BC403

ABAP Debugger

SAP NetWeaver

Date _____

Training Center _____

Instructors _____

_____

Education Website _____

Participant Handbook
Course Version: 92
Course Duration: 1 Day(s)
Material Number: 50103037

An SAP course - use it to learn, reference it for work

Internal Use SAP Partner Only

Copyright

Trademarks

Disclaimer

Internal Use SAP Partner Only

About This Handbook

This handbook is intended to complement the instructor-led presentation of this course, and serve as a source of reference. It is not suitable for self-study.

Typographic Conventions

American English is the standard used in this handbook. The following typographic conventions are also used.

Type Style	Description
Example text	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths, and options. Also used for cross-references to other documentation both internal and external.
Example text	Emphasized words or phrases in body text, titles of graphics, and tables
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example SELECT and INCLUDE.
`Example text`	Screen output. This includes file and directory names and their paths, messages, names of variables and parameters, and passages of the source text of a program.
`Example text`	Exact user entry. These are words and characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

Internal Use SAP Partner Only

Icons in Body Text

The following icons are used in this handbook.

Icon	Meaning
	For more information, tips, or background
	Note or further explanation of previous point
	Exception or caution
	Procedures
	Indicates that the item is displayed in the instructor's presentation.

Contents

Internal Use SAP Partner Only

Course Overview

This course discusses the functionality of the ABAP Debugger in detail.

Target Audience

This course is intended for the following audiences:

- Developers
- Consultants

Course Prerequisites

Required Knowledge

- SAPTEC (Fundamentals of SAP NetWeaver Application Server)
- BC400 (ABAP Workbench - Foundation)

Recommended Knowledge

- BC401 (ABAP Objects)

Course Goals

This course will prepare you to:

- Use the complete functionality of the ABAP Debugger

Course Objectives

After completing this course, you will be able to:

- Start and end a debugger session
- Configure the debugger layout
- Use all kinds breakpoints and watchpoints
- Explore data objects
- Analyze the memory consumption of data objects and programs
- Use all kinds of debugger tools
- Restrict debugging on predefined software layers
- Debug applications involving HTTP and RFC calls and user switches
- Create and use debugger scripts

Internal Use SAP Partner Only

SAP Software Component Information

The information in this course pertains to the following SAP Software Components and releases:

- SAP NetWeaver 7.0EhP 2

Internal Use SAP Partner Only

Unit 1

Debugger Functionality

Unit Overview

In this unit, the following aspects are covered: Starting and ending a debugger session, configuring the debugger layout, processing the source code, breakpoints and watchpoints, saving and loading debugger sessions, exploring data values, analyzing the memory consumption, setting debugger options and debugger settings, and using all kinds of debugger tools.

Unit Objectives

After completing this unit, you will be able to:

- Configure the layout of the ABAP Debugger
- Use breakpoints and watchpoints
- Save and load debugger session metadata
- Display and manipulate data objects
- Influence the program flow
- Analyze the memory consumption of variables and programs
- Analyze the programs loaded in the current internal session
- Explore screens and Web Dynpro controllers
- Analyze exceptions objects
- Start and stop SE30 and ST05 trace files from a debugger session
- Configure the ABAP Debugger
- Debug programs that access the database

Unit Contents

Lesson: Debugger - Basic Functionality

Lesson Overview

In this lesson you learn about the basic features of the ABAP Debugger. This includes the configuration of the layout, the usage of breakpoints and watchpoints, exploring the values of data objects - especially internal tables and objects, and the navigation in the source code.

The discussion is based on the new debugger, which is available as of SAP Net Weaver 2004.

Lesson Objectives

After completing this lesson, you will be able to:

- Configure the layout of the ABAP Debugger
- Use breakpoints and watchpoints
- Save and load debugger session metadata
- Display and manipulate data objects
- Influence the program flow

Business Example

You would like to understand how to use the ABAP Debugger to analyze existing programs. Two scenarios are of interest for you: First you want to understand the program flow. Second, you would like to find out the program module responsible for an unexpected program behavior.

Overview

The ABAP Debugger is the most important tool to analyze ABAP programs. During the program implementation the tool can be used to analyze error situations of the program (dumps, unexpected program behavior). For existing programs, the tool can also be used to understand the program flow. This is essential if you want to adapt or extend the program.

Two versions of the ABAP Debugger exist, the old debugger available in all SAP systems and the new debugger available as of SAP Net Weaver 2004. With the new debugger, the functionality of the ABAP Debugger has increased tremendously. For this reason, this lesson focusses on the new debugger.

Start / End a Debugger Session

There are several options to start an ABAP program and set it directly under the control of the ABAP Debugger. The implementation depends on the program type and of the tool used to display the program object (e.g. an executable program may be displayed by the ABAP Editor directly or by the Object Navigator embedding the ABAP Editor):

- Executable programs, function modules, methods, or transactions:
  - If the program is displayed in the object tree of the Object Navigator, the program can be debugged by selecting the corresponding context menu item (e.g. for an executable program, the menu item *Execute → Debugging*).
  - A menu item to start the program in the debugging mode can also be found in the leftmost SAP GUI menu. The name of the menu and the name of the menu item depends on the program kind (e.g. *Program → Test → Debugging* for an executable program).
- Background jobs:
  - Debugging is switched on by entering *jdbg* in the command line of the system function bar.
- ICF services:
  - To switch on debugging, the menu item *Edit → Debugging → Activate Debugging* has to be selected.

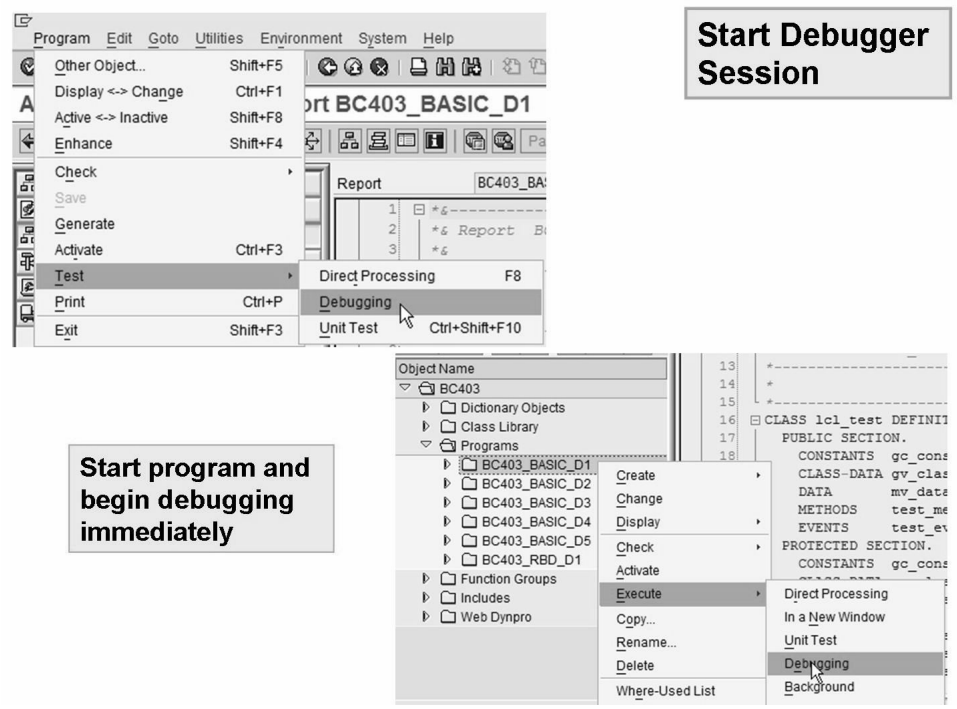


Figure 1: Start program in debugging mode

If a program is executed, debugging can be switched on as follows:

- Executable programs, function modules, methods, or transactions:
  - Enter ***/h*** in the command field or select the menu item *System → Utilities → Debug Screen*. ABAP statements and screen logic are debugged.
  - Enter ***/ha*** in the command field or select the menu item *System → Utilities → Debug ABAP*. The screen processing is not debugged. However, the modules called from PAI / PBO are debugged.
  - Enter ***/hs*** in the command field or select the menu item *System → Utilities → Debug System*. Not only your program, but also the system programs are debugged.
- All programs currently executed by work process:
  - In transaction SM50, the work process has to selected. Then the menu item *Administration → Program → Debugging* has to be selected. The executed program is interrupted at the next interruptible statement that is processed.

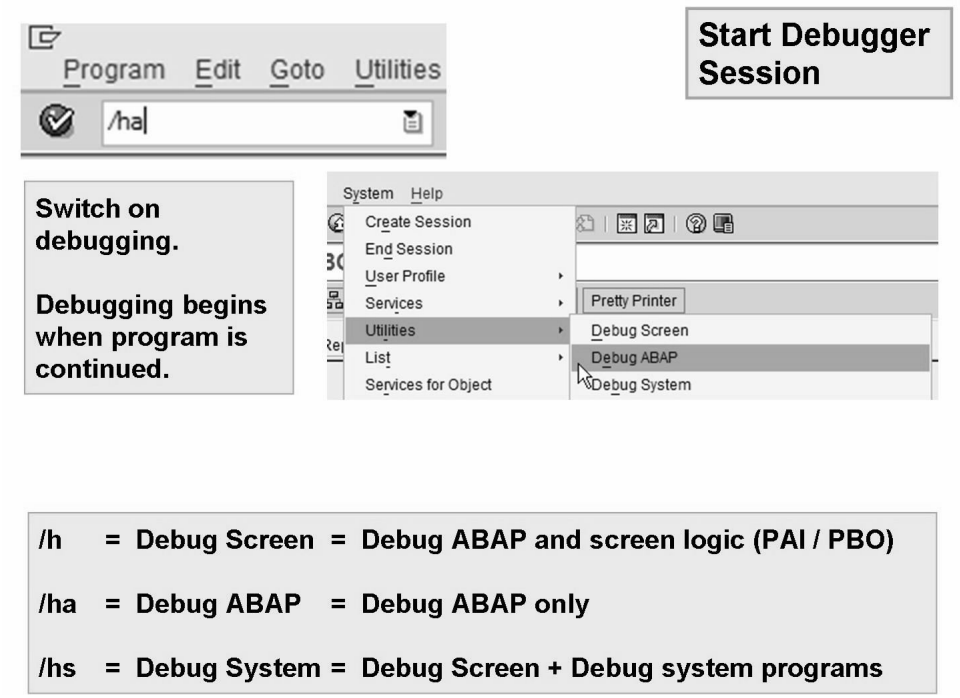


Figure 2: Switch on debugging

Finally, the processing of a program is stopped and the ABAP Debugger is displayed, if an appropriate breakpoint or watchpoint is reached. Details about breakpoints and watchpoints can be found below.

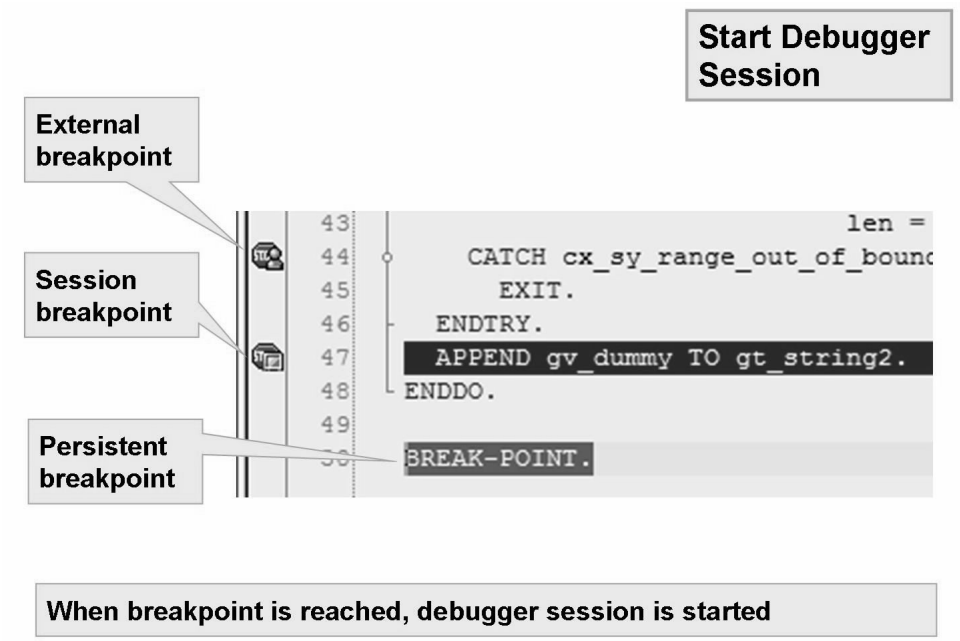


Figure 3: Switch on debugging - breakpoints

Each user can configure, if the old debugger or if the new debugger is to be used. This is done by selecting the menu item *Utilities* → *Settings* in the Object Navigator. Next, the tab *ABAP Editor* has to be selected. The tab page contains another tabstrip. Selecting the tab labeled *Debugging* opens the debugger configuration dialog. The debugger type is chosen by clicking the corresponding radio button in the group *ABAP Debugger*.

At runtime, the debugger type can also be toggled. In the leftmost menu (*Debugger*) a corresponding menu item can be found. Changing from the new debugger to the classical debugger is not always possible (e.g. if a field exit or if a Web Dynpro controller method is debugged).

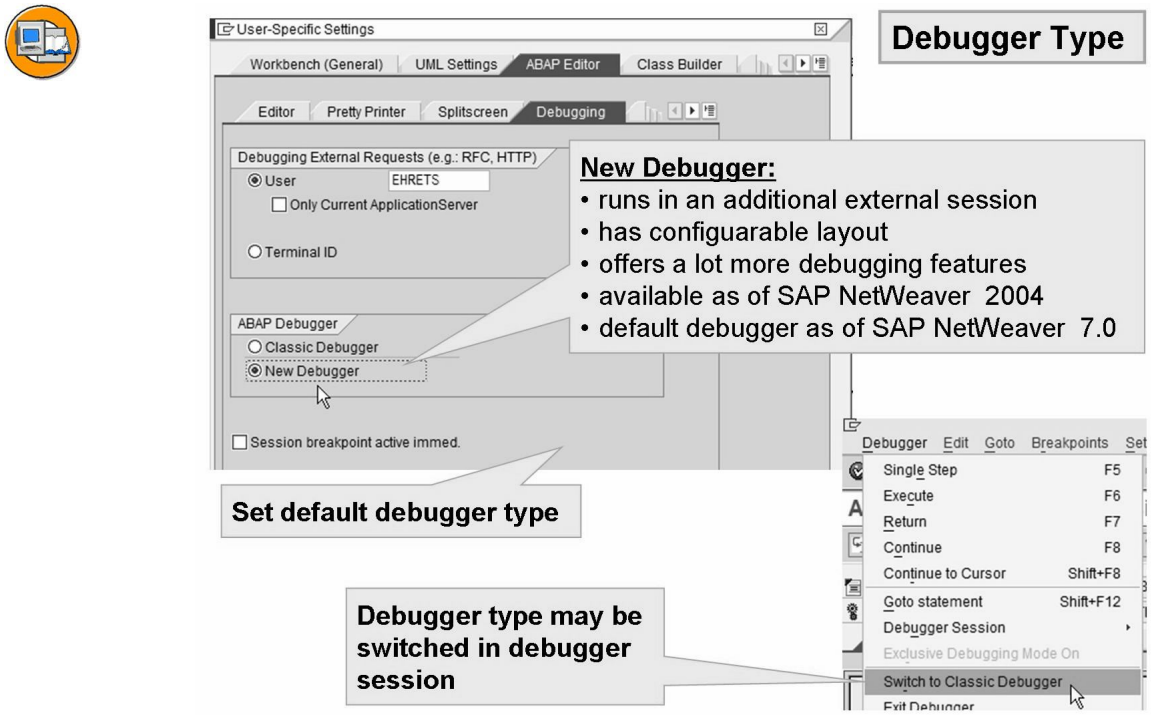


Figure 4: Choose debugger type

The new ABAP Debugger is executed in an extra external session, while the application to analyze (the "debuggee") occupies the original main session. This guarantees an interdependency between debugger and debuggee. Thus, as long as the debugger session is not ended, two dialogs are displayed, one containing the last dialog of the debuggee and one containing the debugger dialog. However, only one of the dialogs is active at the same time: Either the program waits for a user input (between two dialog steps), or the debugger waits for a user input (during a dialog step).

The debugger session may be finished by one of the following options:

- If the debugger dialog is active, the menu item *Debugger → Exit Debugger* can be selected. The debugger session is ended and the application is continued.
- If the debugger dialog is active, the menu item *Debugger → Exit Application and Debugger* can be selected. The debugger session is ended, In addition, the application is stopped.

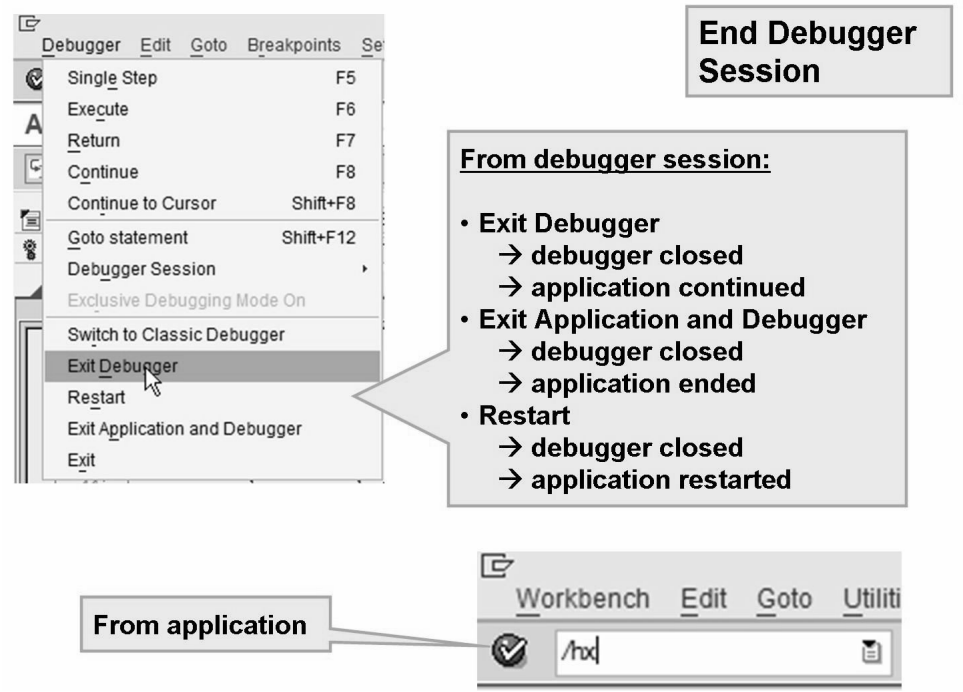


Figure 5: End debugger session

If the debugged application is ended, the debugger dialog may be closed or the dialog may stay open. In the latter case, the variable list is not cleared and can be reused in the next debugging session. To define this behavior, the menu item *Change Debugger Profile / Settings* has to be selected in the Debugger's *Settings* menu. In the dialog popping up, the checkbox labeled *Close Debugger After 'Continue' (F8) and Roll Area End* has to be set accordingly.

