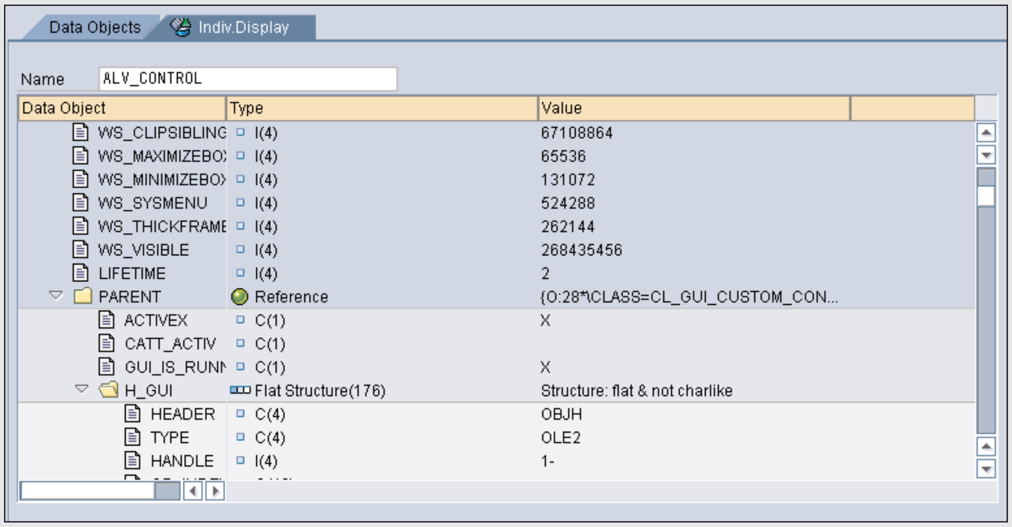
Global Variables of Selected Programs SAPLSECU

Loaded Programs
- SAPMSEU0
- SAPMSSY0
- SAPMSSYD
- SAPFSYSCALLS
- RSDBRUNT
- RSDBSPBL
- SAPDB__S
- %_CSYDB0
- RSDBSPVA
- SAPLSVAR
- RSDBSPMC
- SAPLICON
- %_CICON
- SAPLSABE
- SAPLSECU
- SAPLDSYA

Global Variables

V...	St...	Variable Name	V...	Val.	Technical Type	H
		TSL1D		Structure: flat & not chai	Flat Structure(24)	2
		TPFYPROPTY		Structure: flat & not chai	Flat Structure(960)	2
🟩		SLOTINFO		Structure: flat & not chai	Flat Structure(184)	2
🟩		ERR_MESS			C(80)	2
🟩		G_PROGRAM			C(40)	2
🟩		GT_FIELDCAT		Structure: flat & not chai	Flat Structure(1436)	0
🟩		GS_SELFIELD		Structure: flat & not chai	Flat Structure(396)	2
🟩		GT_EXCLUDING		Structure: flat & charlik	Flat Structure(40)	2
🟩		G_EXIT			C(1)	2
🟩		LOCAL_DESTS		Structure: flat & charlik	Flat Structure(64)	2
🟩		ALL_DESTS		Structure: flat & not chai	Flat Structure(68)	2
🟩		TAB_SPFID		Structure: flat & charlik	Flat Structure(938)	2

Figure 11 Using the Loaded Programs (Global Data) tool to display global variables of all loaded programs



Name ALV_CONTROL

Data Object	Type	Value
WS_CLIPSIBLING	I(4)	67108864
WS_MAXIMIZEBO)	I(4)	65536
WS_MINIMIZEBO)	I(4)	131072
WS_SYSMENU	I(4)	524288
WS_THICKFRAME	I(4)	262144
WS_VISIBLE	I(4)	268435456
LIFETIME	I(4)	2
PARENT	Reference	{O:28*\CLASS=CL_GUI_CUSTOM_CON...
ACTIVEX	C(1)	X
CATT_ACTIV	C(1)	
GUI_IS_RUNI	C(1)	X
H_GUI	Flat Structure(176)	Structure: flat & not charlike
HEADER	C(4)	OBJH
TYPE	C(4)	OLE2
HANDLE	I(4)	1-

Figure 12 Using the Data Explorer to show complex nested structures at a glance

Helpful hints for analyzing variables

- It is very time consuming to analyze the content of a variable that contains 2,000 bytes of hex code, for example. Therefore, the detail view for simple types and strings allows you to translate hex strings into readable text using a given code page (Field “View” → Text translation). For binary XML data, an XML viewer is available (Field “View” → XML browser).
- Imagine you want to perform a single test of a function module using the test framework of the Function Builder (transaction SE37). First you

need some input data before you can run the function module. The New ABAP Debugger allows you to download the current parameters of a function module as test data to the test framework of the Function Builder. Especially for big internal tables, this technique will save a tremendous amount of time compared with the time needed to insert the test data manually. You can access this functionality (when the debugger is in a function module) in the tool services of the Variable Fast Display tool by following the menu path Tool Specific → Save Parameters as Test Data.

- The Display References button in the Object View delivers all references that point to the displayed object. This makes it easy to figure out, during runtime, where the object is used (WHERE-USED list) and if the garbage collector will free this object instance.

Setting breakpoints and watchpoints

While breakpoints have not changed much between the Classic and New ABAP Debuggers, a few enhancements have been made to help make your debugging tasks easier. We'll look at these enhancements here in this section. In contrast to breakpoints, watchpoints in the Classic and New ABAP Debuggers are completely different implementations. We'll examine these differences as well.

Breakpoints

In ABAP there are three different kinds of breakpoints, each of which serve a specific purpose:

You can set **debugger breakpoints** inside the debugger. They exist only as long as the debugger is active. Once the debugger is closed, all debugger breakpoints are gone.

Usually, you set **session breakpoints** in the ABAP Editor. The scope of session breakpoints is the current logon session. This means that your session breakpoints are present in all external sessions (i.e., SAP GUI windows) of the current logon session. If you select the Save button in the Classic ABAP Debugger (in a dialog logon session), all current debugger breakpoints are automatically converted to session breakpoints.

If you set a breakpoint inside the coding of a Web Dynpro or BSP application, then the ABAP Workbench automatically sets a **user breakpoint**.[9] User breakpoints are stored in the SAP database, and they are valid for all logon sessions of the current user on the current application server. If you set a user breakpoint, then all following logon sessions of this user will have those breakpoints set. In the Classic ABAP Debugger, user breakpoints can be set by saving debugger breakpoints in an *external debugging session* (e.g., when debugging Web Dynpro or BSP applications).