Note: For GUI based applications, the debugger dialog can be closed from the main session the debugger is attached to: Enter */hx* in the command field.

Configure the Debugger Layout

The debugger layout consists of a tabstrip, each tab page (desktop) displaying a different set of **debugger tools**. The tool set displayed on a certain tab page can be configured. A toolbar button allows each user to store the current layout setting. However, only the layout of the first three tab pages labeled *Desktop 1*, *Desktop2*, and *Desktop 3* can be saved. The layout changes on the remaining tab pages are dropped at the end of the debugging session.

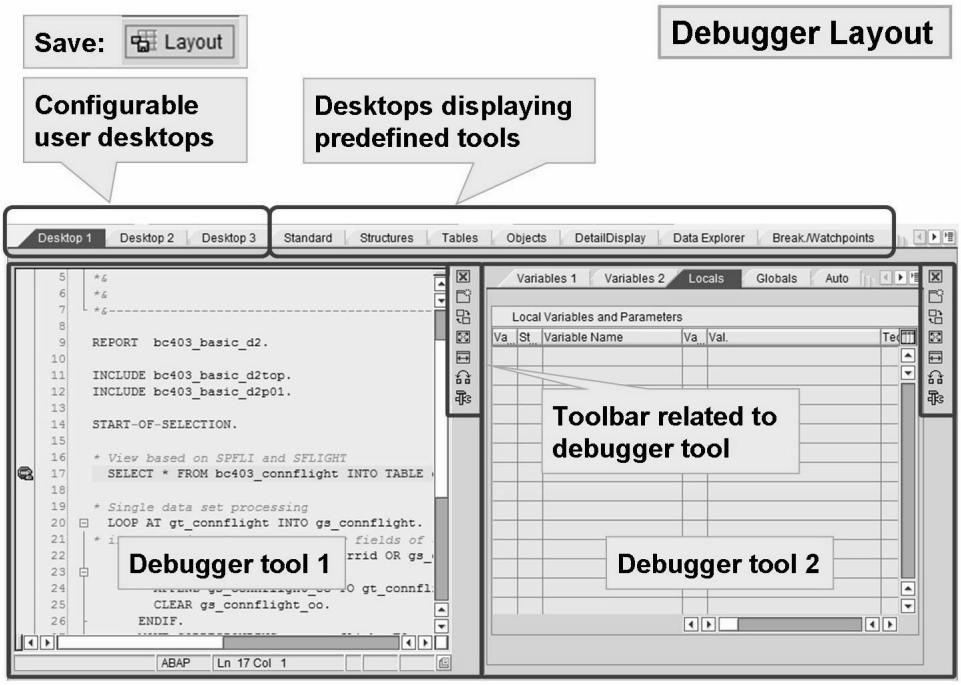


Figure 6: Debugger layout - overview

On each desktop, between 1 and 4 debugger tools may be displayed in parallel. A toolbar displayed right of each debugger tool on the desktop offers the following functionality:

- ***Close Tool*:** Remove the tool from the desktop. This icon is only available if more than 1 tool is displayed on the desktop.
- ***New Tool*:** Add a new tool to the desktop. Between 1 and 4 tools may be displayed on the same desktop.
- ***Replace Tool*:** Replace tool by another one.
- ***Full Screen*:** Remove all tools but the one related to this toolbar button.
- ***Maximize Vertically*:** Stretch tool vertically. The remaining tools on the same desktop are rearranged. If 4 tools are displayed, the tool in the same column is displaced.
- ***Maximize Horizontally*:** Stretch tool horizontally. The remaining tools on the same desktop are rearranged. If 4 tools are displayed, the tool in the same row is displaced.
- ***Swap*:** Change location of tool on desktop. If more than 2 tools are displayed, an additional dialog is displayed to define the associate for the swapped tool.
- ***Services*:** Functionality offered by debugger tool (not available for all tools).

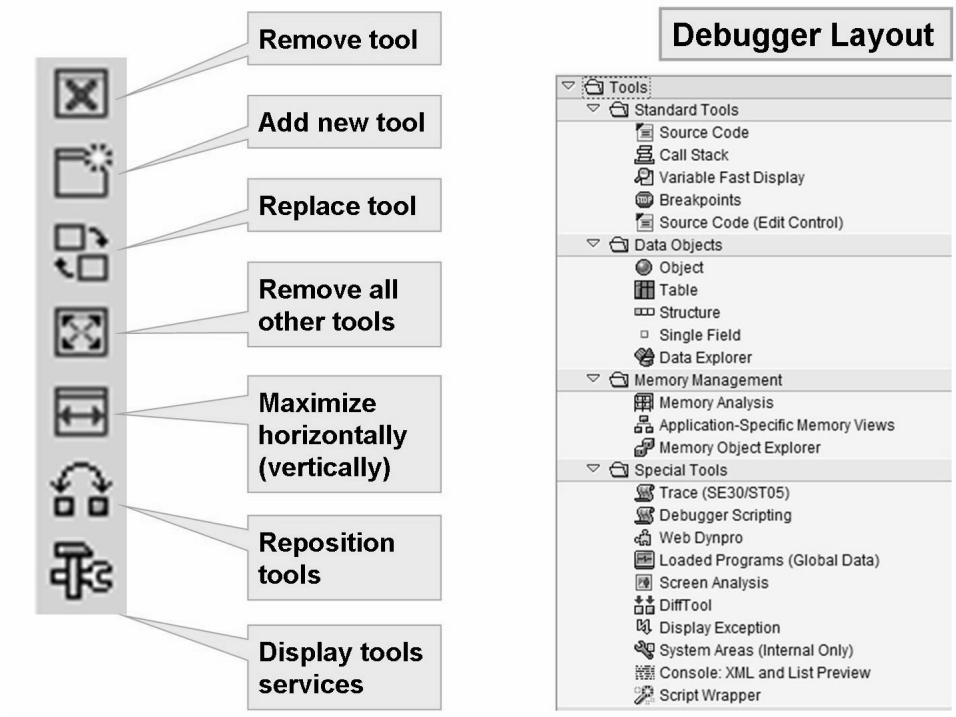


Figure 7: Configure debugger layout

As of SAP NetWeaver 7.0 EhP 2, the names displayed on the first three tabs can be changed. This is done by selecting the menu item *Debugger → Debugger Session → Designation of the User Desktop*.

To save the current layout, the toolbar button with the quick info text *Save layout* can be clicked. This user-specific layout is then loaded automatically when starting the next debugger session. This functionality can also be found in the menu *Debugger → Debugger Session*.

Process the Source Code

If the ABAP source code is displayed, a yellow arrow indicates the current processing position. The statement displayed in this line has not yet been processed. To step through the ABAP source code, the following options exist:

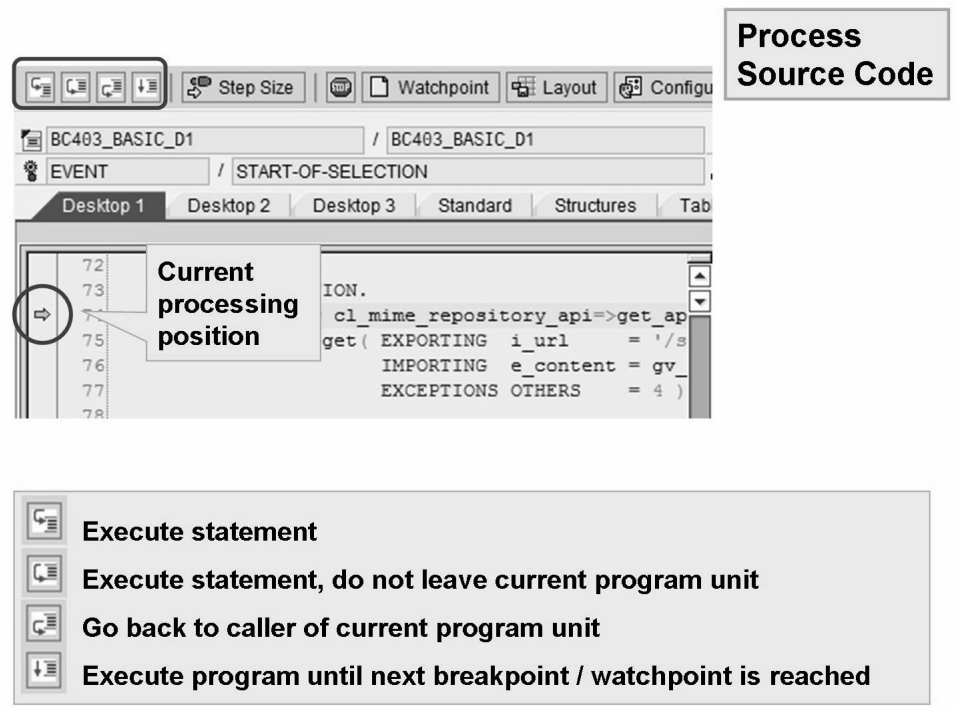


Figure 8: Process source code

- Process next statement, then stop again.
  - Press F5 on keyboard
  - Click toolbar button with quick info text *Single Step*
  - Select menu item *Debugger → Single Step*
- Process next statement, then stop again. This includes callable program unit (form, function module, method, module), which are executed completely before the debugger stops.
  - Press F6 on keyboard
  - Click toolbar button with quick info text *Execute*
  - Select menu item *Debugger → Execute*
- Return to caller of current callable program unit, then stop again.
  - Press F7 on keyboard
  - Click toolbar button with quick info text *Return*
  - Select menu item *Debugger → Return*
- Process source code until a breakpoint or a watchpoint is reached.
  - Press F8 on keyboard
  - Click toolbar button with quick info text *Continue*
  - Select menu item *Debugger → Continue*
- Process source code until cursor position.
  - Left mouse click statement, then press SHIFT+F8 on keyboard

Skip Source Code Section or Process Source Code multiple times

The ABAP debugger allows you to skip source code sections or to reprocess a coding section that has already been processed (if technically possible). Skipping a source code section may be used to omit processing a statement that would lead to a dump. Reprocessing a coding section multiple times may be used to check what happens for different variable values without having to restart the debuggee again.

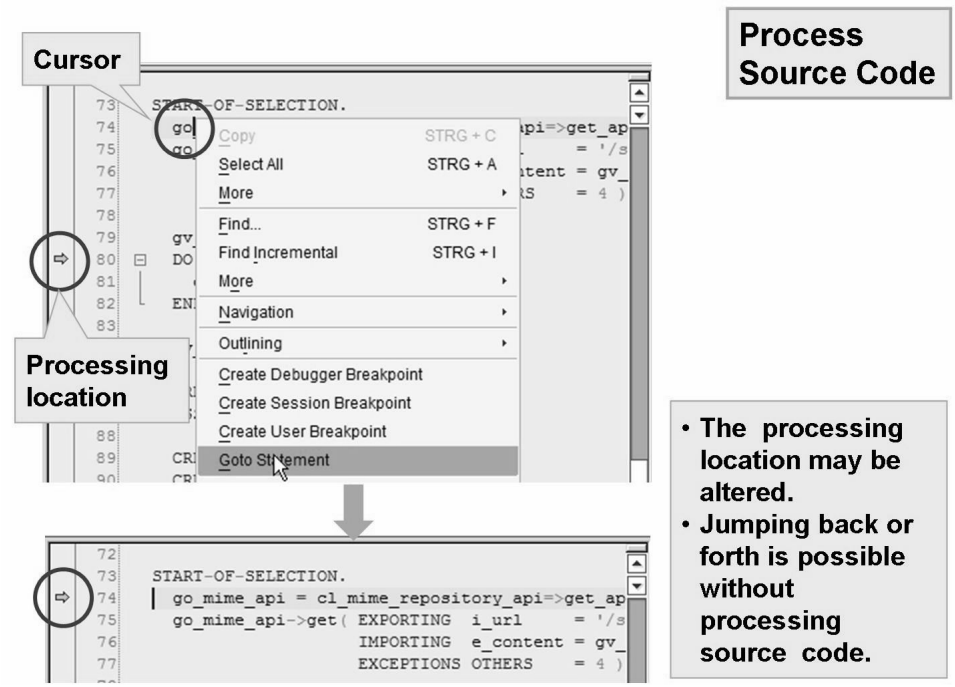


Figure 9: Alter processing location

To define which statement is to be processed next, the following options exist:

- Right mouse click statement and select context menu item *Goto Statement*.
- Left mouse click statement and select menu item *Debugger → Goto Statement*.

Display the Call Stack

The current call stack is displayed by the *Call Stack* debugger tool. By default, this tool is embedded on the second user desktop. The program entity displayed in row *n* is called by the program entity displayed in row *n+1*.

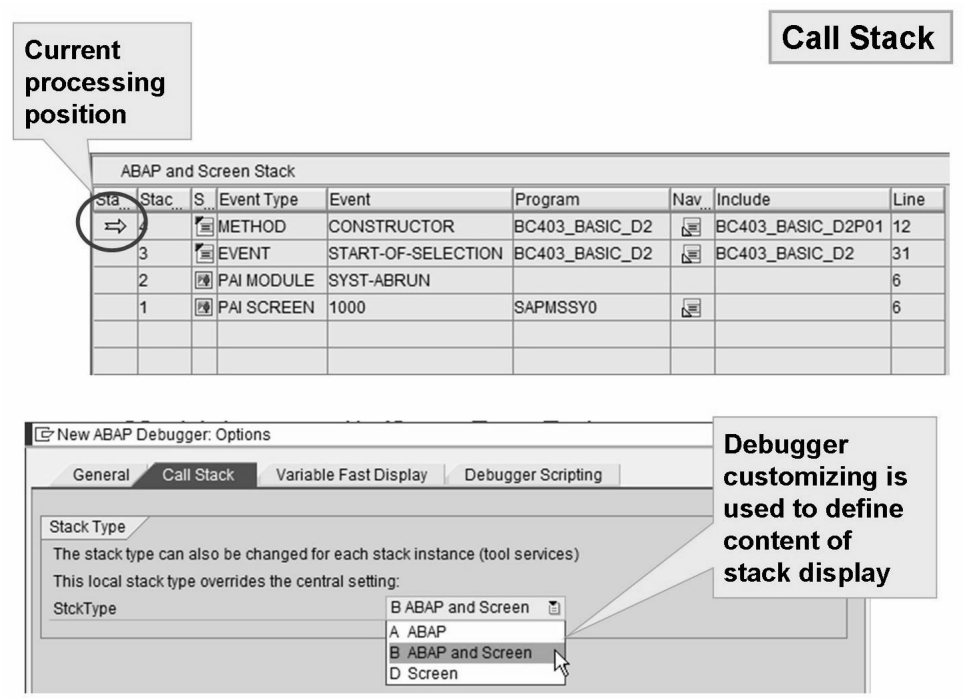


Figure 10: Source code processing - call stack

Any line displayed in the stack may be double-clicked. If the source code is displayed on the same desktop, the source code will be repositioned. The yellow arrow will now indicate the last statement of the selected stack level, which is responsible for creating the next stack level. However, this does not mean that the processing position is altered.

Note: Debugger customizing permits to define whether only programs, whether only screens, or whether programs and screens are displayed by the *Call Stack* debugger tool.

Breakpoints and Watchpoints

Different kinds of breakpoints allow the developer to mark positions in the source code. If a breakpoint is reached, the debugger stops, or - if it has not been displayed yet - the debugger is displayed in an extra main session. A watchpoint is related to a variable. It is used to stop processing the source code if the variable value changes in a predefined manner.

Note: Each user may define a maximum of 30 breakpoints.

Breakpoints

Breakpoints can have different scopes and different lifetimes.

Persistent breakpoints can be defined by adding a related statement to the source code of the program. The following options exist:

- `break-point` defines a persistent break point. This breakpoint is user-independent.
  - Addition `id = <checkpoint group name>` permits to define a breakpoint that may be activated / deactivated.
  - Addition `<n>` permits to index the breakpoint with an integer value.
- `break <user name>` is a macro. It defines a persistent and user-dependent break point.

Note: Hard coded breakpoints should only be used during the development process. They should be removed before the program object is transported.

Note: For program units called via HTTP or via RFC, persistent breakpoints are ignored as long as external debugging is not switched on.

If breakpoints should be related to the current user session, **session breakpoints** are the right choice. These breakpoints only work for the user who sets the breakpoints. If the user closes all main sessions of his/her user session (log off), the breakpoints are dropped. Session breakpoints may be set before debugging or while debugging:

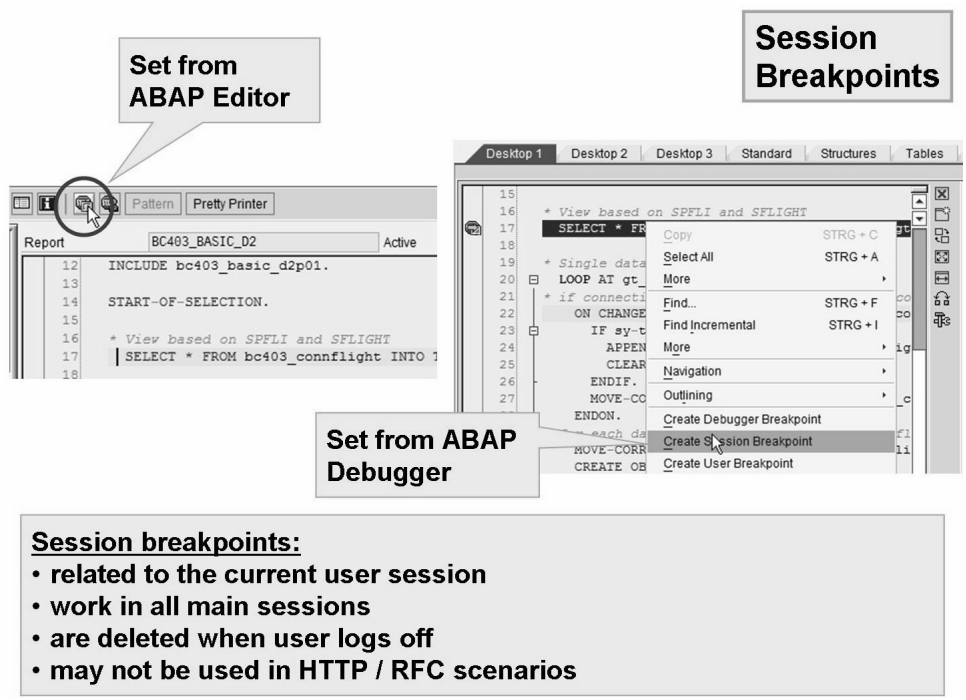


Figure 11: Define session breakpoints

- Define in ABAP Editor:
  - Left mouse click statement, then press *CTRL* + *SHIFT* + *F12* on keyboard.
  - Left mouse click statement, then click corresponding toolbar button displaying a stop sign and a blue screen.
  - Right mouse click in the column left of the source code and choose context menu item *Set Session Breakpoint*
- Define in the *Source Code* debugger tool:
  - Right mouse click the statement and choose context menu item *Create Session Breakpoint*

Note: Session breakpoints cannot be set in programs that may only be called via HTTP (e.g. Web Dynpro controller methods, BSP event handler methods).