In releases prior to SAP NetWeaver 2004s, only one breakpoint icon (the debugger breakpoint icon) was displayed in the debugger to denote all three types of breakpoints. In SAP NetWeaver 2004s, three different breakpoints icons are displayed in the New ABAP Debugger, as well as in the ABAP Editor, so you know immediately which kind of breakpoint you are dealing with.

Figure 13 describes the scope of the different breakpoint types. User breakpoints are valid for all logon sessions of a specific user on one server, session breakpoints are valid inside all external sessions of a single logon session, and debugger breakpoints only live inside a certain debugger instance.

In addition to the new icons, there are two main differences concerning the behavior of the breakpoints between former releases and SAP NetWeaver 2004s:

- **The scope of debugger breakpoints is expanded in the Classic ABAP Debugger:** As you now know, the Classic and New ABAP Debuggers have different debugging scopes. While the Classic ABAP Debugger debugs the internal session, the New ABAP Debugger debugs an external session. Of course, you would expect that both debuggers share the same breakpoint resources. But what about the scope of the debugger breakpoints? Does it extend to the internal or the external session?

[9] In releases before SAP NetWeaver 2004s, this kind of breakpoint was called an *external breakpoint*. The term is now *user breakpoint* because SAP NetWeaver 2004s increases its scope, which we explain next.

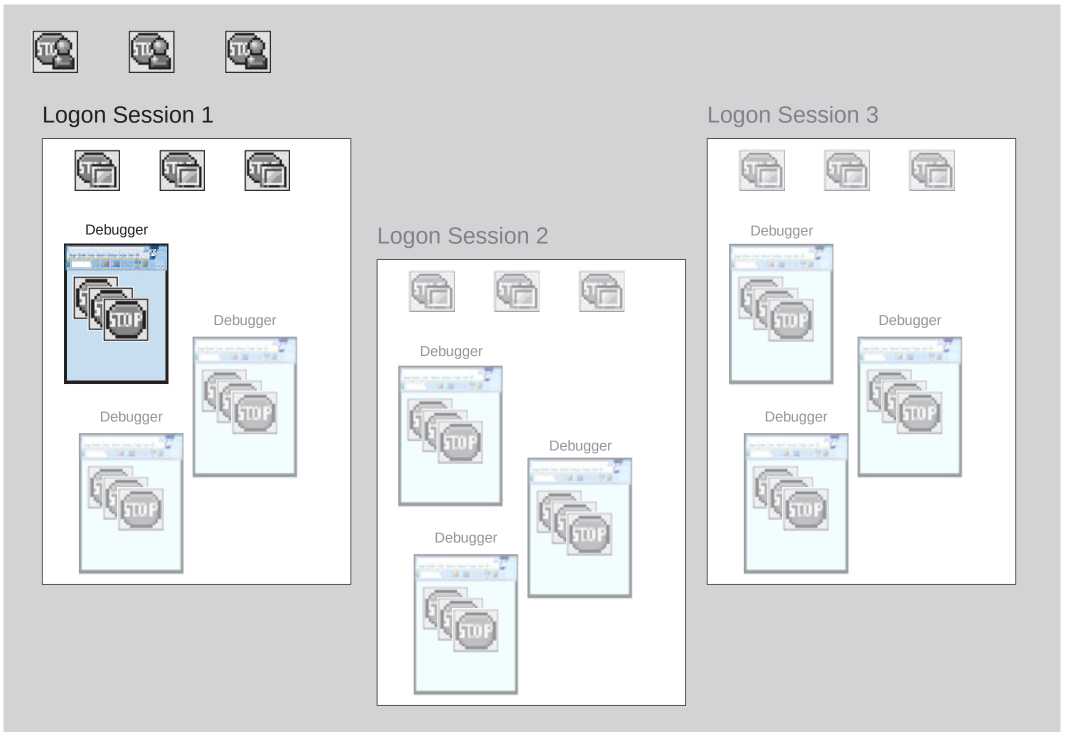


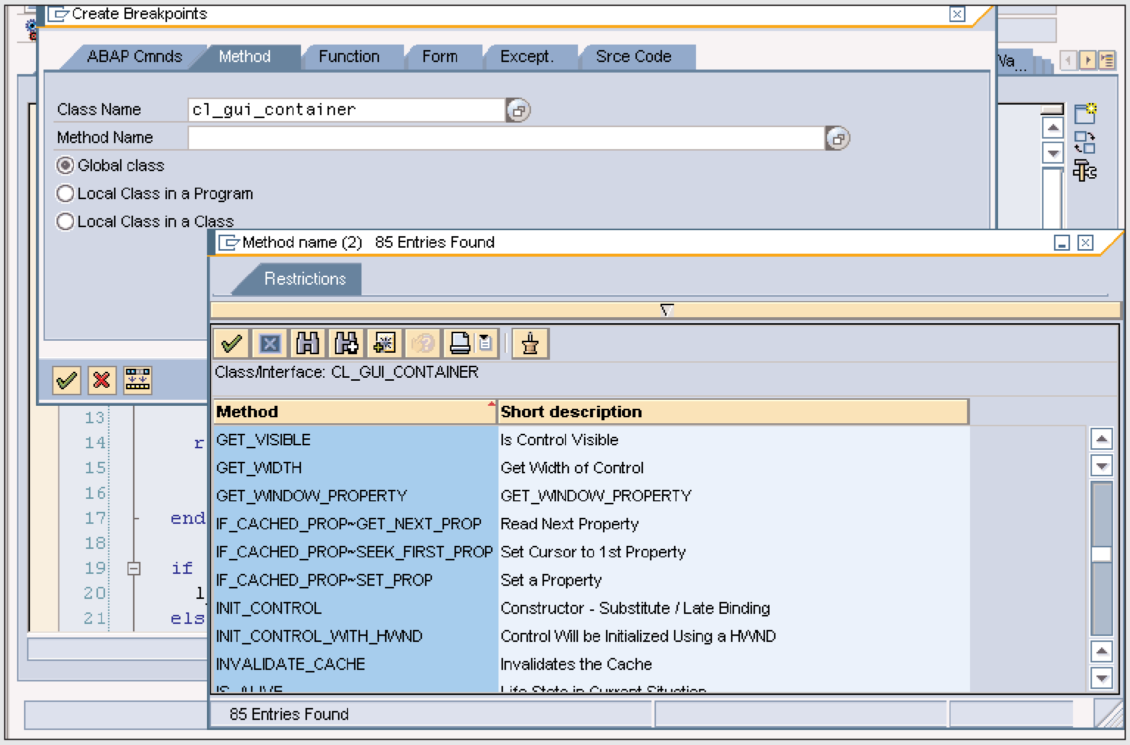
Figure 13 Scope of the three different kinds of breakpoints

This conflict has been resolved by expanding the scope of the debugger breakpoints for the Classic ABAP Debugger to the external session. This means if you use the Classic ABAP Debugger, and the debugger instance terminates at the end of the internal session, all the breakpoints you set during the debugging session survive and live until the external session is closed. That is exactly the behavior for breakpoints that are set in the New ABAP Debugger.