External breakpoints permit to relate breakpoints to a certain user without relating it to a certain session. As of SAP NetWeaver 7.0 EhP 2, external breakpoints may also be assigned an identifier (terminal ID), which depends on the operating system user and on the client PC.

This kind of breakpoint has to be used to debug scenarios where the application runs in one session, while the debugger runs in another session. This is the case for all kinds of HTTP-based applications (Web service, Web Dynpro application, BSP application) or for distributed scenarios involving multiple systems. External breakpoints may be set before debugging or while debugging:

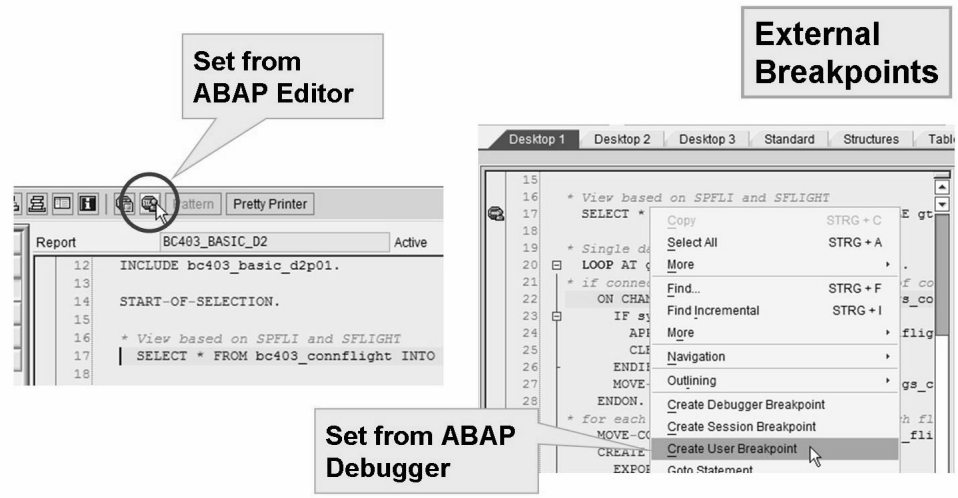


Figure 12: Define external breakpoints

- Define in ABAP Editor:
  - Left mouse click statement, then press *CTRL + SHIFT + F9* on keyboard.
  - Left mouse click statement, then click corresponding toolbar button displaying a stop sign and a person.
  - Right mouse click in the column left of the source code and choose context menu item *Set External Breakpoint*
- Define in the *Source Code* debugger tool:
  - Right mouse click the statement and choose context menu item *Create User Breakpoint*

Note: The lifetime of external breakpoints is restricted to 2 hours. If a new external breakpoint is set during this period, the lifetime of all external breakpoints is reset.

By default, user-dependent external breakpoints are related to the user who creates the breakpoints. However, there are situations where it is necessary to relate the breakpoint to another user (e.g. if the debuggee is started using an anonymous user). In this case, the user name has to be changed before the breakpoints is defined. This is possible for breakpoints set in the ABAP Editor by selecting the menu item *Utilities → Settings*. Next, the tab *ABAP Editor* has to be selected, followed by the tab labeled *Debugging*. The user can be entered in the field labeled *User*.

Note: The debugger is always started on the client where the breakpoint was defined before: If an external breakpoint is defined on client 1, the application is started on client 2, and the breakpoint is reached, the debuggee is blocked on client 2, while the debugger is started on client 1.

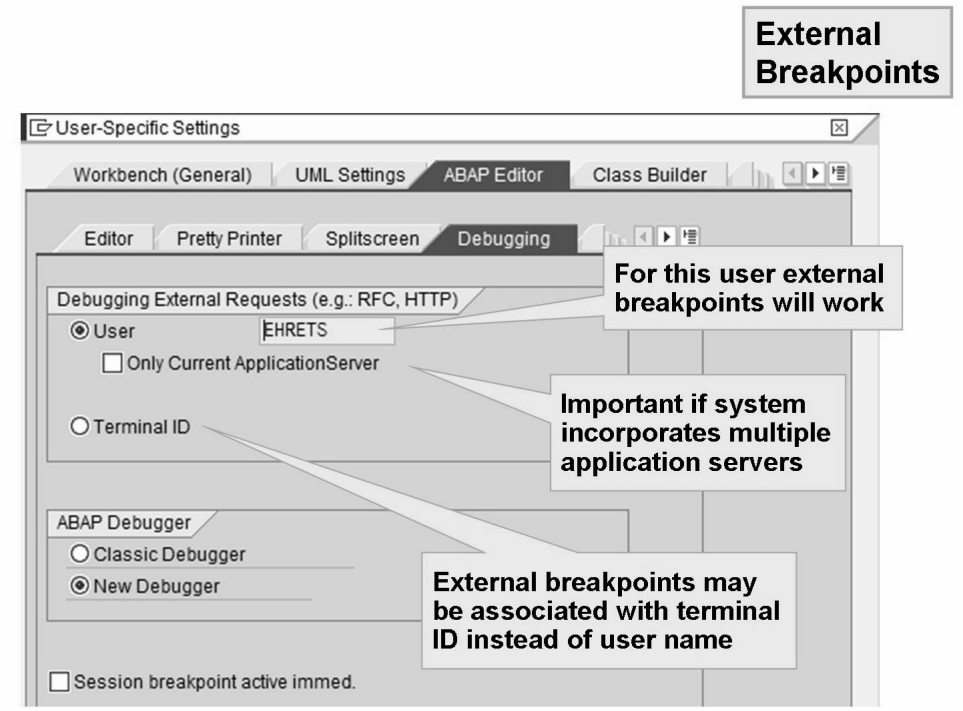


Figure 13: External debugging - user specific Workbench settings

As of SAP NetWeaver 7.0 EhP 2, user-dependent external breakpoints can be restricted to the current application server (checkbox labeled *Only current application server*).

Hint: The transaction **SRDEBUG** is used to define, whether user-dependent external breakpoints are defined on all servers belonging to a logon group, or whether they are defined on all application servers of the SAP system (default).

Finally, **debugger breakpoints** (dynamic breakpoints) can be defined in the debugger session. These breakpoints are only visible in this debugger session and they are dropped when the debugger session is finished. A debugger breakpoint may be defined as follows:

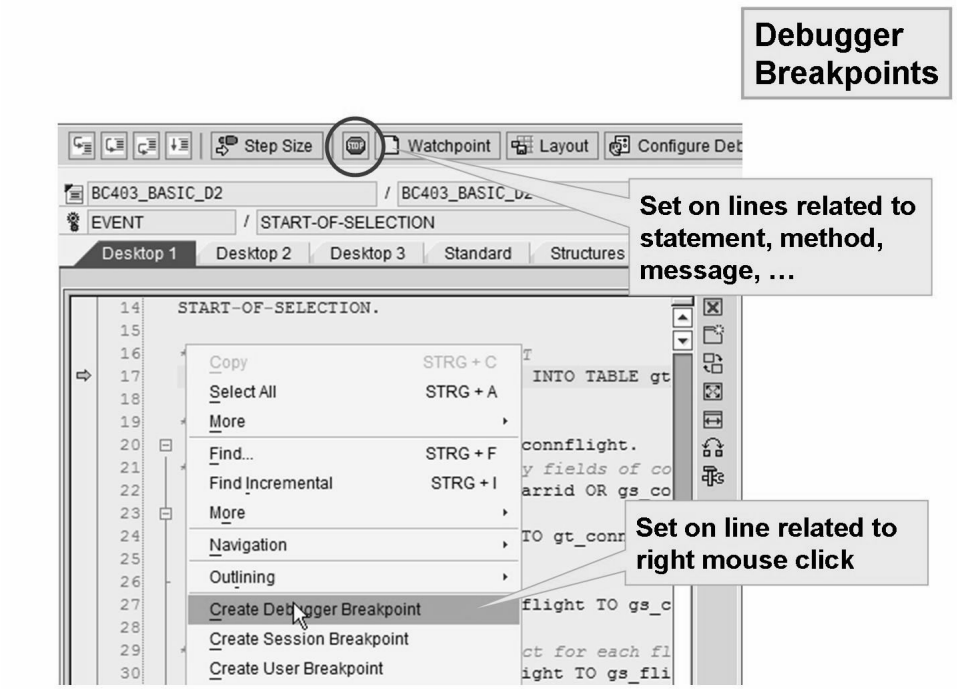


Figure 14: Define debugger breakpoints (1)

- Define at known position:
  - Right mouse click the statement and choose context menu item *Create Debugger Breakpoint*
  - Left mouse click in the column left of the source code
  - Left mouse click the statement and select menu item *Breakpoints → Line Breakpoint → Set/Delete*.
- Define at unknown position:
  - Press *F9* on keyboard.
  - Click toolbar button displaying a stop sign.
  - Select one of the menu items *Breakpoints → Breakpoint at.. → ....*

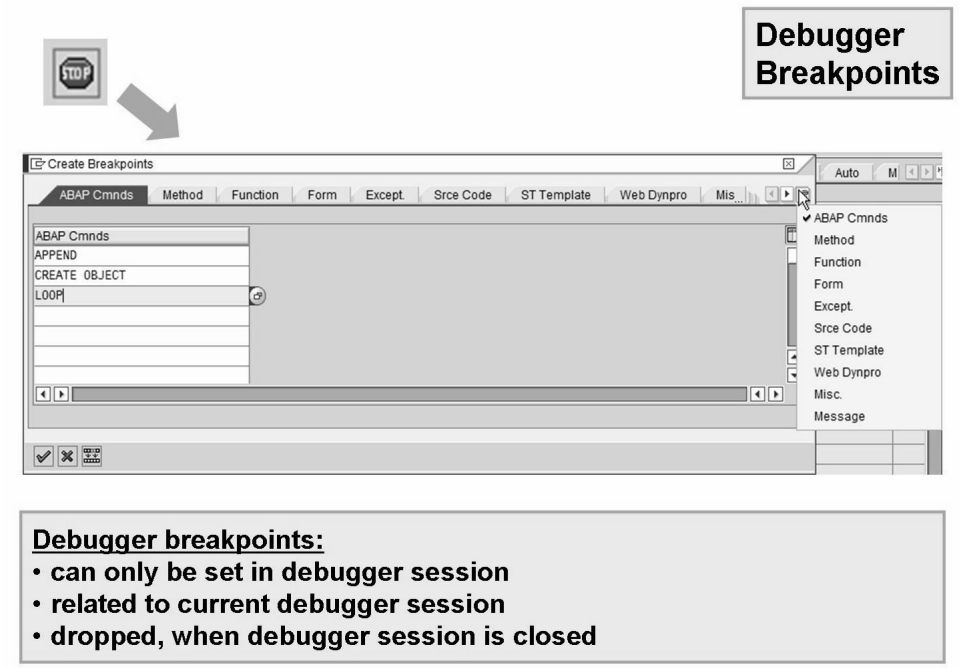


Figure 15: Define debugger breakpoints (2)

If F9 is pressed or if the toolbar button with the stop sign is clicked, an additional dialog pops up. Here, the user has to define requirements for setting the debugger breakpoints. Debugger breakpoint may be set for:

- **ABAP statements**
- **Methods of local class or global class**
- **Function modules**
- **Forms in own program or in submitted program**
- **Classical exceptions or class based exceptions**
- **Line numbers related to embedded include**
- **Line numbers related to PAI / PBO of used screen** (SAP NW 7.0 EhP 2)
- **Simple transformation templates** (SAP NW 7.0 EhP 2)
- **Methods of Web Dynpro controllers** (SAP NW 7.0 EhP 2)
- **Program stack changes** (SAP NW 7.0 EhP 2)
- **Unprecise decfloat computations** (SAP NW 7.0 EhP 2)
- **Defined messages (T100)** (SAP NW 7.0 EhP 2)

All breakpoints currently defined are listed on the tab labeled *Break./Watchpoints*, sub tab *Breakpoints*. On this screen, existing breakpoints may be deleted, activated, deactivated, or new debugger breakpoints may be created. In addition, the breakpoint type may be changed for each breakpoint. The breakpoint list can be refreshed. This reloads all breakpoints relevant for the current debugging session. Deleting, activating, or deactivating breakpoints is also possible from the

context menu related to the break point or from the menu *Breakpoints*. Reloading breakpoints or changing the type of debugger breakpoints can also be found in the menu *Breakpoints*.

In the Object Navigator, an overview over all existing session breakpoints and external breakpoints is displayed if the menu item *Utilities → Breakpoints → Display* is selected. In the dialog popping up, breakpoints can also be deleted.

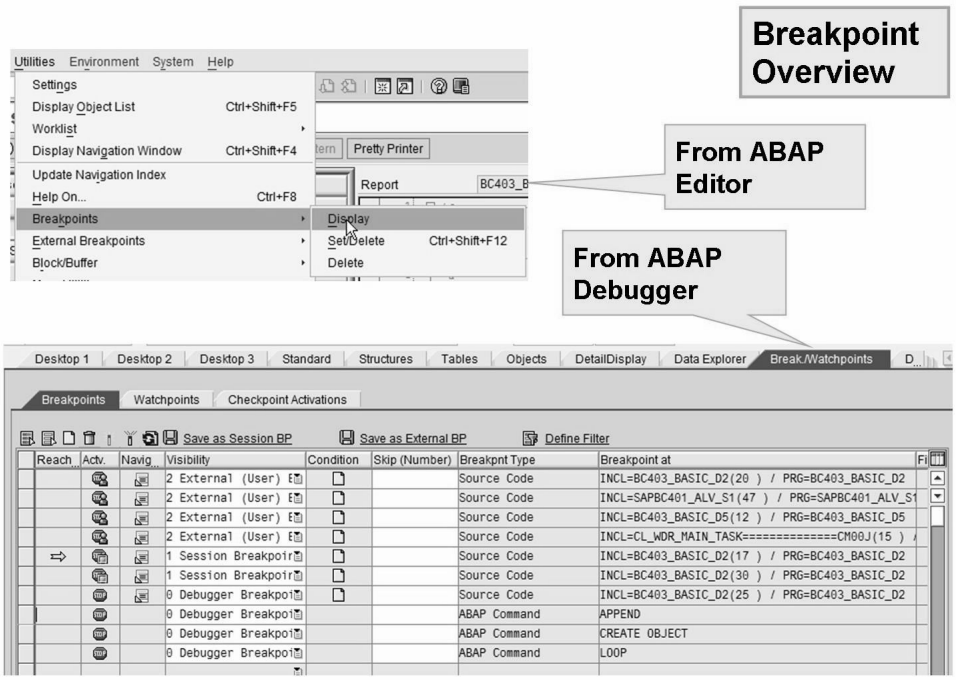


Figure 16: Overview over existing breakpoints

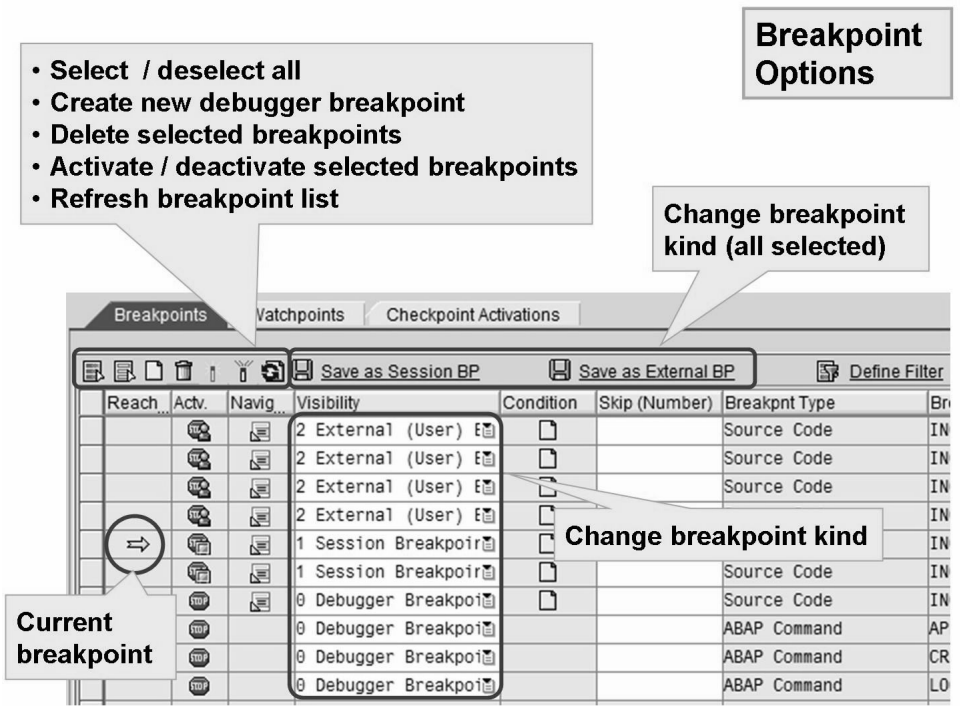


Figure 17: Breakpoint options

Note: If *Save (CTRL+S)* is selected in a debugger session, all debugger breakpoints are changed to session breakpoints.

As of SAP NetWeaver 7.0 EhP 2, each breakpoint can be assigned a logical condition. This allows the user to define **conditional breakpoints** without having to change the source code. The conditions are assigned to the breakpoint in the debugger session: Depending on the breakpoint kind, the condition is defined differently. For all breakpoints that are defined at a known source code line, the following options exist:

- In the *Source Code* debugger tool:
  - Right mouse click existing breakpoint and choose context menu item *Create Breakpoint Condition*
- When creating a debugger breakpoint:
  - Enter condition in field labeled *Free Condition Entry*
- In the *Breakpoints* debugger tool:
  - Click icon displayed in *Condition* column and enter condition in dialog popping up.

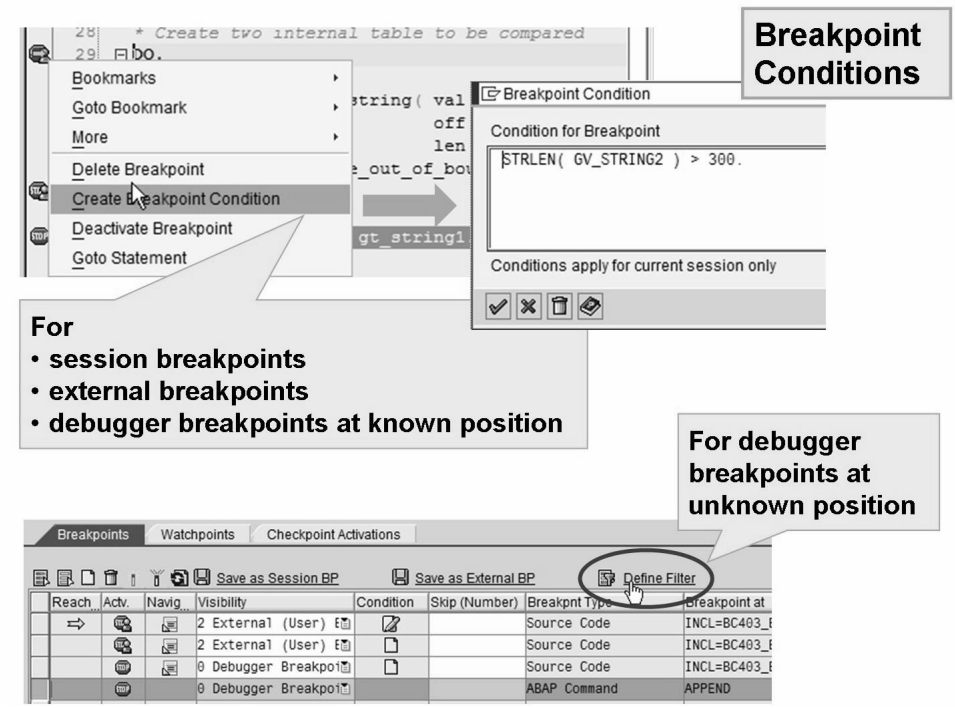


Figure 18: Define breakpoint conditions

For debugger breakpoints defined at an unknown source code line, conditions are defined in the *Breakpoints* debugger tool. Here, a link labeled *Define Filter* may be clicked to set boundary conditions.

Watchpoints

Watchpoints can only be defined in a debugger session. At the end of the debugger session watchpoints are always deleted.

To create a watchpoint, the following options exist:

- Press *SHIFT + F4* on keyboard.
- Click toolbar button displaying the text *Watchpoint*.
- Select menu item *Breakpoints → Create Watchpoint*.

In the dialog screen popping up, the name of the data object to observe has to be entered in the field labeled *Variable*. If the cursor has been positioned on a data object before the creation process is started, the name of the data object is automatically copied to this field.

For local variables a radio button group permits to restrict the monitoring to the current program unit instance (form, function module, method). Alternatively, the variable can be monitored for all instances of the program unit.

The logical condition related to the data object can be entered in the fields labeled *Free Condition Entry*. If this field is left empty, any change in respect to the variable pauses the program processing.

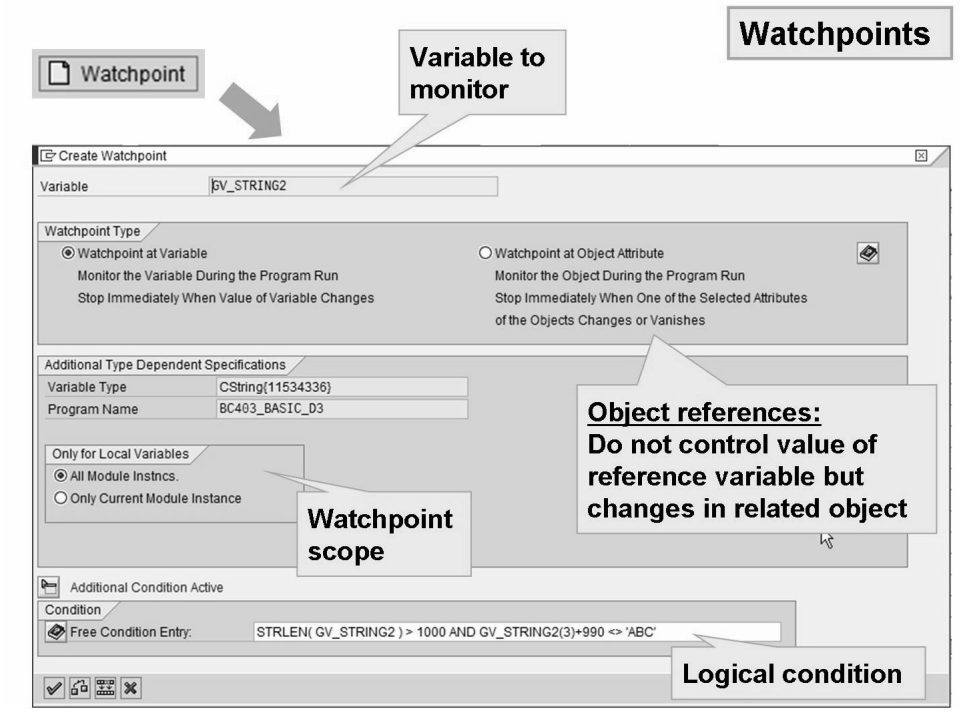


Figure 19: Define watchpoints

As of SAP NetWeaver 7.0 EhP 2, an additional radio button group exists, headed *Watchpoint Type*. This setting is very useful for monitoring changes in respect to objects. The reference variable pointing to an object is not altered if any object attribute value is changed. Thus a watchpoint on the data reference variable will not pause the program processing if an object attribute is changed. However, setting a watchpoint for each object attribute is cumbersome. The new setting *Watchpoint at Object Attribute* can be used to set a watchpoint for certain kinds of object attributes (e.g. all public instance attributes and all private static attributes).

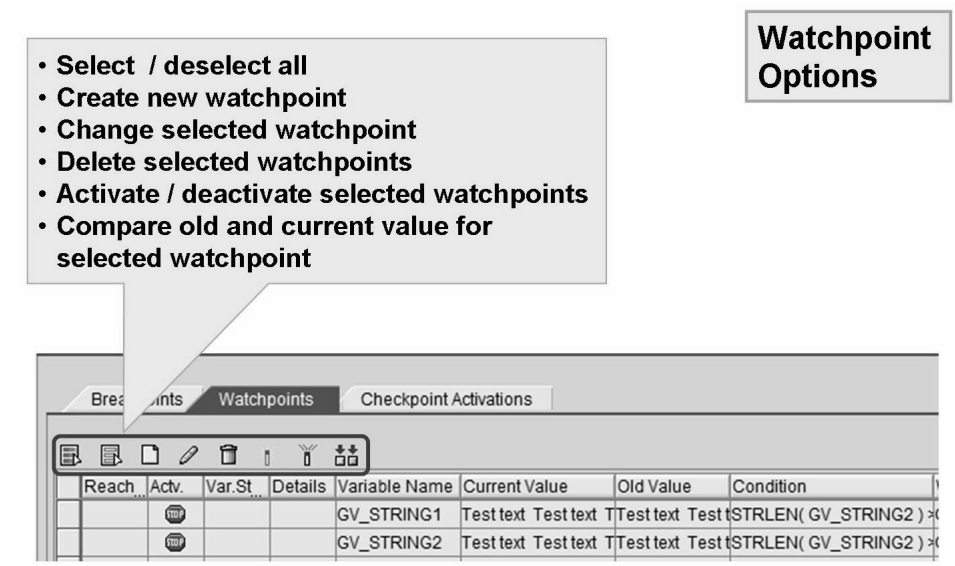


Figure 20: Watchpoint options

All watchpoints are listed on the tab labeled *Break./Watchpoints*, sub tab *Watchpoints*. On this screen, existing watchpoints may be deleted, activated, deactivated, or new watchpoints may be created. In addition, the current and the previous value of each variable is displayed. To compare these values, the rightmost toolbar button with the quick info text *Compare Variables* can be clicked. The two values are then analyzed on the tab labeled *Diff* using the *DiffTool* debugger tool. Here, the comparison can be continued.

If multiple watchpoints are defined, they are combined by a logical OR.

Save / Load Debugger Session

All non persistent breakpoints are deleted when the user logs of. This is inconvenient, if you want to continue with the program analysis later on. However, the debugger offers the functionality to save the current settings related to the debugger session. Later on these settings can be reloaded.

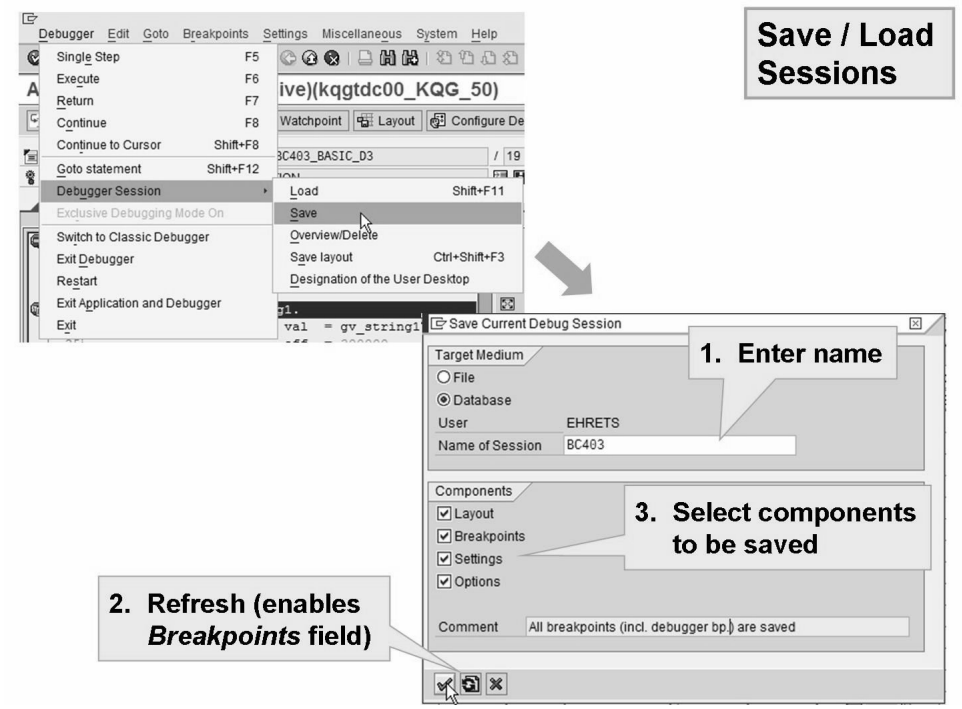


Figure 21: Save a debugger session

To **save** the settings of the current debugger session, the menu item *Debugger → Debugger Session → Save* has to be selected. The settings may be saved in the file system or on the database of the SAP system. The name of the session can be chosen arbitrarily. However, the name *START_UP* is reserved for the default debugger settings. These are loaded automatically at every start of a debugger session. After having entered the session name, the button *Refresh* (F5) has to be clicked. Next, the components to be saved have to be picked. Besides the current breakpoints (session, external and debugger breakpoints), the current layout, debugger options, and debugger settings may be stored.

Note: Breakpoints may not be saved for the default debugger session.

To **load** a debugger session previously stored, the menu item *Debugger → Debugger Session → Load* has to be selected. After having chosen the session name, the button *Refresh* (F5) has to be clicked. Next, the components to be loaded can be picked.

Note: All breakpoints loaded from a debugger session are defined as debugger breakpoints. Thus, these breakpoints are dropped when the debugger session is finished. However, debugger breakpoints may be converted to external breakpoints or to session breakpoints.

An overview over all stored debugger sessions is displayed if the menu item *Debugger → Debugger Session → Overview/Delete* is selected. In the list displaying the debugger sessions one or multiple lines can be marked and the related session data can be deleted.

Explore Data Object Values

Multiple debugger tools exist to display or change the value of data objects. A quick overview over all kinds of variables is provided by the *Variable Fast Display* debugger tool. Elementary fields may be further analyzed using the *Single Field* debugger tool. The *Structure* debugger tool is used to analyze structured variables, the *Table* debugger tool is used to analyze internal tables. The latter tool offers a huge functional range. Objects can be explored using the *Object* debugger tool. Finally, a hierarchical view on data objects is provided by the *Data Explorer* debugger tool.

If the cursor is positioned on the variable name in the *Source Code* debugger tool, a quick info pops up. This quick info displays the type of the variable. If the variable has a simple type, the variable value is also included in this quick info.

Variable Fast Display

To display multiple variable values in parallel, the *Variable Fast Display* debugger tool is used. By default, this tool is embedded on the first desktop. The variables are distributed across four tab pages labeled *Variables 1*, *Variables 2*, *Locals*, and *Globals*. All global variables are automatically added to the *Globals* tab. If a form, a function module, or a method is processed and if local variables are defined in this unit, these local variables are automatically added to the *Locals* tab. The content of theses two tabs is read only.

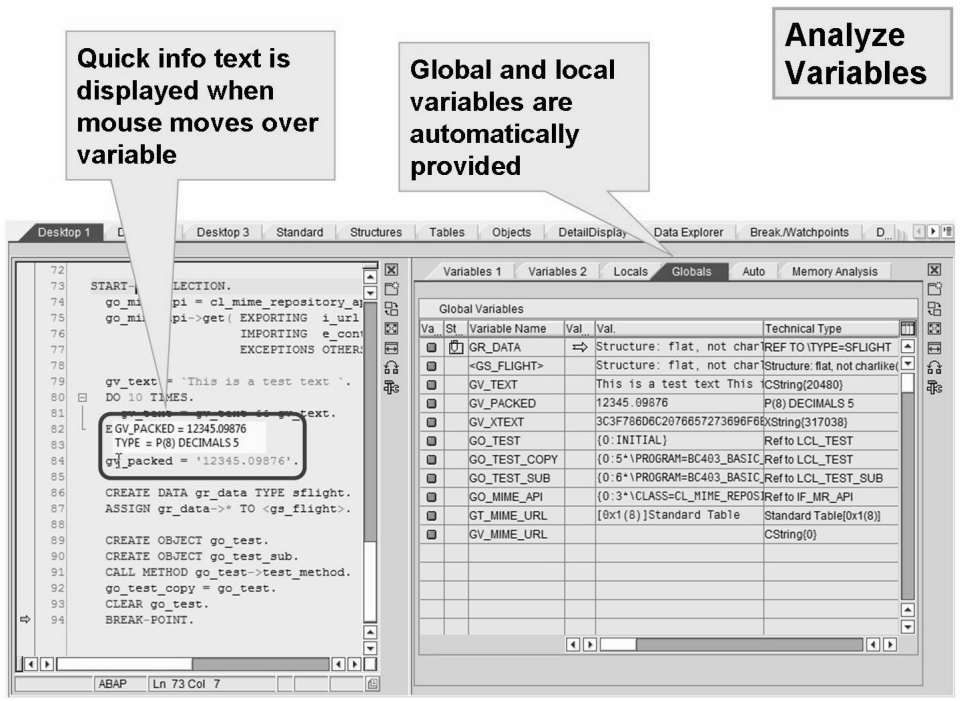


Figure 22: Explore variables

Variables may also be added to the tabs *Variables 1* and *Variables 2* by double clicking on the variable name in the *Source Code* debugger tool. The value of **simple variables** is displayed read only right of the variable name. If the authorization is sufficient, the variable value can be altered. This is done by clicking the icon right of the field value, changing the value, and confirming the new value by pressing *Enter*.

More complex variables have to be displayed in a tool suited for this type of variable. This is most easily done by double clicking the variable in the *Variable Fast Display* debugger tool.

Note: The tool, which is opened if a variable is double clicked in the *Variable Fast Display* debugger tool may be displayed on the same desktop, or on a special desktop only containing this tool. In addition, this tool may be a special tool for this kind of variable type (e.g. the *Structure* debugger tool) or it may be the *Data Explorer* debugger tool.

This navigation is guided by debugger customizing settings. In addition, customizing defines which of the four tab pages (*Variables 1*, *Variables 2*, *Locals*, or *Globals*) is active when the debugger session is started.

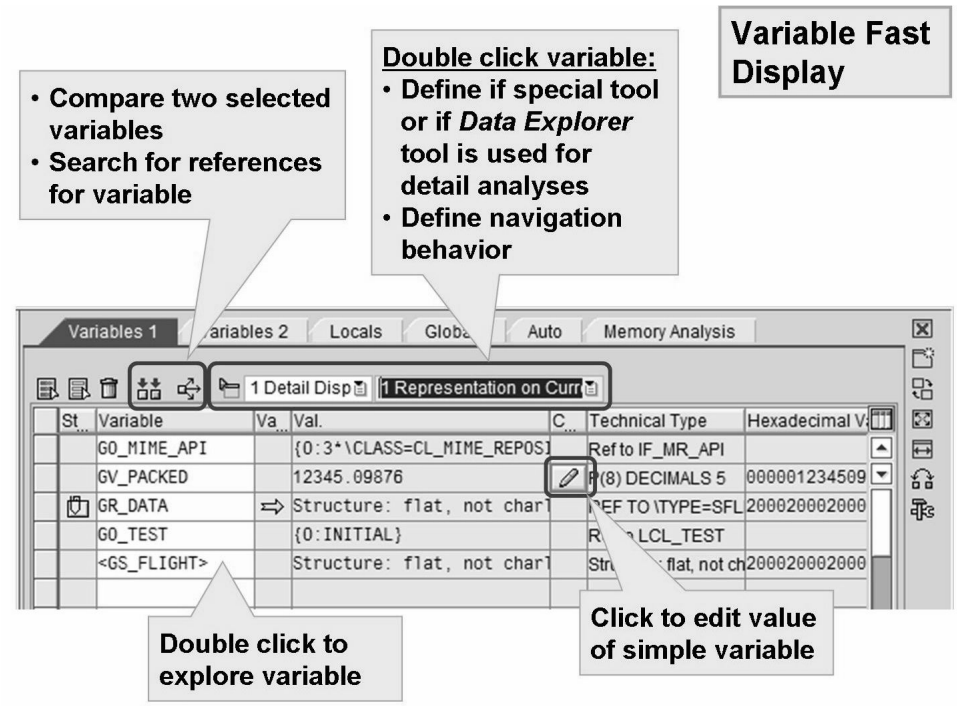


Figure 23: Variable Fast Display - functionality

The **tool services** include two important functions:

The information displayed on the current tab page can be stored as the content of a file in the file system. If the program unit currently debugged is a **function module**, the interface parameter values may be stored as **test data**. This test data may than be reused in the test environment of the function module.

As of SAP NetWeaver 7.0 EhP 2, additional functionality is available for variables displayed on the tab pages labeled *Variables 1* and *Variables 2*:

References on variables may be analyzed by clicking the icon labeled *Display References*. If the icon labeled *Suspend Navigation / Hide Adjustment* is clicked, two additional dropdown boxes are displayed in the toolbar. Initially, these fields display the customizing settings in respect to the tool navigation. By changing the

field values, the customizing settings may be overwritten. However, the persistent customizing is not changed this way. Thus, in the next debugger session, the initial customizing settings are used again.