- **External breakpoints are now called user breakpoints:** It is not only the name that differs, but also the behavior: Before SAP NetWeaver 2004s the former *external breakpoints* could only be active in an *external debugging scenario* (e.g., when debugging a Web Dynpro or BSP application) in which you didn't start your application via a normal SAP GUI within the current dialog logon session, but via a browser that opened a new logon session where you could not set any breakpoints interactively. In dialog logon scenarios, the external breakpoints were not visible. With SAP NetWeaver 2004s, the former external breakpoints are *user breakpoints*, and any new logon session imports the relevant user breakpoints, regardless of whether it is a dialog logon session or not.

Setting breakpoints in the New ABAP Debugger is not much different from setting them in the Classic ABAP Debugger. There are some minor changes, which will make your debugging work easier:

- If you click on the breakpoint icon in the control area of the New ABAP Debugger, the Create Breakpoints dialog appears. Here you select whether you want the breakpoint to occur at ABAP commands, methods, functions, or forms, or whenever an exception is caught. You can also set a breakpoint at an *arbitrary* source code position (even in not-yet-loaded programs). This capability is not available in the Classic ABAP Debugger, in which you can set a source line breakpoint only by



Class Name: cl_gui_container
Method Name:
Global class [✓]
Local Class in a Program
Local Class in a Class

Method name (2) 85 Entries Found

Class/Interface: CL_GUI_CONTAINER

Method	Short description
GET_VISIBLE	Is Control Visible
GET_WIDTH	Get Width of Control
GET_WINDOW_PROPERTY	GET_WINDOW_PROPERTY
IF_CACHED_PROP~GET_NEXT_PROP	Read Next Property
IF_CACHED_PROP~SEEK_FIRST_PROP	Set Cursor to 1st Property
IF_CACHED_PROP~SET_PROP	Set a Property
INIT_CONTROL	Constructor - Substitute / Late Binding
INIT_CONTROL_WITH_HWND	Control Will be Initialized Using a HWND
INVALIDATE_CACHE	Invalidates the Cache

85 Entries Found

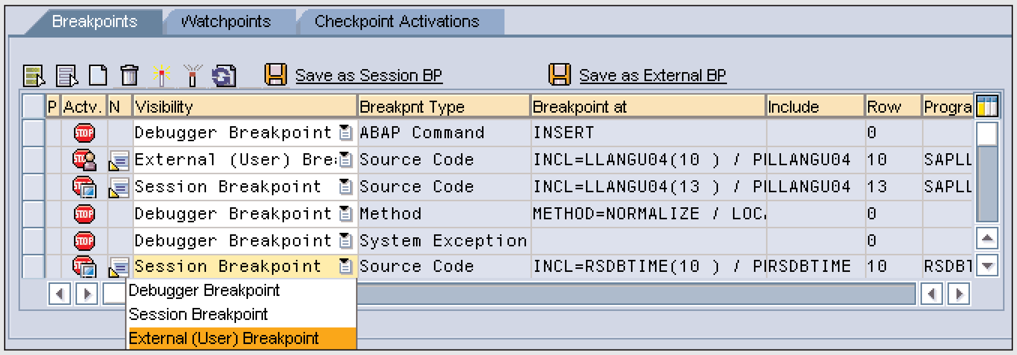
Figure 14 Press F4 to get help in the Create Breakpoints dialog

clicking on a source line in the currently displayed source code.

> ***Note!***
>
> There may be cases in which you need to set a breakpoint for a user other than yourself. For example, if the Web Dynpro or BSP application you are debugging is configured to run with a common user (and all users have been automatically logged on as INTERNET_USR), you can set a user breakpoint for the common user by entering INTERNET_USR in the Users field in the User-Specific Settings dialog (refer back to Figure 1).

- You can get help when you are setting breakpoints at a method, function, or form by pressing F4. For example, pressing F4 in the Method Name field on the Method tab displays all included methods of a given class, as shown in **Figure 14**. So even if you do not have all the details about the method or function module, you can still set a breakpoint. In the Classic ABAP Debugger, there is no such help available.

- Using the new Breakpoints tool, which is on the Break/Watchpoints desktop, you can administer all your currently set breakpoints. You can create, delete, activate, or inactivate breakpoints using this tool. In the Classic ABAP Debugger, you can only promote all breakpoints in one step by clicking on the Save button. The type of promotion is automatically selected, depending on the type of debugging session: In dialog logon sessions, debugger breakpoints are promoted to session breakpoints, and in



P	Actv.	N	Visibility	Breakpnt Type	Breakpoint at	Include	Row	Progra
	STOP		Debugger Breakpoint	ABAP Command	INSERT		0	
	STOP		External (User) Bre	Source Code	INCL=LLANGU04(10) / P	ILLANGU04	10	SAPLL
	STOP		Session Breakpoint	Source Code	INCL=LLANGU04(13) / P	ILLANGU04	13	SAPLL
	STOP		Debugger Breakpoint	Method	METHOD=NORMALIZE / LOC		0	
	STOP		Debugger Breakpoint	System Exception			0	
	STOP		Session Breakpoint	Source Code	INCL=RSDBTIME(10) / P	IRSDBTIME	10	RSDB1

Figure 15 Using the Breakpoints tool to change the breakpoint type

external debugging sessions, they are promoted to user breakpoints. In the New ABAP Debugger, you can treat the breakpoints individually. For example you can promote any single debugger breakpoint to a session breakpoint or any session breakpoint to a user breakpoint (see **Figure 15**).

Note!

Think of a situation in which your application ran into an endless loop. In most cases you probably had a good idea what the problem was, especially if you made changes just before the test run. Wouldn't it be helpful if you could force the running endless loop into the debugger, simple by setting a session breakpoint in a second ABAP Editor window?

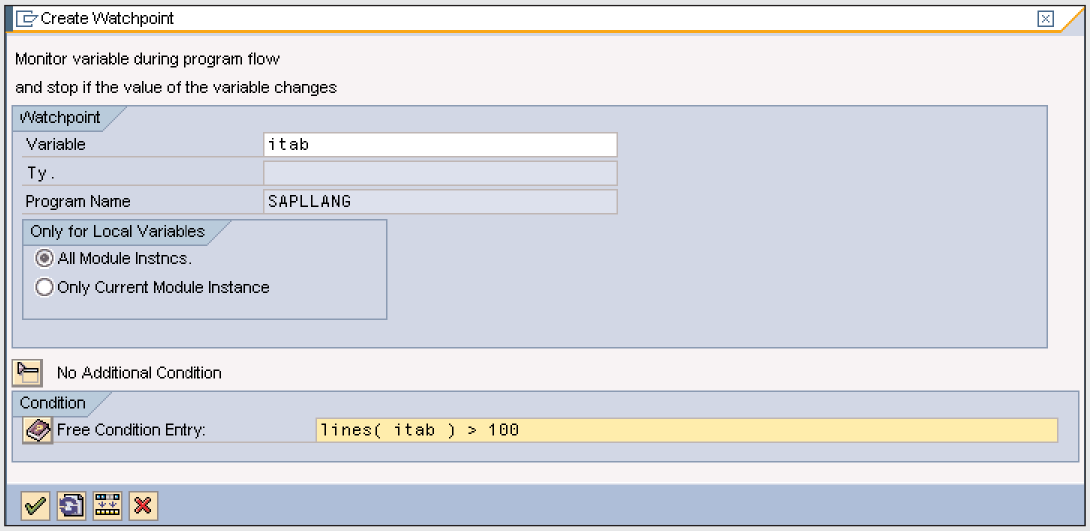
With SAP NetWeaver 2004s, you can. In the User-Specific Settings dialog (Figure 1), simply select the Session breakpoints active immediately option. When you activate this option, all external sessions (both running and waiting) of a logon session immediately inherit session breakpoints set in any of those external sessions. It is recommended that you activate this option in development environments.

Watchpoints

A watchpoint can be set on any variable with local or global scope to break the execution of the program when the variable's content changes. This makes it easier to analyze the program's data flow. It also leads directly to all the positions at which a certain variable has changed.

In order to create a watchpoint, click on the Watchpoint button in the control area of the New ABAP Debugger. In the pop-up dialog that appears, you can specify the variable for which the watchpoint should be set, the scope for a watchpoint on a local variable, and a free condition. To specify the scope for a watchpoint on a local variable means that you can set the watchpoint for only the current instance of the module on the execution stack in which the local variable lives, or for all instances of this module that will be pushed onto the stack. The latter is most likely to be your choice — if you want to stop execution after a method is called whenever a parameter of that method has a certain value, for example.

The ability to associate a free condition with the watchpoint is a convenient way to limit the number of "watchpoint reached" events according to your needs. For example, if you need to know where


Create Watchpoint

Monitor variable during program flow
and stop if the value of the variable changes

Watchpoint

Variable	itab
Ty.	
Program Name	SAPLLANG

Only for Local Variables

(● All Module Instncs.) ○ Only Current Module Instance

No Additional Condition

Condition

Free Condition Entry: `lines( itab ) > 100`

Figure 16 Create Watchpoint dialog

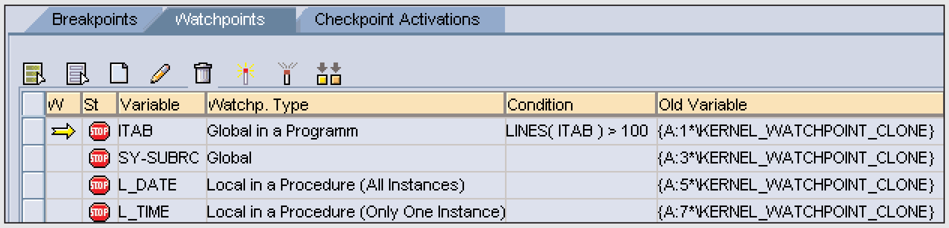
the internal table `ITAB` has increased to more than 100 lines, you can simply specify the condition `lines( itab ) > 100` for the watchpoint at `itab` (see **Figure 16**). (For details on setting a watchpoint to internal tables in the Classic ABAP Debugger, see the sidebar below.)

Setting a watchpoint to internal tables in the Classic ABAP Debugger

You may know that in contrast to the New ABAP Debugger, it is not possible to set watchpoints to internal tables in the Classic ABAP Debugger. This is a pity, since internal tables are heavily used and are often cumbersome to analyze — when the internal table is filled with dubious data, for example. But there is a low-grade workaround that works for most cases.

From inside the Classic ABAP Debugger, you can access not only the data of an internal table itself, but also some of the internal information that is used by the kernel to deal with this kind of data object — that is, the internal table *header* and *reference*. The header and reference of the internal table `ITAB`, for example, can be accessed in the debugger by the symbols `*ITAB` and `&ITAB`, respectively. The header contains, among other things, information about the number of lines internal table `ITAB` is filled with. So setting a watchpoint at `*ITAB` will stop the execution of the program whenever a line is inserted or deleted.

This may sound like an all-encompassing alternative to using direct watchpoints for internal tables in the new debugger — but it is not the same. For example, setting a watchpoint at the header of an internal table will not detect changes that leave the number of lines unchanged (e.g., update operations). Also, the conditions associated with watchpoints are less flexible in the Classic ABAP Debugger (e.g., specifying `lines( itab )` as an operand in the watchpoint condition is not possible).



W	St	Variable	Watchp. Type	Condition	Old Variable
⇨	🛑	ITAB	Global in a Programm	LINES(ITAB) > 100	{A:1*\KERNEL_WATCHPOINT_CLONE}
	🛑	SY-SUBRC	Global		{A:3*\KERNEL_WATCHPOINT_CLONE}
	🛑	L_DATE	Local in a Procedure (All Instances)		{A:5*\KERNEL_WATCHPOINT_CLONE}
	🛑	L_TIME	Local in a Procedure (Only One Instance)		{A:7*\KERNEL_WATCHPOINT_CLONE}

Figure 17 Watchpoints tab of the Breakpoints tool

All currently set watchpoints are listed in the Watchpoints tab of the Breakpoints tool. Here you can create, edit, delete, activate, or inactive watchpoints. In addition to information such as the variable name, the scope, and the condition, you will find in the watchpoint list a symbol for the "old variable" for all watchpoints. This information enables you to view the value of the variable of the recently hit watchpoint before it was changed.

For example, let's say you set a watchpoint at variable `L_TIME` and the watchpoint is "hit" during program execution because `L_TIME` changes from `11:15:00` to `11:30:00`, enabling you to access the old value of `11:15:00` via the symbol `{A:7*\KERNEL_WATCHPOINT_CLONE}` (see **Figure 17**). The comment `KERNEL_WATCHPOINT_CLONE` inside the symbol identifies the underlying data object from the kernel that is created as a "clone" of the watchpoint variable when the watchpoint is created. You can use this symbol as a normal variable symbol in all appropriate tools of the debugger.