Note: For dynamic variables (strings, xstrings, data reference variables, internal tables, objects), the memory allocation may change during runtime. Here, the variable data is referenced as follows:

A reference variable points to a memory area containing administrative information (header), while the header contains the information about the data location in memory. Both, reference and header can be explored in the ABAP Debugger:

- **<var>**: Variable data.
- ***<var>**: Header value (containing data location).
- **&<var>**: Reference value (containing header location).

Analyze Single Fields

The ***Single Field*** debugger tool displays details of a simple filed. The value and the data type is displayed below the field name. Depending on the data type, additional display options exist. Long literals may be broken into segments displayed as lines of a table. Instead of a text display, the hexadecimal representation may be selected. XML files may be displayed using the Microsoft Internet Explorer browser. Finally, an internal representation of a literal may be treated using different code pages.

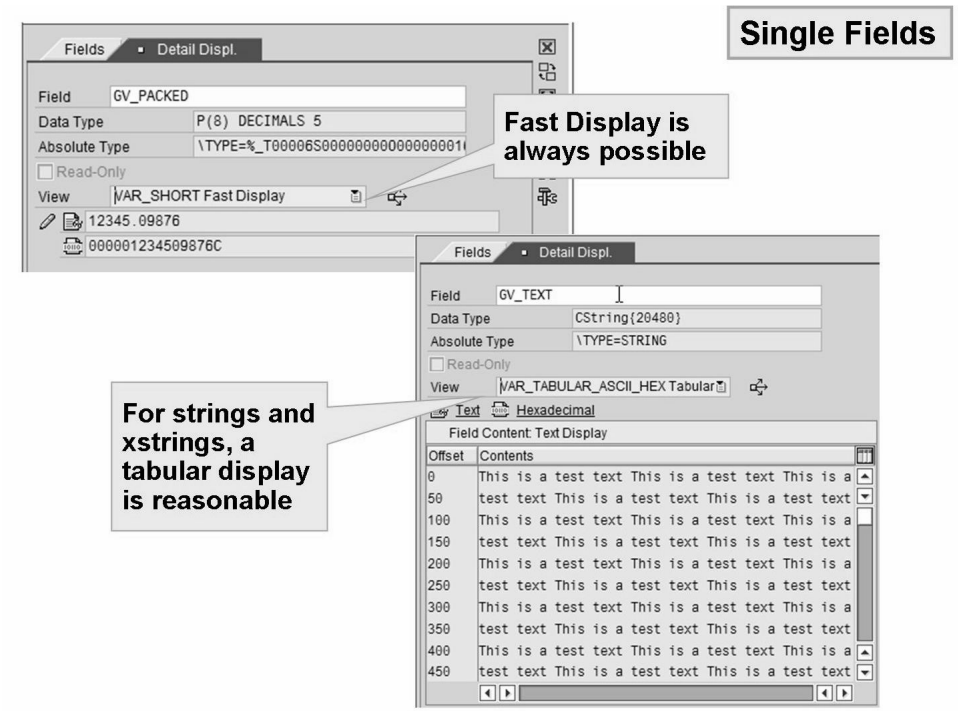


Figure 24: Explore single fields (1)

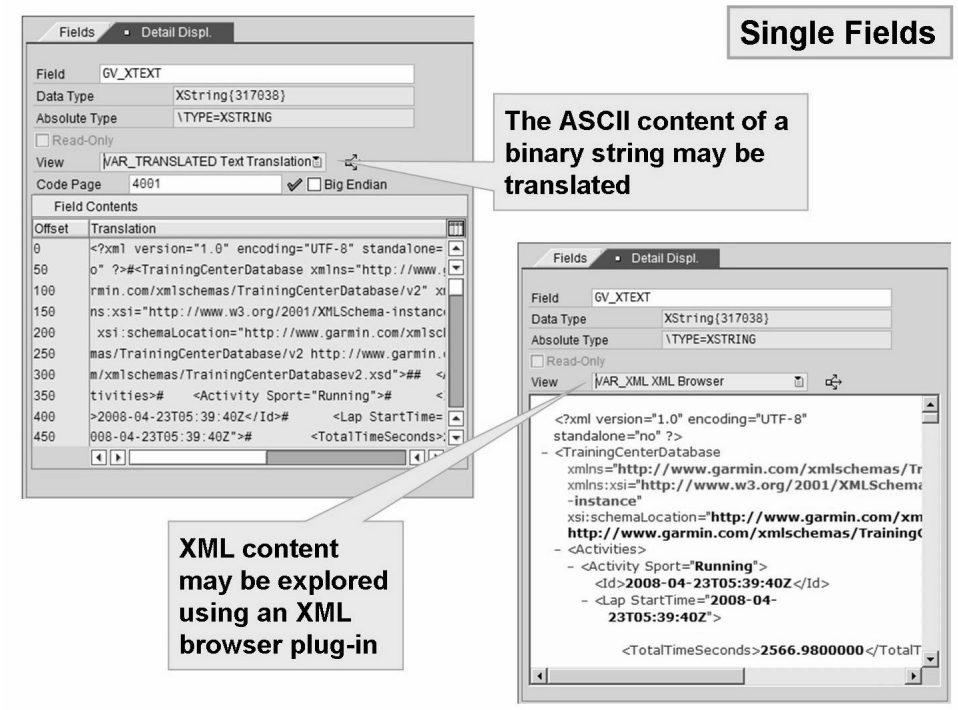


Figure 25: Explore single fields (2)

The **tool services** include the following important functions:

The field value can be stored as the content of a file in the file system. The browser display can be saved as a plain text document. As of SAP NetWeaver 7.0 EhP 2, the browser display can also be saved as an XML document or as an HTML file.

Analyze Structures

The ***Structure*** debugger tool displays the single components of the structure. If a component has a simple type, the value is also displayed. If the component is structured, the debugger allows the user to expand the embedded structure. If the structure component is neither simple nor structured, the component may be double clicked to open a tool suited to explore the structure component.

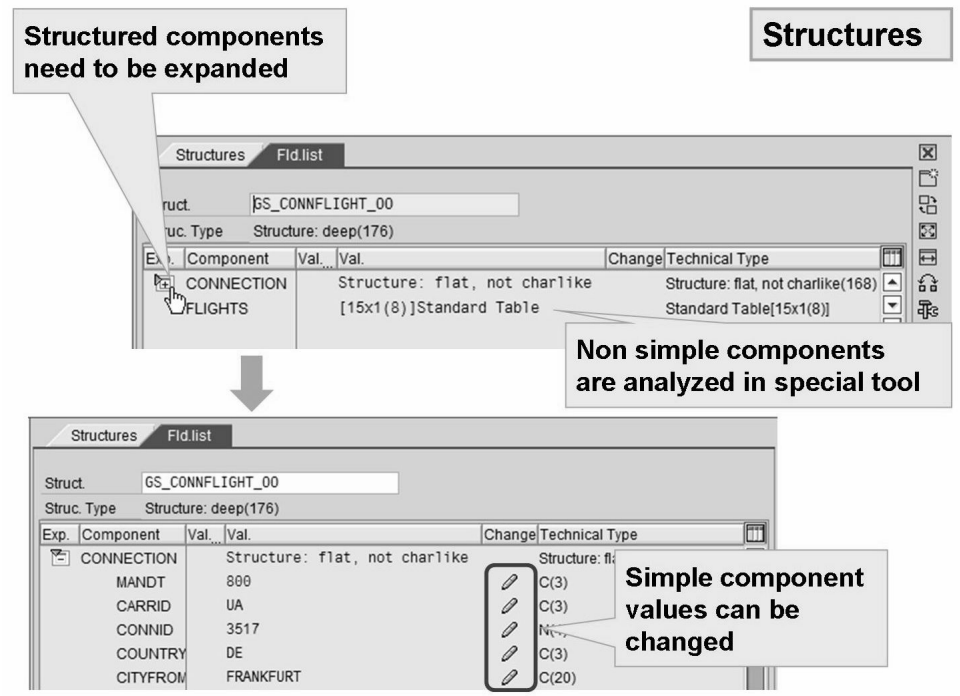


Figure 26: Explore structures

The **tool services** include the following important functions:

The information displayed on the current tab page can be stored as the content of a file in the file system. As of SAP NetWeaver 7.0 EhP 2, an XML download permits to save complex structures in an appropriate way. The read-only mode can be toggled for all components at once.

Analyze Internal Tables

The ***Table*** debugger tool displays the content of internal tables. All functions available before SAP NetWeaver 7.0 EhP 2 are assigned to the **tool services**. This includes the following options:

- Row operations:
  - Selected rows may be deleted or edited.
  - A row may be appended or inserted using index access. Here another table row may serve as a template.
  - All rows may be deleted.
  - The cursor may be positioned in an arbitrary table row according to a given index
- Column operations.
  - The order of the columns may be altered / reset.
  - The width of the columns may be optimized / reset
  - Selected rows may be deleted or edited

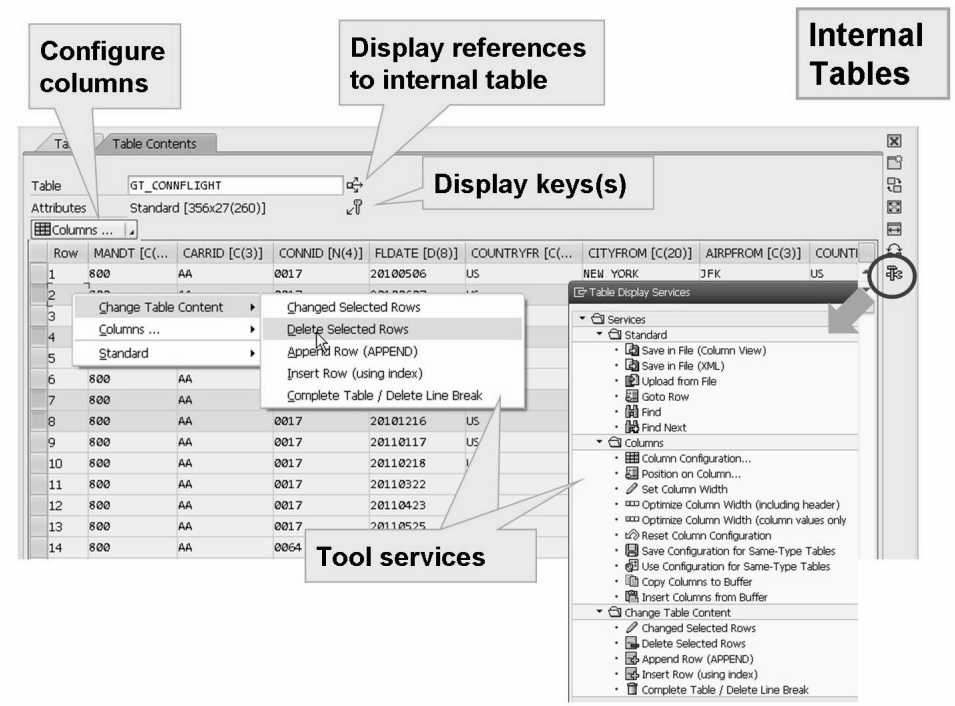


Figure 27: Explore tables - overview

In addition, standard ALV functions are available (interchanging columns via drag-and-drop, changing column width via drag-and-drop, scroll vertically / horizontally, mark lines / columns).

As of SAP NetWeaver 7.0 EhP 2, new service options have been developed related to the column settings:

The service option ***Column Configuration ...*** is used to change the column order, to delete columns or to add new columns to the table display. These additional columns may be used to display details related to a more complex information contained in an existing column (e.g. an attribute of an object, the component of a structure). The final column sequence can then be copied to the clipboard or inserted from the clipboard. This may be used to copy the column names to a mail body and send it to another person. This person may pasts the column names to his/her configuration dialog.

The service option ***Position on Column ...*** is used to display the names of all columns. If a column name is clicked, the first visible column is adjusted accordingly. Searching for a column name facilitates the handling of tables containing a large number of columns.

The service option ***Set Column Width*** permits to set the width of one column or of all columns at once (width in characters).

The service option ***Save Configuration for Same-Type Tables*** saves the current column configuration (column sequence and column widths). If an internal table with the same line type is displayed in a latter debugger session, this configuration is loaded automatically. In this case, a yellow bulb is displayed above the table display.

The service option ***Use Configuration for Same-Type Tables*** is used to reload the default configuration for the current line type.

The service option ***Delete Default Settings*** permits the deletion of the default configuration for the current line type.

The service option ***Copy Columns to Buffer*** copies the column names to the clipboard.

The service option ***Insert Columns from Buffer*** adapts the column configuration if possible.

Finally, new options to save or load the table content have been developed (*Save to File (XML)*, *Upload from File*).

Note: Default table configurations are user-dependent.

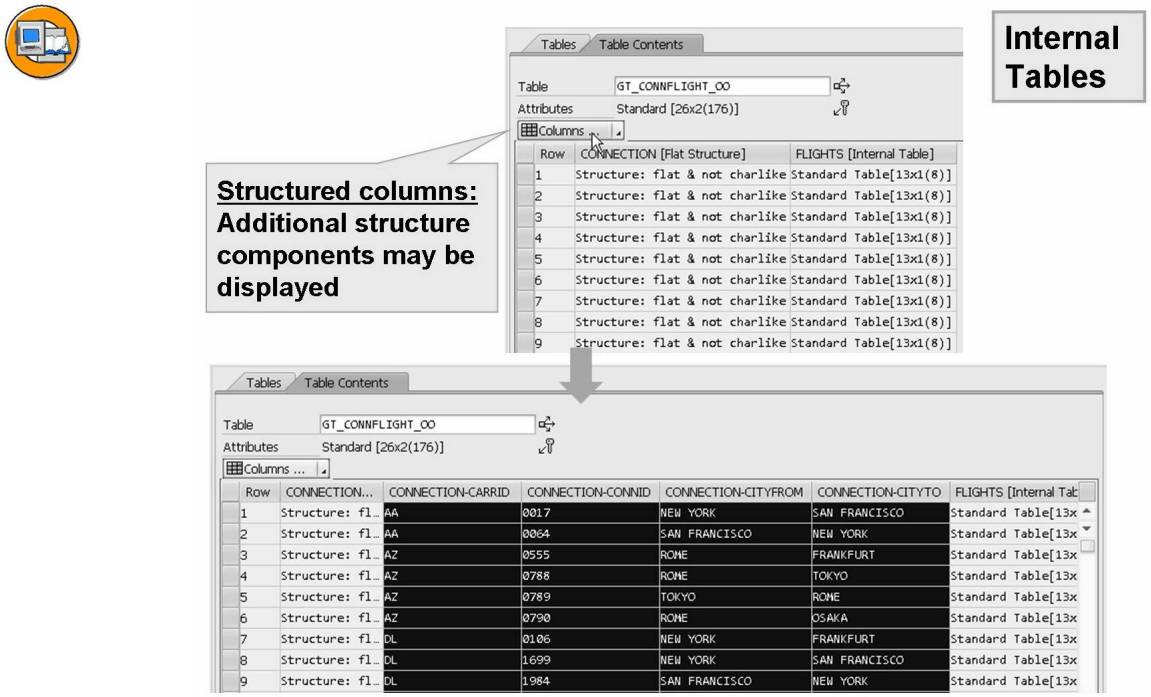


Figure 28: Add table columns not defined in internal table (1)

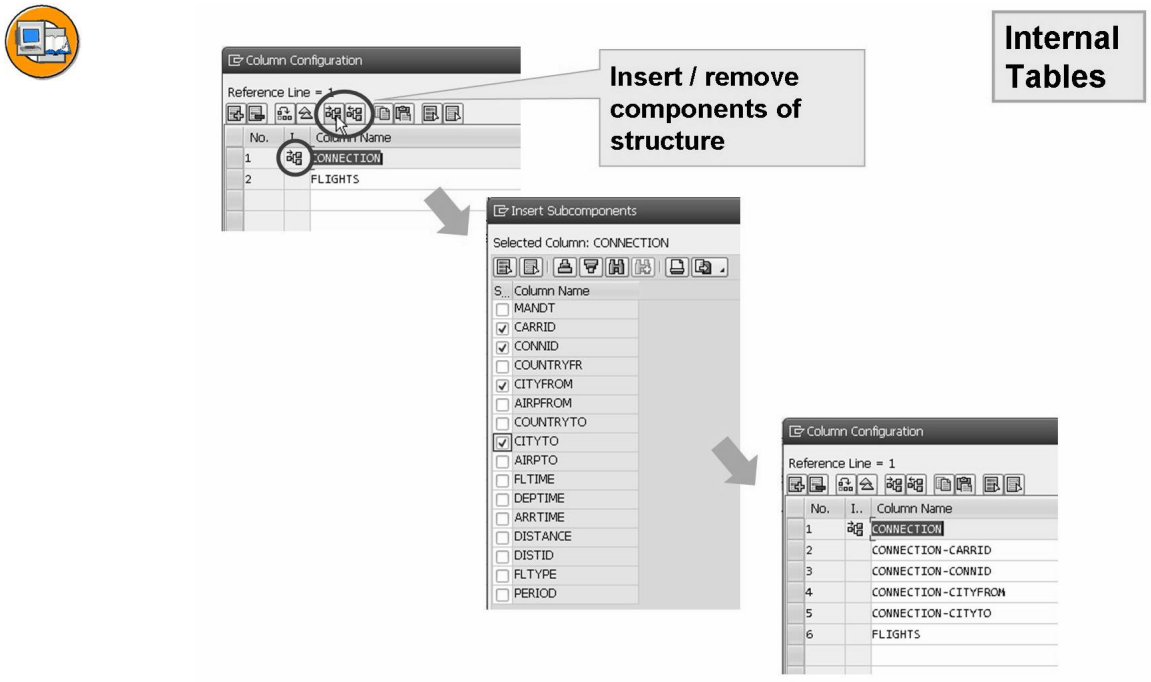


Figure 29: Add table columns not defined in internal table (2)