Helpful hints for setting watchpoints

With the New ABAP Debugger, it is now possible to set watchpoints at internal tables.

- You can specify a condition along with watchpoints. It is a free-style condition for which you can choose two arbitrary operands (no need to specify the watchpoint variable, but you can) and one operator. Write the condition in the same way you would write it in ABAP.
- You can use two built-in functions as operands in the watchpoint condition: `lines( itab )` and `"strlen( str )"` (the number of lines of the internal table itab and the length of the string `str`, respectively).

Now that you know about the most important tools of the New ABAP Debugger, let's walk through a real-life debugging scenario.

Let's debug — a real-life debugging scenario

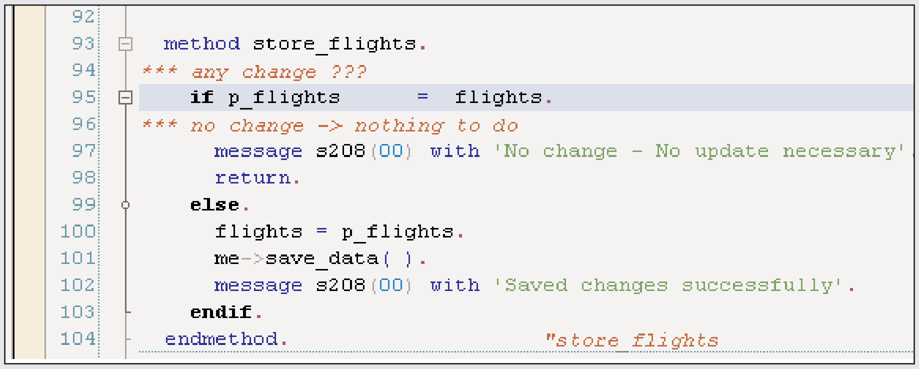
The following debugging scenario illustrates how easy troubleshooting can be with the New ABAP Debugger.

Imagine that you are running a transaction that allows the changing of some flight data in a list of flight connections for a number of different airlines. After clicking around in the flight list, you click on the Save button. Despite the fact you've changed nothing, you get the following message:

```
'Saved changes successfully'.
```

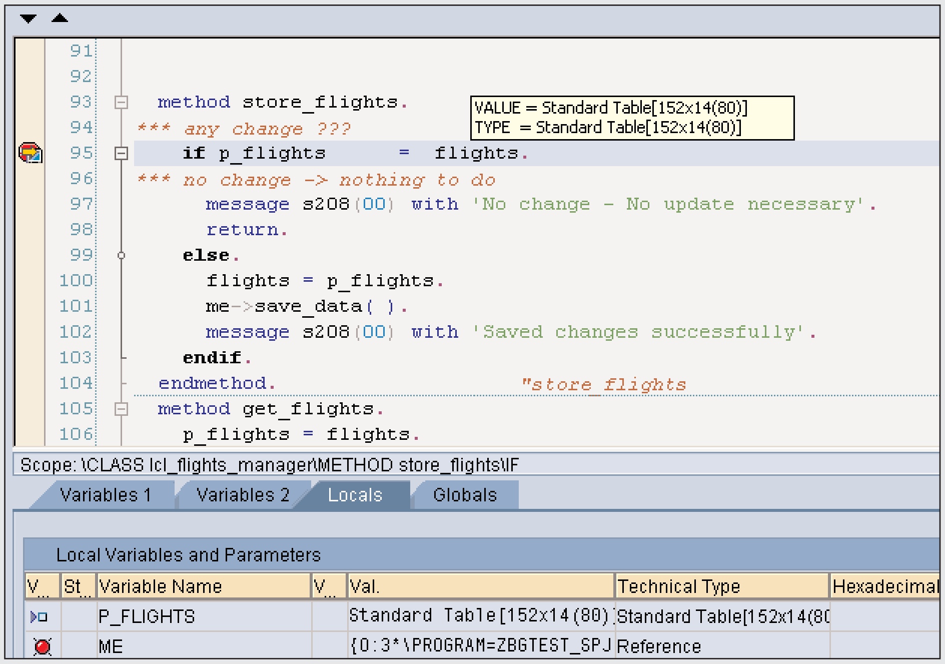
In order to analyze this error, you need to reproduce it. Run the transaction again, but this time run a SQL trace (transaction ST05) in parallel. This trace shows that the data is "updated" with the "unchanged" values. Let's find out what's going on here. First, a quick WHERE-USED search for the "successful" message leads to the code shown in **Figure 18** (which appears on the next page).

The message `'Saved changes successfully'` (line 102) appears in the `store_flights` method of a local class. Because no changes were made, we



```
 92
 93    method store_flights.
 94  *** any change ???
 95      if p_flights      =  flights.
 96  *** no change -> nothing to do
 97        message s208(00) with 'No change - No update necessary'
 98        return.
 99      else.
100        flights = p_flights.
101        me->save_data( ).
102        message s208(00) with 'Saved changes successfully'.
103      endif.
104    endmethod.                        "store_flights
```

Figure 18 Code that displays the “Saved changes successfully” message



```
 91
 92
 93    method store_flights.
 94  *** any change ???
 95      if p_flights      =  flights.
 96  *** no change -> nothing to do
 97        message s208(00) with 'No change - No update necessary'.
 98        return.
 99      else.
100        flights = p_flights.
101        me->save_data( ).
102        message s208(00) with 'Saved changes successfully'.
103      endif.
104    endmethod.                        "store_flights
105    method get_flights.
106      p_flights = flights.
```

VALUE = Standard Table[152x14(80)]
TYPE = Standard Table[152x14(80)]

Scope: \CLASS lcl_flights_manager\METHOD store_flights\IF

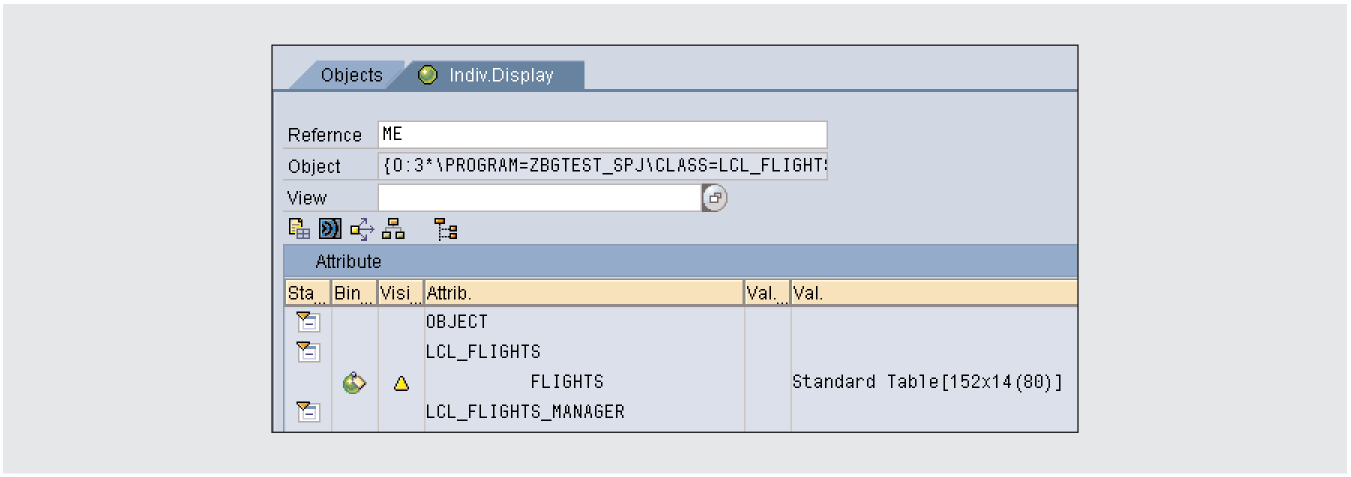
Local Variables and Parameters

V...	St...	Variable Name	V...	Val.	Technical Type	Hexadecimal
		P_FLIGHTS		Standard Table[152x14(80)	Standard Table[152x14(8	
		ME		{O:3*\PROGRAM=ZBGTEST_SPJ	Reference	

Figure 19 Setting a breakpoint to analyze variable p_flights

expected the message `'No change – No update necessary'` (line 97). Obviously we reached the wrong branch of the `IF` statement in line 95, because the comparison of the internal tables `p_flights` and `flights` found differences between them. So here is where we start our debugging session. Let’s set a session breakpoint in line 95 (`if p_flights = flights`) and reproduce the error.

Reaching the breakpoint in the debugger, we can see in the Local Variables and Parameters section of the Variable Fast Display (see **Figure 19**) that `p_flights`



Refernce ME
Object {O:3*\PROGRAM=ZBGTEST_SPJ\CLASS=LCL_FLIGHT
View

Attribute

Sta	Bin	Visi	Attrib.	Val.	Val.
			OBJECT		
			LCL_FLIGHTS		
			FLIGHTS		Standard Table[152x14(80)]
			LCL_FLIGHTS_MANAGER		

Figure 20 Displaying the attributes of the ME object

is an importing parameter of the `store_flights` method and that the `p_flights` parameter contains 152 lines. Moving the cursor over the `flights` variable in line 95 of the source code displays a Data Quick Info pop-up that also reveals that this table contains 152 lines. But what kind of variable is `flights`?

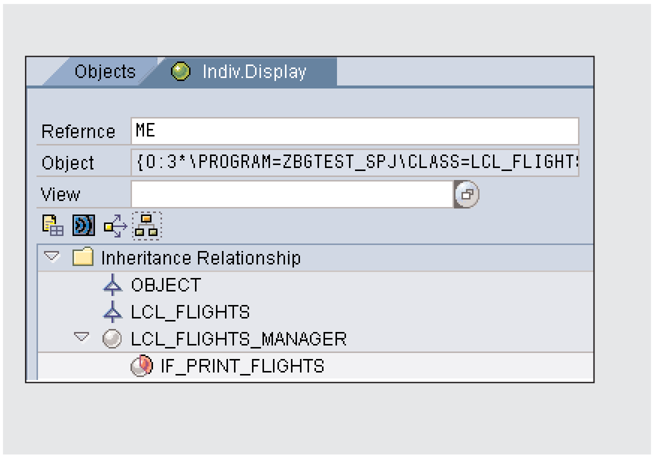
Double-clicking on the local variable `ME`, which is the object instance in which we are currently running, leads us directly to the appropriate detail view — the Object View — as shown in **Figure 20**.

Here we see that the internal table `FLIGHTS` is a protected instance attribute of the local class `LCL_FLIGHTS`, which is the superclass of the local class `LCL_FLIGHTS_MANAGER`, which is the runtime type of the `ME` local variable.

Let's check out the inheritance relationship of the local class `LCL_FLIGHTS_MANAGER`, which is also available in the Object View. As shown in **Figure 21**, `LCL_FLIGHTS_MANAGER` inherits from `LCL_FLIGHTS` and implements the interface `IF_PRINT_FLIGHTS`, which prints out the table of flights.

Ok, let's summarize:

- The flights that we want to save in the SAP database are stored in the internal table `p_flights`.
- The internal table `p_flights` is an importing parameter of the public method `store_flights`.
- Before the content of the importing parameter is



Refernce ME
Object {O:3*\PROGRAM=ZBGTEST_SPJ\CLASS=LCL_FLIGHT
View

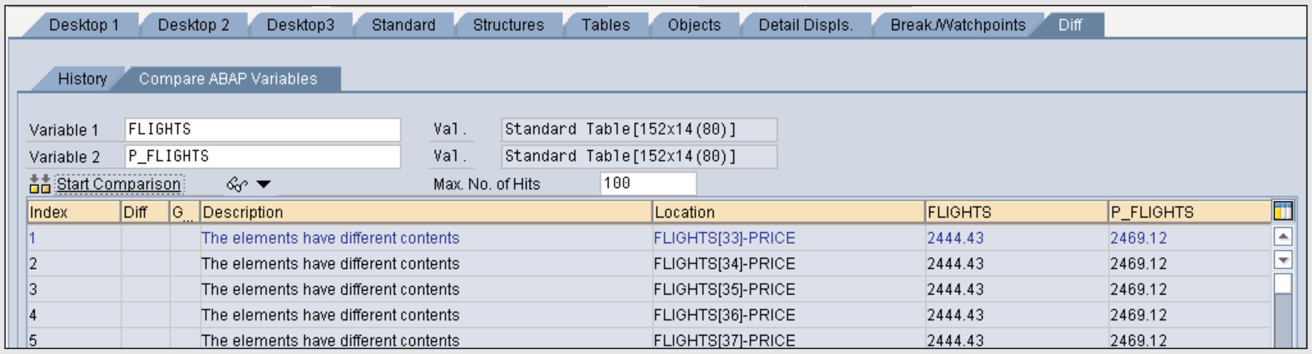
```
Inheritance Relationship
    OBJECT
    LCL_FLIGHTS
  LCL_FLIGHTS_MANAGER
      IF_PRINT_FLIGHTS
```

Figure 21 Displaying the inheritance relationship of an object.

saved on the database, the method `store_flights` compares the internal table `p_flights` with the original data stored in its protected instance attribute, the internal table `flights`.