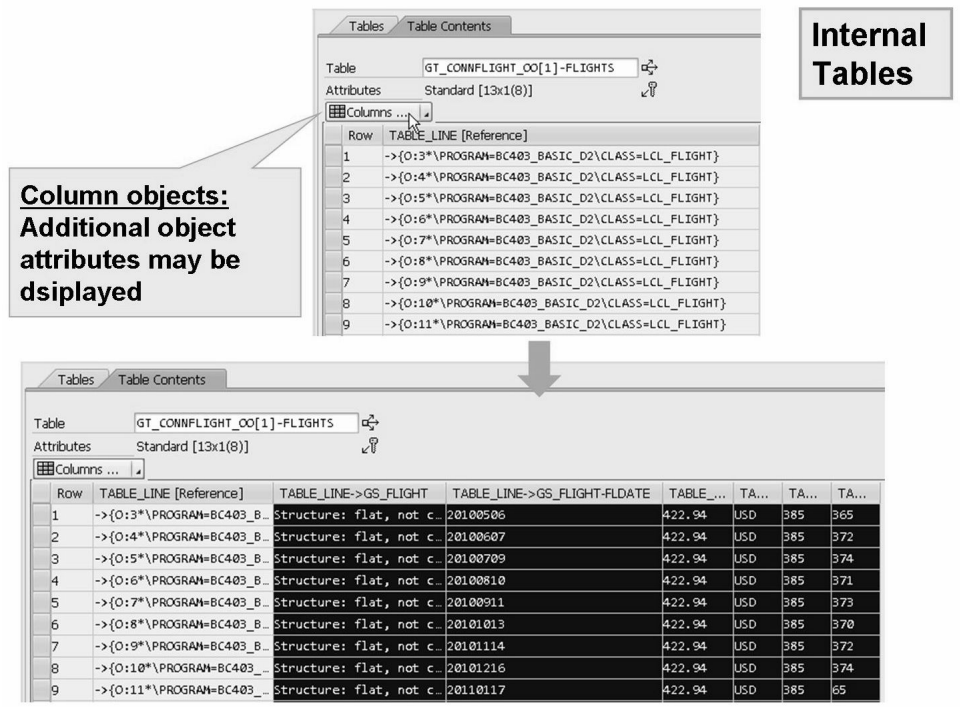


Figure 30: Add table columns not defined in internal table (3)

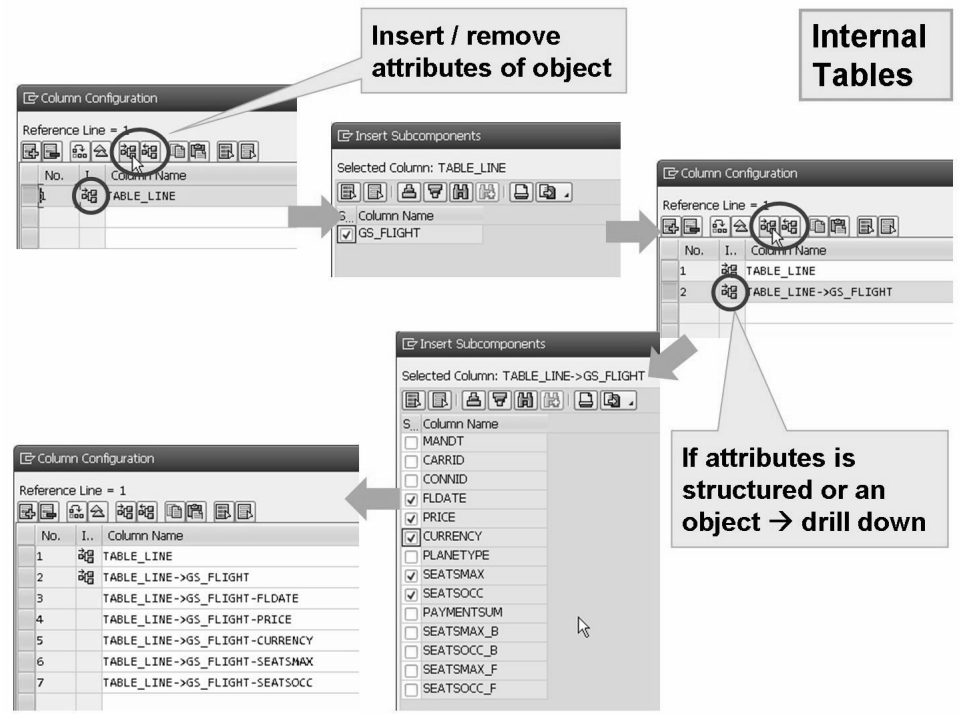


Figure 31: Add table columns not defined in internal table (4)

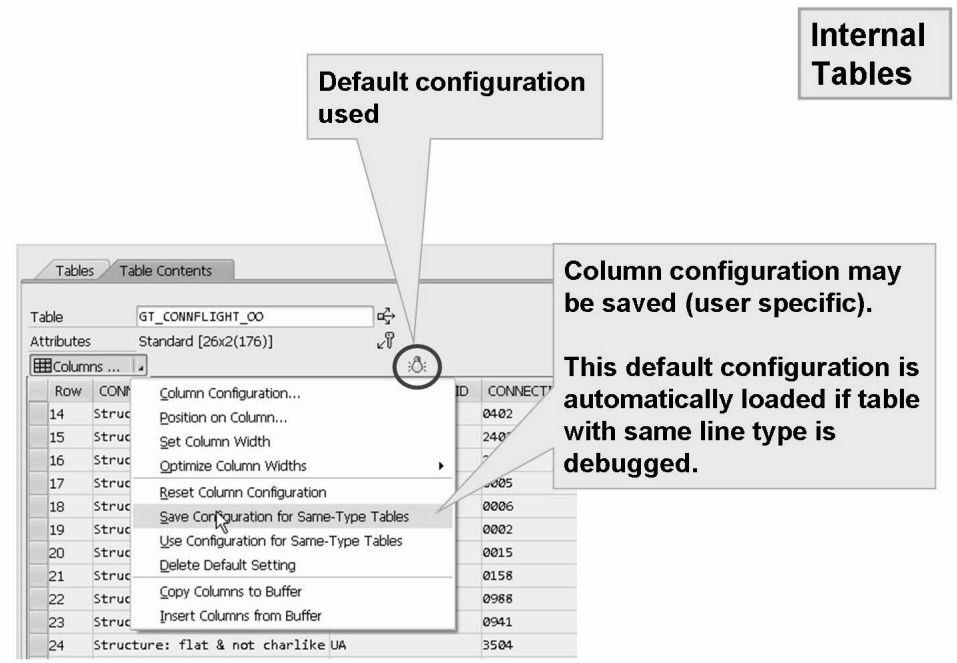


Figure 32: Save / reuse column configuration

To open the column configuration dialog, a button has been added above the table display. A key symbol may be clicked to display all keys of the internal table. An icon right of the field displaying the table name can be clicked to find out all references pointing to the internal table (body).

Analyze Object Reference Variables

The ***Object*** debugger tool is used to explore ABAP objects. Toolbar buttons displayed below the input field labeled *View* are used to define, which object constituents are displayed in the area below the toolbar:

- **Display Attributes**:

  Constants, static attributes and instance attributes defined in the object class or a super class are displayed together with their visibility. If an attribute has a simple type, the value is displayed right of the attribute name. For all other types the values have to be explored based on the appropriate debugger tool (double-click the attribute name).

  If an inheritance relations exist between the object class and a super class, the attributes are grouped by the classes (default). This behavior can be switched off and on again by clicking the toolbar button **Superclasses On/Off**. A filter can be imposed on the attributes by clicking the toolbar button **Filters**. In the dialog popping up, checkboxes permit to configure which attribute kinds should be hidden (e.g. private and protected instance attributes).

- **Display Events**:

  Events defined in the object class or a super class are displayed together with their visibility.

- **Display References**:

  All reference variables pointing on the object are displayed.

- **Display Inheritance Relationship**:

  Displays hierarchy of object class and all super classes.

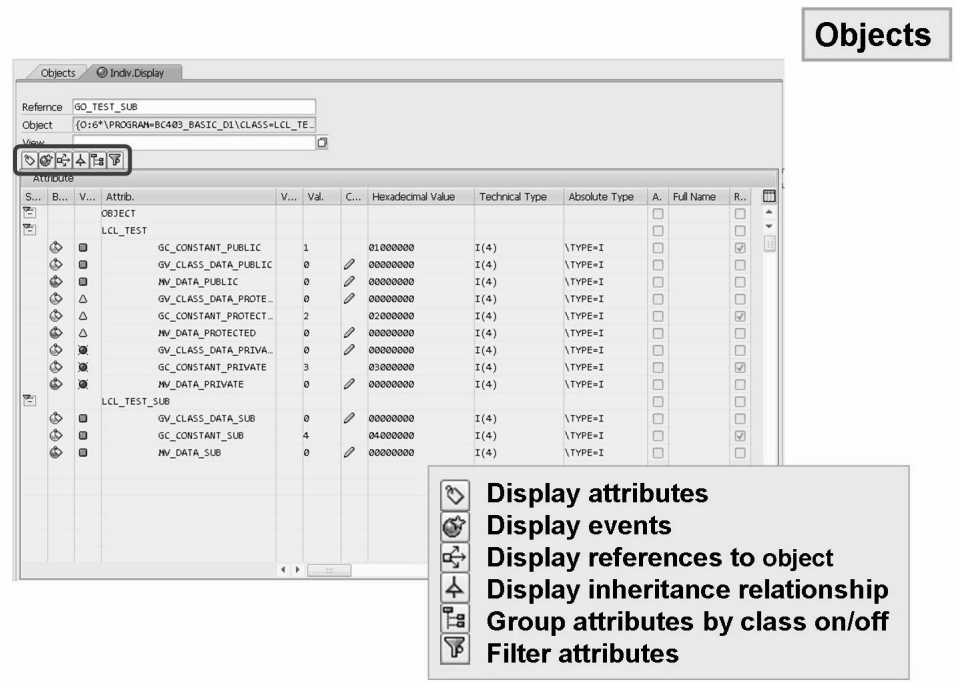


Figure 33: Explore objects

If an inheritance relations exist between the object class and super classes, the field labeled *View* is of interest. Here, the name of any super class may be entered. Then all constituents related to a class inheriting from the entered class are hidden.

The **tool services** include the following functions:

The information currently displayed can be stored as the content of a file in the file system. As of SAP NetWeaver 7.0 EhP 2, the content can also be exported as an XML file.

The Data Explorer

The ***Data Explorer*** debugger tool is used to display all types of data objects using a hierarchical tree. This allows the user to drill down into the details without having to switch between different debugger tools. In addition, the correlation between the variables is always visible.

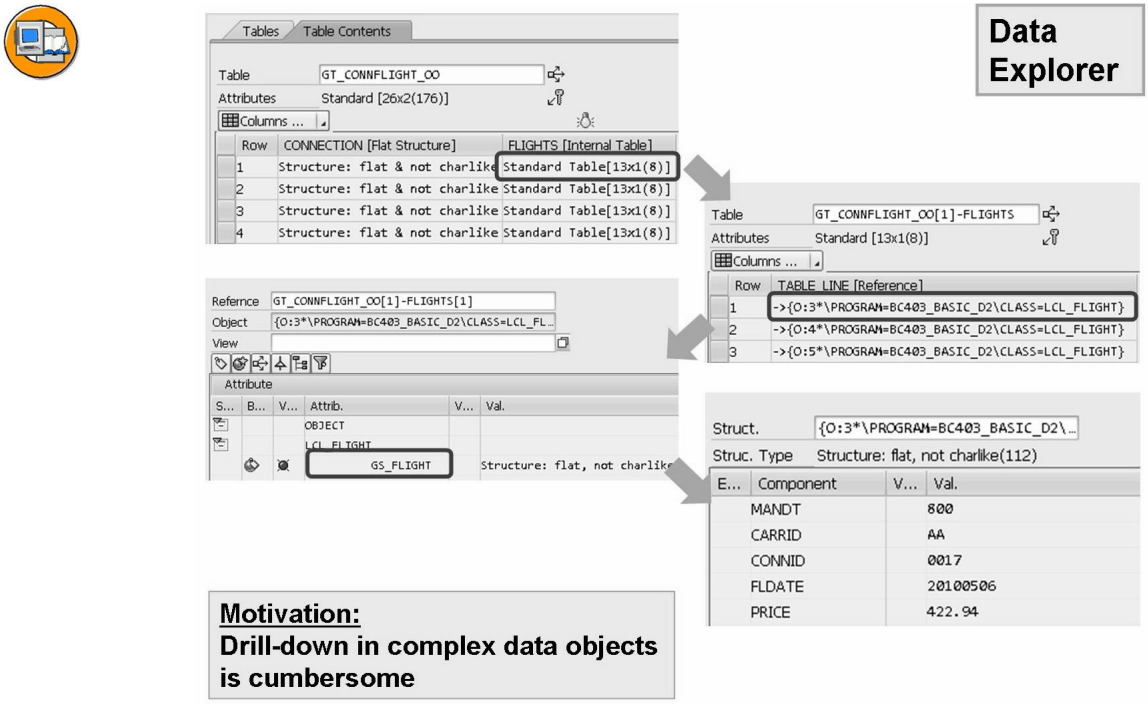


Figure 34: Explore complex variables using the Data Explorer (1)

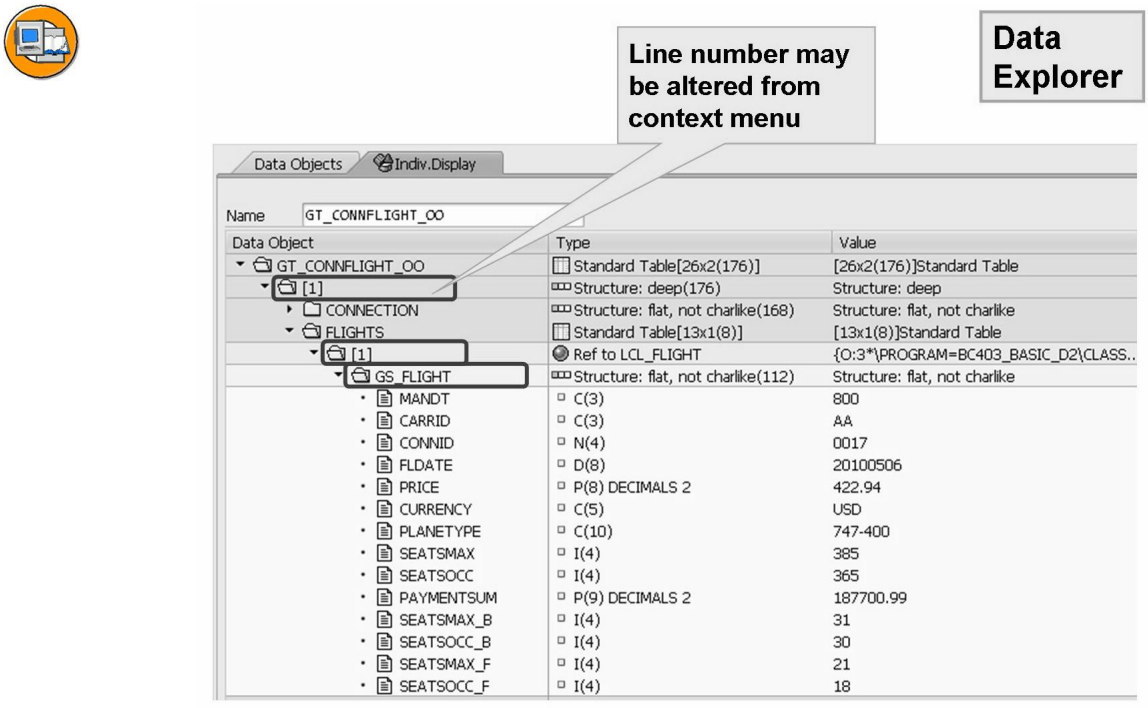


Figure 35: Explore complex variables using the Data Explorer (2)

Compared to the specialized debugger tools the following functions are not available:

- Only variable names, variable values, and the related data types are displayed.
- The values cannot be changed.
- Additional tool services are not offered.
- For any table only one row is displayed at the same time.

Note: The row number may be changed from the context menu that pops up when right mouse clicking on the row index.

Compare Variables

The ***DiffTool*** debugger tool is used to compare two variables having the same type but different values. This is very helpful to find differences between large strings or xstrings or between internal tables containing many rows. Attributes related to different objects being instances of the same class may also be compared.

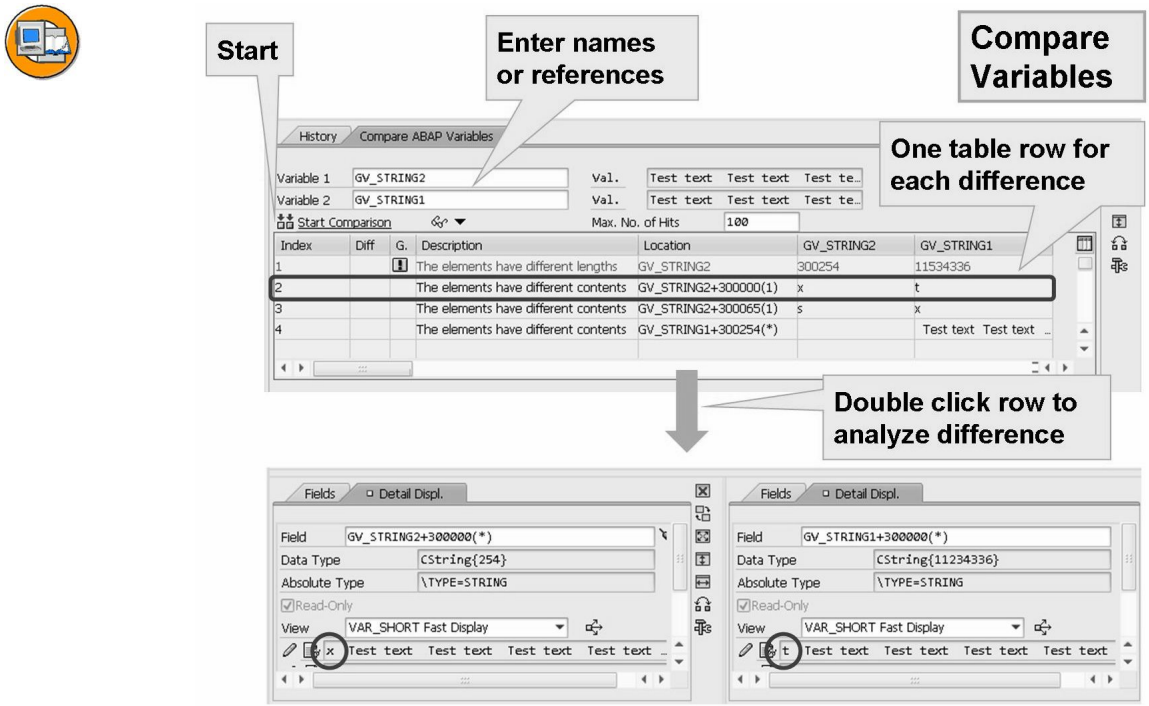


Figure 36: Compare variables

The names of the variables have to be entered in the fields labeled *Variable 1* and *Variable 2*. If the names of two reference variables are entered, the value of the reference variables can be compared (reference semantics). To compare the constituents of two objects, the values of the related reference variables have to be entered in the fields (value semantics).

The comparison is started by clicking the link labeled *Start Comparison*. For each difference one row is filled in the table. This row contains a description of the difference and a description of the location. For both variables, the segment related to the difference is displayed.

To explore a difference, any text contained by the table row may be double clicked. This opens two instances of the suitable debugger tool for this type of variable. Each variable value is displayed by an own instance. The values are positioned according to the difference location.

Lesson Summary

You should now be able to:

- Configure the layout of the ABAP Debugger
- Use breakpoints and watchpoints
- Save and load debugger session metadata
- Display and manipulate data objects
- Influence the program flow

Lesson: Debugger - Advanced Functionality

Lesson Overview

In this lesson you learn about advanced features of the ABAP Debugger. In the first section, the memory consumption of variables and programs is discussed. The next section deals with the analysis of loaded programs. Then, an overview over debugger tools to analyze screens, Web Dynpro controllers, or exceptions is given. Next, debugging issues related to programs that access the database are listed. Finally, debugger options and debugger settings are discussed.

This lesson is based on the new debugger, which is available as of SAP Net Weaver 2004.

Lesson Objectives

After completing this lesson, you will be able to:

- Analyze the memory consumption of variables and programs
- Analyze the programs loaded in the current internal session
- Explore screens and Web Dynpro controllers
- Analyze exceptions objects
- Start and stop SE30 and ST05 trace files from a debugger session
- Configure the ABAP Debugger
- Debug programs that access the database

Business Example

You would like to find out which programs and which variables are responsible for the observed memory consumption. In addition, you need to analyze programs that access the database.

Memory Analysis - Variables

In order to analyze the memory consumption of variables, the *Memory Analysis* debugger tool is available.

As of SAP NetWeaver 7.0 EhP 2, two additional tools, the *Memory Object Explorer* debugger tool and the *Application Specific Memory Views* debugger tool, are available. In addition, the memory consumption of a variable can be monitored in the *Fast Variable Display* debugger tool.

Internal Use SAP Partner Only

Analyze Single Variables

The fastest way to determine the memory consumption of a variable is provided by the ***Fast Variable Display*** debugger tool. On the tab labeled *Memory Analysis*, multiple variable names may be entered in the corresponding column of the table. The allocated memory and the used memory is displayed in bytes.

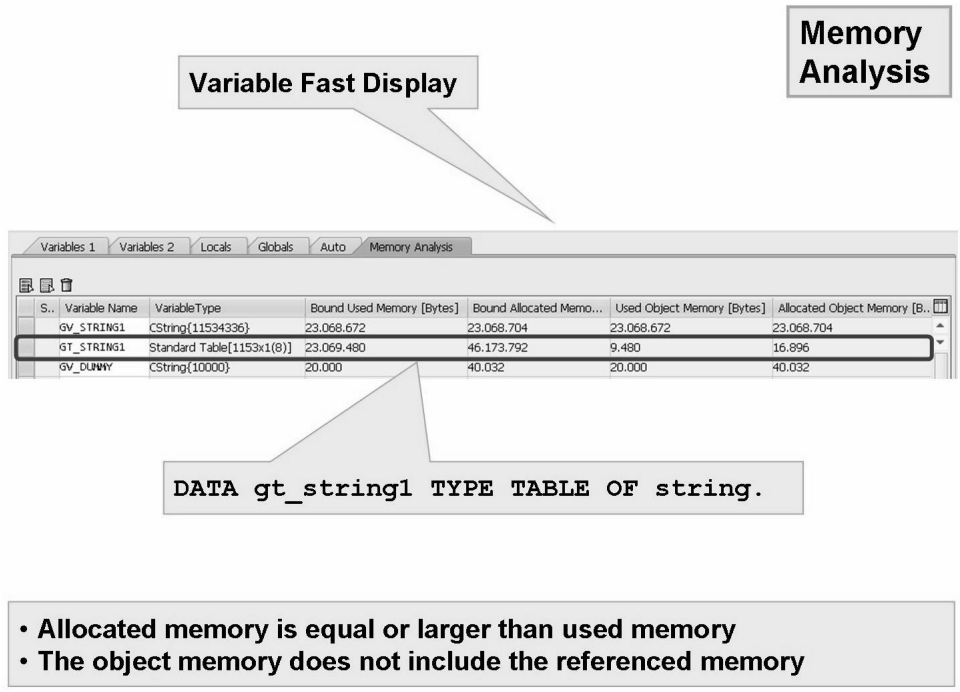


Figure 37: Analyze memory consumption of single variables

Note: For dynamic variables the allocated memory may be larger than the used value.

Note: The object memory contains only the memory that is directly related to the variable. The bound memory also includes memory referenced by the variable.

Example: A table column contains references to strings. Then, the object memory contains the memory related to the table administration plus the memory used by all string references (each reference variable occupies 8 bytes). However, the object memory does not include memory related to the string content.

Analyze dependent Variables

The ***Memory Object Explorer*** analyzes dependencies between memory objects. This tool helps visualization and navigation in the tree of "children" (referenced memory objects) or the "parents" (referencing memory objects), of an individually specified memory object. In addition, the memory bound by and allocated to memory objects is displayed.

A single memory object is taken as the start point for the dependency analyses. The name or the ID of the memory object has to be entered in the input field labeled *Name*. Example: An object can be qualified by the name of the reference variable (e.g. GO_ALV) or by the value of the reference variable (e.g. {O:1276...}).

To refine the display, the navigation direction for the analysis of the dependency has to be set:

- **Memory Objects Contained:**

  Each node of the hierarchy contains the memory objects that depend on the node object.

- **Higher-Level Memory Objects:**

  Each node of the hierarchy contains the memory objects that contains the node object.

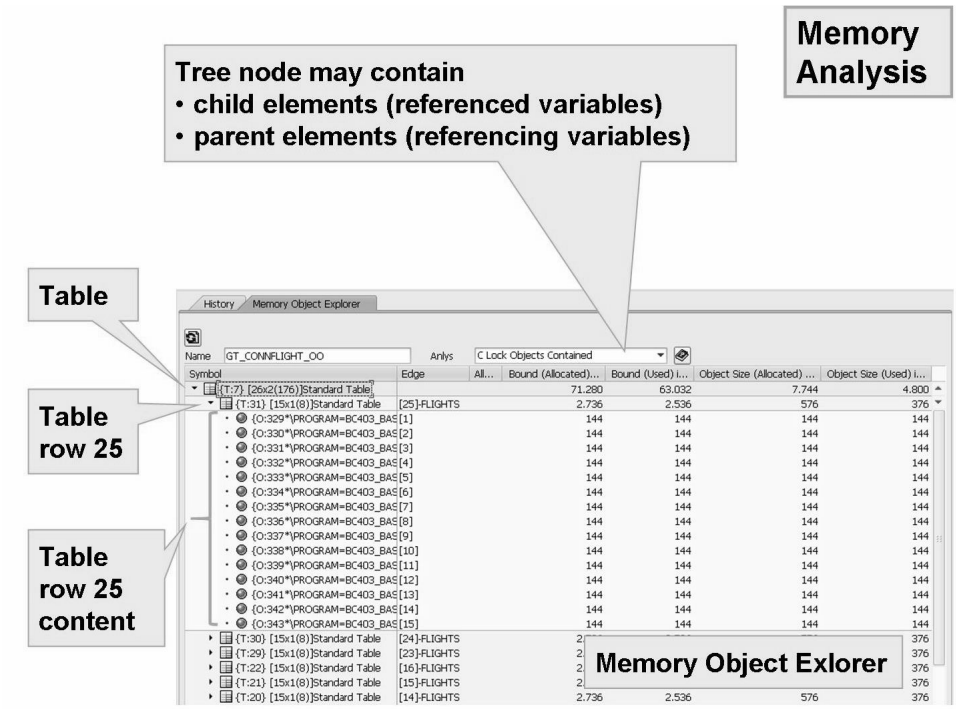


Figure 38: Analyze memory consumption of dependent variables

Using the context menu of the memory objects in the tree view, every memory object can be set as the new start point for the representation of the display (context menu item *Use As Start Point*). By double-clicking on a memory object in the tree, the object is displayed in the respective type-specific debugger tool.

For each memory object, a symbol name, the name of the respective edge to the memory object, as well as various memory sizes are displayed. The following differentiations are made:

- **Bound (Alloc.) in Bytes**:

  Allocated memory bound directly or indirectly by the memory object.

- **Bound (Used) in Bytes**:

  Used memory bound directly or indirectly by the memory object.

- **Object Size (Alloc.) in Bytes**:

  Memory allocated by the memory object directly.

- **Object Size (Used) in Bytes**:

  Memory used by the memory object directly.

If multiple direct connections exist between two memory objects, the respective referenced object is only represented once in the tree view. In this case, a pushbutton is created in the column labeled *All Edges*. If this button is clicked, the names of all connections can be displayed. The type of the individual memory objects is shown with a corresponding icon in the tree. The following types are possible:

- **Strings**: {S:id}
- **Internal Tables**: {T:id}
- **Boxes**: {B:id}
- **Objects**: {O:id}
- **Anonymous Data Objects**: {A:id}

Cycles in the graphs of memory objects are represented by a corresponding icon and the quick info text *Recursive Reference*. The root objects in graphs of memory objects are also marked as such by an icon of their own.

Analyze current internal Session

The ***Memory Analysis*** debugger tool is used to display the memory consumption for the current internal session. For this purpose, the tool provides three display screens with different information. The first screen (***Memory Sizes***) displays the following information (SAP NW 7.0 EhP 2):

- **ABAP Application**:

  Memory consumption related to the application grouped by:

  – **Static Variables:** The sum of all static variables of all loaded programs and all static attributes of all loaded classes.

  – **Stack (Used):** The total of the use data for all parameters and the local variables of the ABAP procedures on the ABAP stack.

  – **Dyn. Memory Objects (Used);** The total of the use data for all existing dynamic memory objects, such as class instances, anonymous data objects (field symbols, data references), Internal tables, strings, and boxed components.

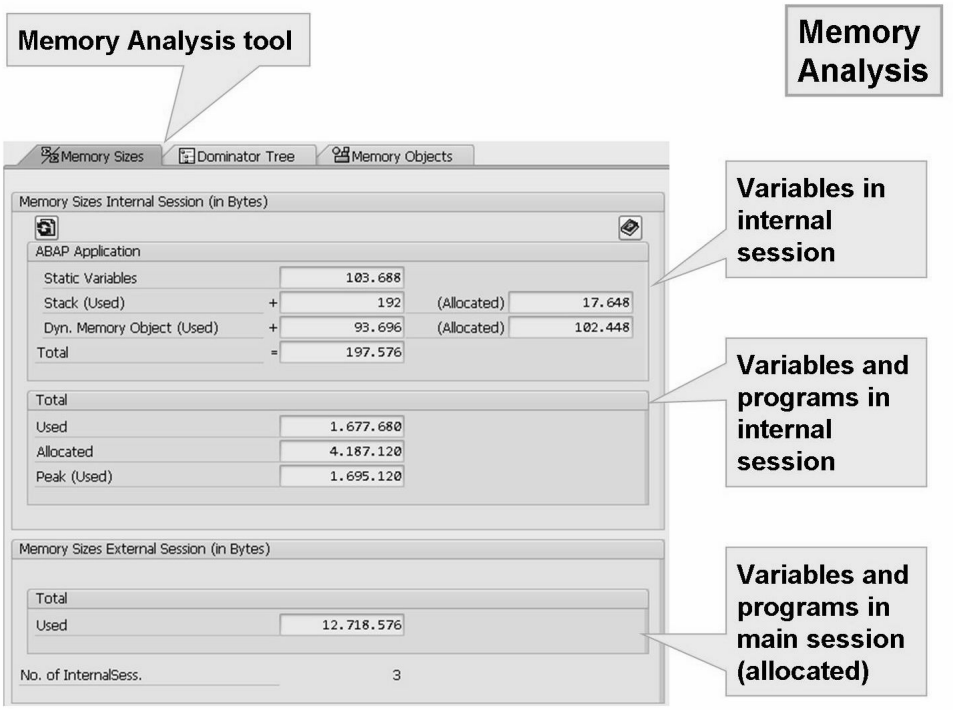


Figure 39: Analyze memory consumption of current internal session

Together these make up the total memory usage of the ABAP application (displayed in the field labeled *Total*). In addition, the values for the allocated memory usage of the ABAP application for the stack (*Stack (Allocated)*) and the dynamic memory objects (*Dyn. Memory Objects (Allocated)*) are specified. Since

the complete graph of the memory objects must run to determine these memory sizes, the sizes are not determined automatically. These values are only refreshed when the refresh button is pressed. This can be changed in the **tool settings**.

- **Total - internal session**:

  Memory consumption related to the internal session grouped by:

  – **Used:** The value that is currently used by the internal session of the SAP system in total. As well as the memory for the ABAP application, further kernel layers and kernel components are included.

  – **Allocated:** The total currently allocated virtual memory against the operating system for this internal session

  – **Peak (Used):** The maximum value of the memory usage value in the transaction running.

- **Total - external session**:

  The total currently allocated virtual memory against the operating system for all internal session of the main session. Additionally, the current number of internal sessions is specified.

The screen labeled ***Memory Objects*** displays a ranked list of all dynamic memory objects, sorted by size. The number of entries may be restricted by using the *Entries* input field. The memory object display can be influenced by using the *View* dropdown box. The following views are available:

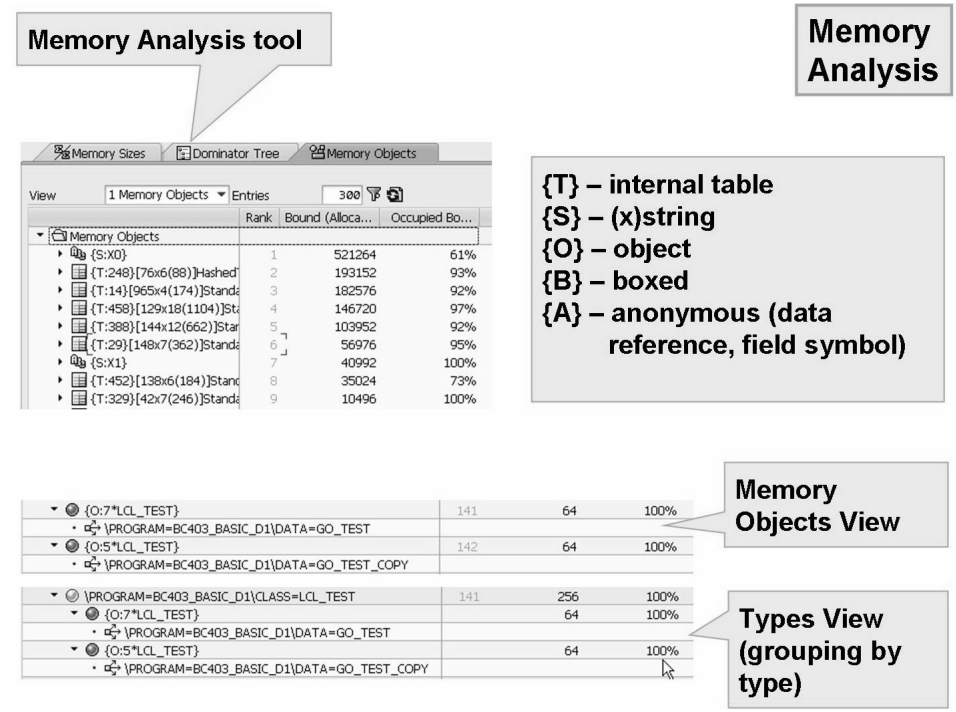


Figure 40: Analyze memory consumption of internal session variables

- **Memory objects:** In the *Memory* Objects view, individual memory objects such as class objects, anonymous data objects, internal tables, and strings are displayed.
- **Types:** In the *Types* view, the class objects and anonymous data objects are grouped into classes and types. The used memory is summed. In the case of classes, the static memory of the class is also considered.
- **Aggregates (Cycles):** The *Aggregates (Cycles)* view groups together individual memory objects that are part of a strongly connected component. Strongly connected components can be viewed as composite objects that can only be deleted as a whole.

In all views, the bound memory (in bytes), the occupied memory (in %) and the rank index is displayed.

As of SAP NetWeaver 7.0 EhP 2, a third screen labeled ***Dominator Tree*** is provided by the *Memory Analysis* debugger tool. This view offers a list of the dynamic memory objects ranked by size. In contrast to the *Memory Object* view, the cross references between dynamic memory objects are taken into account. Related memory objects form a tree, thus on the first hierarchy level only the memory objects that are not references by other memory objects are displayed. The memory consumption related to a node includes the memory consumption of all child elements.

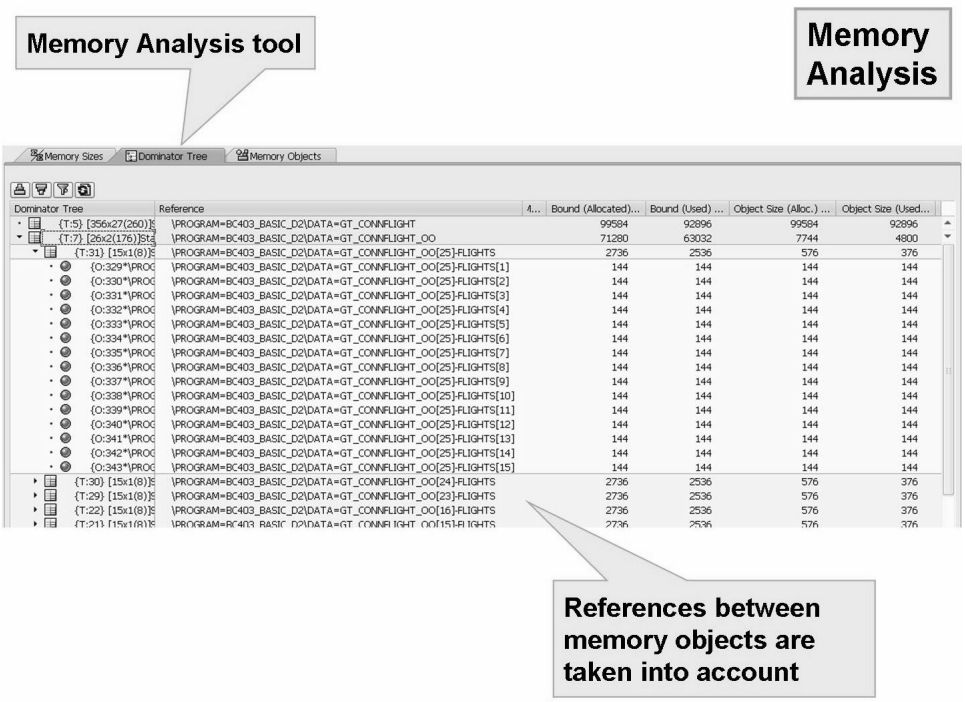


Figure 41: Analyze memory consumption of dependent variables

The following important **tool services** should be mentioned: Memory snapshots can be created and compared using the **Memory Inspector** (transaction code DBG_MEMORY_DIFFTOOL).