- This comparison delivers differences.

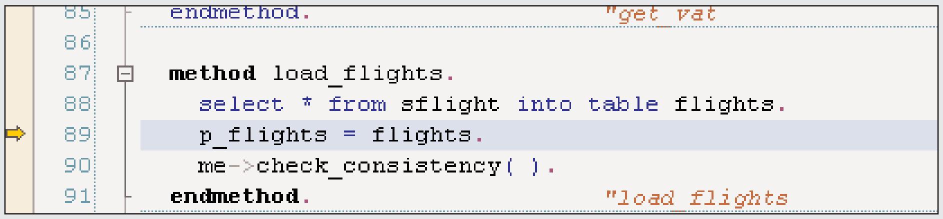
Our next step in the debugging session is to find these differences between the two tables. Using the Tables desktop, we can easily create two table views in order to compare these two tables. However, for large internal tables, finding out the differences within an acceptable period of time is not an easy task.



Variable 1	FLIGHTS	Val.	Standard Table[152x14(80)]
Variable 2	P_FLIGHTS	Val.	Standard Table[152x14(80)]
		Max. No. of Hits	100

Index	Diff	G...	Description	Location	FLIGHTS	P_FLIGHTS
1			The elements have different contents	FLIGHTS[33]-PRICE	2444.43	2469.12
2			The elements have different contents	FLIGHTS[34]-PRICE	2444.43	2469.12
3			The elements have different contents	FLIGHTS[35]-PRICE	2444.43	2469.12
4			The elements have different contents	FLIGHTS[36]-PRICE	2444.43	2469.12
5			The elements have different contents	FLIGHTS[37]-PRICE	2444.43	2469.12

Figure 22 Using the DIFF Tool to find differences between two large tables



```
85    endmethod.                              "get_vat
86
87    method load_flights.
88      select * from sflight into table flights.
89      p_flights = flights.
90      me->check_consistency( ).
91    endmethod.                              "load_flights
```

Figure 23 Stopping at the watchpoint

Therefore, the New ABAP Debugger provides a special tool for this task — the DIFF Tool, which finds the differences between two arbitrary ABAP data objects. You can compare internal tables, structures, strings, or even objects using the DIFF Tool, which provides the differences concerning the types (e.g., one table is a sorted table and the other a hashed table), and of course, the values.

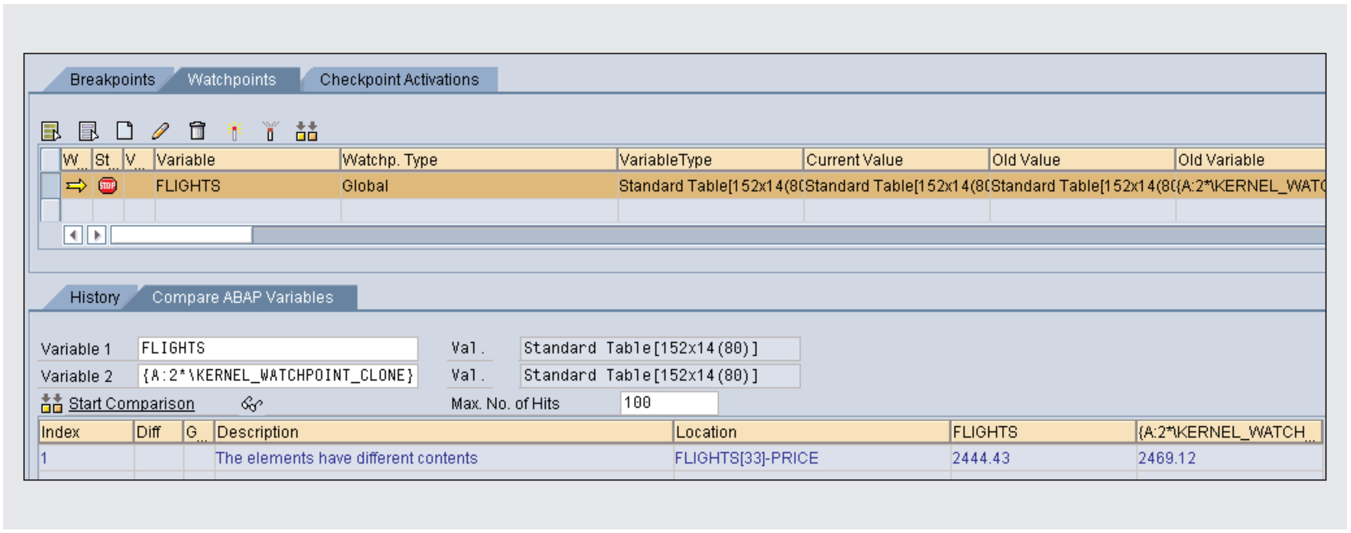
To start the DIFF Tool for the internal tables `p_flights` and `flights`, simply click on the Start Comparison button (see **Figure 22**), which displays a list of all differences between the two tables. As indicated in the display, the price is different for most of the table lines. This is strange, because we do not remember having the price changed in the original data shown in the transaction that we started earlier.

So, next we need to find out where the `flights` table is filled and where it is changed afterward.

This is the perfect task for a watchpoint. We restart the application again in debug mode and step to the `constructor` of the local class `lcl_flights_manager`. Here we set a watchpoint on internal table `flights`. This will allow us to keep track of all value changes that take place in the `flights` table.

Now we continue with the application by pressing F8, after which we stop because our `flights` watchpoint has been reached (line 89), as shown in **Figure 23**.

The reason that we stop at this point is easy to see — it's the initial filling of the `flights` table with the original data from the `sflights` database table via a `SELECT` statement. Pressing F8 to continue, we stop again. This time we'll switch to the Break/Watchpoints desktop, and then select the Watchpoints tab, where we find our reached watchpoint marked (see **Figure 24**). As you can see, the New ABAP Debugger creates a "clone" of the variable that is

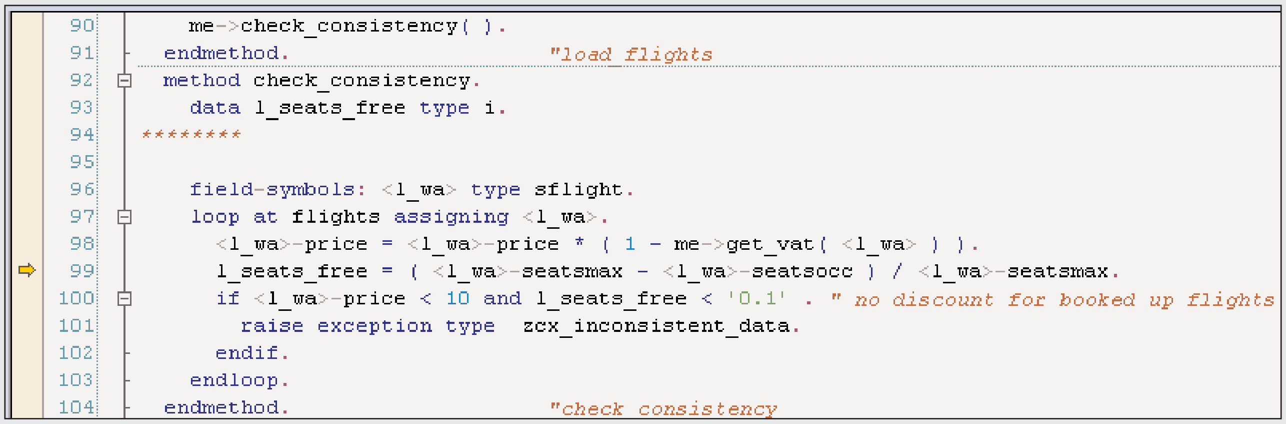


W...	St...	V...	Variable	Watchp. Type	VariableType	Current Value	Old Value	Old Variable
⇨	STOP		FLIGHTS	Global	Standard Table[152x14(8(	Standard Table[152x14(8(	Standard Table[152x14(8(	{A:2*\KERNEL_WAT(

Variable 1	FLIGHTS	Val.	Standard Table[152x14(80)]
Variable 2	{A:2*\KERNEL_WATCHPOINT_CLONE}	Val.	Standard Table[152x14(80)]
		Max. No. of Hits	100

Index	Diff	G...	Description	Location	FLIGHTS	{A:2*\KERNEL_WATCH...
1			The elements have different contents	FLIGHTS[33]-PRICE	2444.43	2469.12

Figure 24 Using the DIFF Tool to compare the current watchpoint variable with the old version



```
      me->check_consistency( ).
    endmethod.                    "load_flights
    method check_consistency.
      data l_seats_free type i.
********

      field-symbols: <l_wa> type sflight.
      loop at flights assigning <l_wa>.
        <l_wa>-price = <l_wa>-price * ( 1 - me->get_vat( <l_wa> ) ).
        l_seats_free = ( <l_wa>-seatsmax - <l_wa>-seatsocc ) / <l_wa>-seatsmax.
        if <l_wa>-price < 10 and l_seats_free < '0.1' . " no discount for booked up flights
          raise exception type  zcx_inconsistent_data.
        endif.
      endloop.
    endmethod.                    "check_consistency
```

Figure 25 Source code position where internal table FLIGHTS has changed again

watched. Therefore, you can always compare the old value of the watchpoint variable with the current value of the watchpoint variable. For complex data objects like our internal tables, we can use the DIFF Tool to compare the `flights` table with its former value using the watchpoint clone.

Figure 25 shows the output from the DIFF Tool: The price in line 33 (under the Location column in Figure 24) of the internal table `FLIGHTS` has changed.

To see where the table has changed again, switch back to the Source Code display tool on the standard desktop.

In the method `check_consistency`, we have a transaction that makes a consistency check to avoid entries with discount prices (gross price minus VAT) of less than $10 for flights with less than 10% free seats (see the comment `"no discount for booked up flights`). But by accident this code changes the original internal table `flights`. Look at line 98,

```
<l_wa>-price = <l_wa>-price * ( 1 –
me->get_vat( <l_wa> ) ).
```

where the field symbol `<l_wa>-price` is used as a temporary field to keep the value from the calculation of the "gross price minus VAT," which will be used later in the `IF` statement. But this field symbol points directly into the internal table `flights`, as indicated in the `loop ... assigning` statement shown in line 97 in Figure 25. As a consequence, the original value of `price` inside the `flights` table is changed implicitly by moving a new value to `<l_wa>-price`.

So, while the loop statement is being executed, the values for component `price` for all lines of the internal table `flights` are changed unintentionally. So now we know how to correct the coding: Simply use a local variable, replacing all `write` access occurrences of `<l_wa>-price`.

Unfortunately, we have reached the end of our New ABAP Debugger journey — but we hope that you feel that it has been worth the trip and that you now have a better understanding of this new tool. And if you want to further your understanding, you can test drive the latest version of SAP NetWeaver, which includes the New ABAP Debugger, by downloading the Sneak Preview of SAP NetWeaver at https://www.sdn.sap.com/irj/sdn/downloads.

Conclusion

As you have learned in this article, with the New ABAP debugger you get all the capabilities of the Classic ABAP Debugger but with a current and very flexible user interface and lots of new features. For example, in the Object View you can now easily access and analyze the inheritance hierarchy, which was not available until now. In the new Variable Fast Display, there is additional information available about all global and all local variables along with the procedure parameters. You can now view the source code with a new ABAP edit control that provides syntax highlighting and displays variable values in a data tooltip directly in the source code.

Finally, you benefit from debugging capabilities not available in debuggers of other languages. For example, you have the powerful forward navigation of variables that gives you detailed views of the different data types, watchpoints that allow you to compare the old and the new variable content, and a DIFF Tool that guides you directly through all differences between even huge and complex data types, including structures, internal tables, or even nested object graphs.

The New ABAP Debugger will save you many troubleshooting hours (or even complete weekends) so that you can instead enjoy your time — perhaps by reading *SAP Professional Journal* articles like this one.

Boris Gebhardt studied physics at the University of Erlangen-Nürnberg, Germany. He joined SAP AG in 1998, where he currently works in the ABAP QM group. Boris is responsible for customer support and SAP internal consulting for the ABAP programming language and its surrounding tools. He is also involved in the development of ABAP tools, including the New ABAP Debugger, and was engaged in a development project for the public sector.

Christoph Stöck studied chemistry at the Georg-August-University of Göttingen, Germany, and received his doctorate in physical chemistry at the Max Planck Institute of Flow Research, Göttingen. He joined SAP in 1996, where he became a member of the Business Programming Languages Group. As a development architect, Christoph is responsible for the arithmetic and conversion features of the ABAP language, the time service of the SAP application server, the time zone and the time stamp handling in the ABAP language, and many other fields of the ABAP processor. Currently, he is working on ABAP-related tools, especially the New ABAP Debugger.