Analyze Web Dynpro and ICF

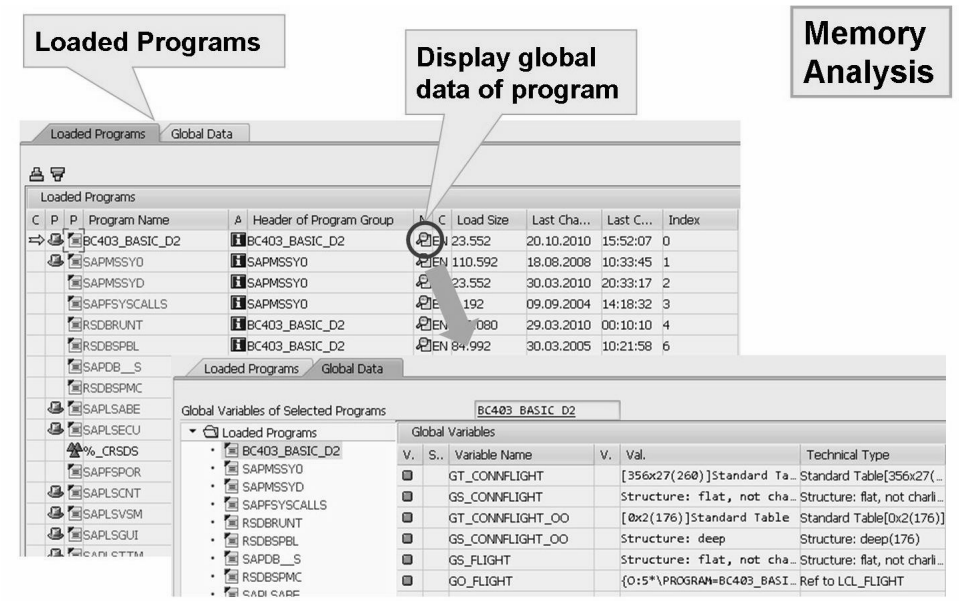
The ***Application Specific Memory Views*** debugger tool is similar to the *Memory Object Explorer* debugger tool: Dependent memory objects are displayed in a hierarchical tree together with the object type and memory consumption. However, the following differences exist:

- Multiple memory objects are displayed on the first hierarchy level at the same time.
- The names of the memory objects are predefined and cannot be altered. Which objects are displayed depends on the *Memory View* setting. A dropdown list allows the user to select *Internet Communication Framework* or *Application Web Dynpro*.
- The memory object list may be sorted according to the memory consumption.
- A filter allows the user to hide certain memory object types.

Memory Analysis - Programs

The ***Loaded Programs*** debugger tool displays all loaded programs in the current internal session and the current values and attributes of all global variables of the loaded programs.

The tab page labeled ***Loaded Programs*** displays the names of all loaded programs and their attributes. The table of loaded programs consists of the following information:



System programs (blue) may be hidden by service tool

Figure 42: Analyze loaded programs

- **Current Program**: The current program is indicated by a yellow arrow with the quick info text *Current Program*.
- **Program Group Header**: All programs that head a program group are indicated with an icon displaying a head and the quick info text *Head of a Prg. Group*. If the icon is clicked, a dialog box appears in which all of the program group's programs are displayed.
- **Prog. Type**: The program types are indicated by icons with the following quick info texts: *Program*, *Type pool*, *Class*, or *Interface*. System programs are colored in blue.
- **Program Name**: Name of the compilation unit (program). System programs are displayed in blue. Double-clicking the program name displays the source code in a new window.
- **Attributes of Program**: If the icon with the quick info text *Program Attributes* is clicked, the attributes of the program are displayed in a dialog box.
- **Header of Program Group**: Program that heads the program group. Double-clicking the program name displays the source code in a new window.
- **Navigation to Global Variables**: If the icon with the quick info text *Global Data* is clicked, the system opens the *Global Data* tab page and displays the global data of the selected program.
- **Original language**: Original language of the program.
- **Load Size**: Size of the compilation unit.
- **Last Change (Date)**: Date of last program change (including includes).
- **Last Changed (Time)**: Time of last program change (including includes).
- **Index**: Order of program instantiation.

The tab page labeled ***Global Data*** is divided into the following screen areas:

- On the left there is a list of the loaded programs. If a program is double-clicked in the list, the global variables of this program are displayed in the right screen area.
- On the right, the system displays the global variables of a program. This screen area provides complete information about the variables. If the variable name is double clicked, the system automatically opens the corresponding detailed display.

The **tool services** offer the following functionality: The table display can be saved as a file. In addition, the system programs can be hidden.

Additional Debugger Tools

This section gives a brief overview over debugger tools not mentioned above.

Screen Analysis

The ***Screen Analysis*** debugger tool is used to explore the following aspects of screens:

- Display hierarchy of screens and container objects.

  Note: The following screen objects are considered as container objects:

  Frame, subscreen, tabstrip, table control, step loop, custom container, splitter container control, docking container control.

- Display screen properties and the field list related to a screen.
- Display container properties.
- Check, whether sizing of container objects is possible (scroll bars).
- Display screen stack and navigate to other screens on current stack.

Web Dynpro Analysis

The ***Web Dynpro*** debugger tool is used to explore the following aspects of Web Dynpro controller objects:

- Display list of controller objects.
- Display list of used components and related controller objects.
- For each controller object:
  - Display properties.
  - Display attributes.
  - Allow navigation to debugger tool suited to display attribute value.
  - Display context hierarchy.
  - Display attributes related to context elements.
- For view controller object:
  - Display view hierarchy.
  - Display attributes of view elements.
- For window controller object:
  - Display hierarchy of embedded views.

Exception Analysis

The ***Display Exception*** debugger tool is used to explore exception objects and exception object chains. After having entered the name of the exception object reference in the field labeled *Object / Reference*, the following information is available:

- All exception objects related to the current exception object chain, are displayed in a table.
- The text stored in the exception class is displayed right of the object.
- The long text related to the exception is displayed if the icon with the quick info text *Display Long Text* is clicked.
- The position related to the exception is marked red if the icon with the quick info text *Position in Source Code Display* is clicked.
- If an exception object is double clicked in the table, details of the object are displayed in the *Object* debugger tool.

Note: This tool is available as of SAP NetWeaver 7.0 EhP 2.

Define Traces while Debugging

The ***Trace (SE30/ST05)*** debugger tool is used to start and stop tracing from within the debugger session. In addition, the trace tools may be started from here in order to display the traces. The following traces are supported:

- Runtime Analysis Trace (transaction SE30), without aggregation of results.
- Runtime Analysis Trace (transaction SE30), with aggregation of results.
- SQL trace file using transaction ST05.
- Database buffer trace file using transaction ST05.
- RFC trace file using transaction ST05.
- Enqueue trace file using transaction ST05.

To start or stop tracing, the icon in the column labeled *On/Off* has to be double-clicked. If tracing is active, a green led icon is displayed in the related table line, if tracing is inactive, a red led icon is displayed instead. If a trace file exists, an icon is displayed in the first column labeled *Trace File*. In this case, the columns labeled *Start Time*, *Start Date*, *Stop Time*, and *Stop Date* are filled accordingly. Double clicking the trace file icon opens the related transaction and displays the trace file.

Note: This tool is available as of SAP NetWeaver 7.0 EhP 2.

Other tools

The debugger tools *Debugger Scripting* and *Script Wrapper* are related to the processing of user defined scripts while debugging. The *XML and List Preview* debugger tool is used to display the XML stream created by a simple transformation or the content of the list buffer (alternatively unformatted or formatted).

Note: These tools are available as of SAP NetWeaver 7.0 EhP 2.

Debugger Customizing

The customizing of the debugger is split in two parts, the customizing of debugger options and the customizing of debugger settings.

Customize Options

Each user can personalize the debugger options (menu item *Settings* → *Customizing*. The options are distributed among four tab pages.

On the first tab page labeled ***General***, navigation settings can be adjusted.

The first setting decides if the navigation to the source code (*Goto Source Code*) is performed in the same window or in a new window. If the navigation is performed in the same window, F3 (Back) has to be used to get back to the debugger.

The second navigation setting defines, if a debugger tool used to explore a variable is added to the current desktop, or if the desktop related to this tool is used. However, only 4 tools may be displayed on one desktop. Thus, two additional settings exist to handle this limit situation: The current tool may be replaced by the new tool, or the desktop related to the new tool may be opened.

The last setting is used to define if double clicking a variable in the *Source Code* debugger tool opens the variable in the *Variable Fast Display* debugger tool, or in the special tool for this kind of variable.

On the second tab page labeled ***Call Stack***, the stack type can be set. The stack type defines the content displayed by the *Call Stack* debugger tool (only ABAP program units, only screens, or both information).

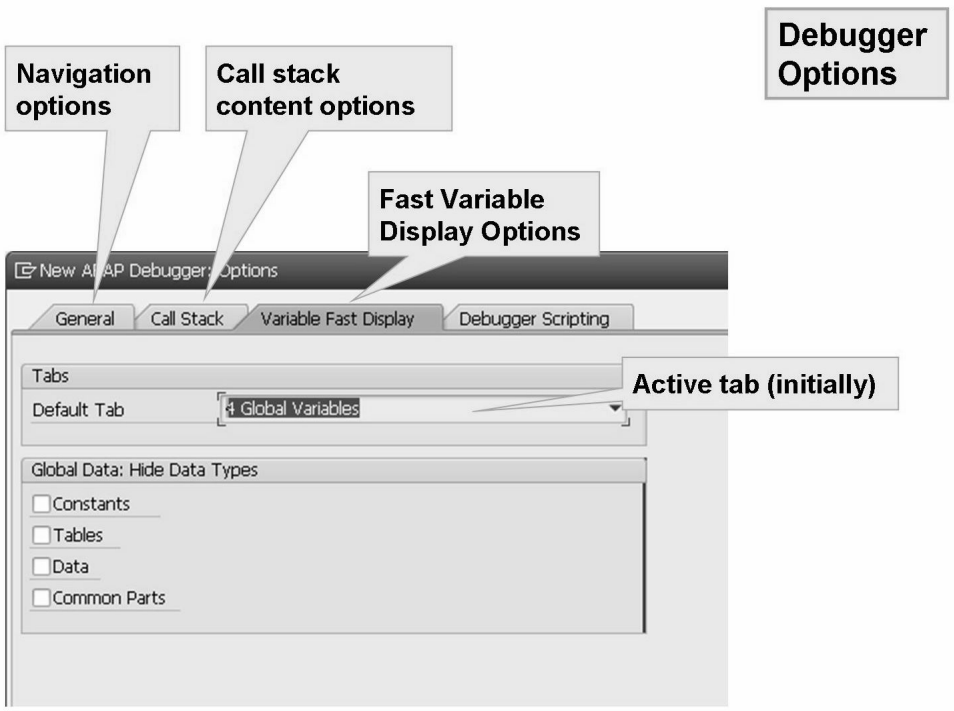


Figure 43: Adjust debugger options

On the third tab page labeled ***Variable Fast Display*** default setting for the *Variable Fast Display* debugger tool can be set. This includes the definition of the tab that is initially active and the definition of global data objects kinds that should be hidden on the tab labeled *Globals*.

The last tab labeled ***Debugger Scripting*** allows the user to define how the system behaves, if a debugger script is executed, and if the debugger functionality *Continue (F8)* is selected.

To save the debugger options, the *Save* button has to be clicked. The options are saved with the default debugger session (name: *START_UP*).

Customize Settings

Each user can personalize the debugger settings (menu item *Settings → Change Debugger Profile / Settings*. The settings dialog contains two groups labeled *Debug Mode* and *Specific Settings*.

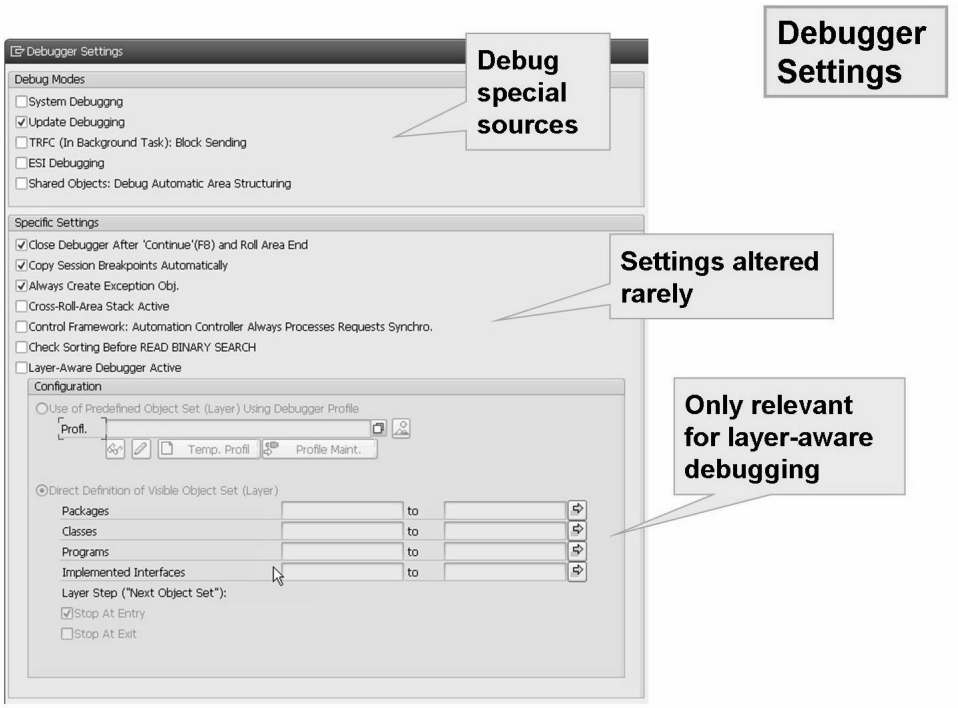


Figure 44: Adjust debugger settings

In the section ***Debug Mode*** the following settings can be adjusted:

- **System Debugging**: Defines, if programs marked as system programs (Program Status = 'S') are debugged.

  Note: This default setting may be changed in the debugger session (menu item *Settings → System Debugging On/Off*) or in the debuggee mode (menu item *System → Utilities → Debug System*).

- **Update Debugging**: Defines, if the update process is debugged (which starts after the statement *COMMIT WORK*). Details about debugging programs that access the database are explained below
- **TRFC (in Background Task): Block Sending**: Defines, if function modules called with the addition *IN BACKGROUND TASK* are processed. If the flag is set, the system collects the function calls, but does not start the background work process. The debugger assigns a transaction ID, which identifies the background work process uniquely. Later on, the Debugger for the background work process can be started using transaction SM58.
- **ESI Debugging**: Defines, if a service triggered by ESI is debugged (SAP NW 7.0 EhP 2).
- **Shared Objects: Debug automatic Area Structuring**: Defines, if the automatic creation of an area structure is debugged. This means that the processing of the constructor class related to the area is debugged (SAP NW 7.0 EhP 2).

In the section ***Specific Settings*** the following settings can be adjusted:

- **Close Debugger After Continue (F8) and Roll Area End**: Defines, if the main session related to the debugger is automatically closed when the debuggee is finished.
- **Copy Session Breakpoints Automatically**: Defines, if session breakpoints that are set from another main session of the same user session while debugging are respected. If this flag is not set, the *Reload Breakpoints* functionality of the debugger has to be used to update the breakpoint list (SAP NW 7.0 EhP 2).
- **Always Create Exception Object**: Defines, if exception objects for class-based exceptions are created even if the INTO addition is missing in the CATCH statement. When an exception occurs, a message appears in the status bar and two additional icons appear in the debugger toolbar having the quick info texts *Display Exception Object* and *Statement that Caused the Exception*.
- **Cross-Roll-Area Stack Active**: Defines, if the stack is displayed for the current internal session only (default), or if the stack is displayed for all internal sessions (checked). A new internal session is opened after a `submit...and return`, `call transaction`, or `call dialog` (SAP NW 7.0 EhP 2).
- **Control Framework...**: Defines, if control framework related communication steps are performed synchronously, or not (SAP NW 7.0 EhP 2).
- **Check Sorting Before READ BINARY SEARCH**: Defines, if the system checks whether the internal table is sorted before every execution of the statement. If the table is not sorted, a runtime error occurs. This setting should only be activated shortly before reaching the point in the source code, because there can be a significant loss in performance depending on the table size.
- **Layer-Aware Debugger Active**: Defines, if layer-aware debugging is active. If activated, additional settings related to the debugging layer(s) need to be set (SAP NW 7.0 EhP 2).

To save the debugger settings, the *Save* button has to be clicked. The options are saved with the default debugger session (name: *START_UP*).

Debug Programs accessing the Database

In a SAP system, a program is executed by a work process. In case of a waiting situation, the program is rolled out so the work process can handle another request. However, if the program is rolled out, the database connections has to be prepared for the next program that is handled by the work process. This means that the work process has to end the current database session (database logical unit of work = database LUW). This is done by sending an implicit database commit. All changes related to an ended database LUW cannot be rolled back.

In respect to the debugging of a program that accesses the database, this has the following impacts:

- If a waiting situation occurs while debugging a SELECT loop, the connection is closed and the database cursor is lost. If debugging is continued, this leads to a dump.
- If a waiting situation occurs while (or after) debugging a statement that changes database content, all changes performed since the last database commit are saved on the database. The changes may not be rolled back.

Since the contents of the database in a production client must always be consistent, the second impact is very critical. Thus, the user performing the debugging has to be sure that unwanted waiting situations may not occur.

The only waiting situation that may occur when debugging a program, but that does not occur when processing the program regularly, arises from the additional debugger dialogs. Thus, each time the debugger stops, an implicit database commit may be triggered by the corresponding work process. This can only be avoided if the work process handling the debuggee is not released as long as the debugging session is active. The automatic conversion of dialog processes to such exclusive debugging work processes takes place as long as a certain number of exclusive debugger work processes is not exceeded. This limit is defined by the **profile parameter *rdisp/wpdbug_max_no***.

Note: To ensure that debugging activities cannot block the whole system, only half of all dialog processes can be used for debugging.

Caution: Since only a restricted number of dialog work processes can switch to debugging mode, you should exit the Debugger as soon as you no longer need it. Otherwise, you unnecessarily block the exclusive work process.

If the conversion of a dialog work process to an exclusive debugger work process was successful, this information is added to the GUI title: *(exclusive)(<host> <SID> <sys.no.>)*. If the conversion was not successful, this addition reads *(NOT exclusive)(<host> <SID> <sys.no.>)*.

In the latter case, debugging should be deferred until a manual conversion is possible (menu item *Debugger → Exclusive Debugging Mode On*).

When debugging database changes in an exclusive database process, a rollback of the changes or an explicit database commit may be triggered from the debugger (menu items *Edit → Database → Commit (unlock)* and *Edit → Database → Rollback*). Thus, database changes can be tested without committing them (Rollback), or database locks can be released (Commit) while debugging.

Administration Aspects

The debugging authorization is determined by the authorization object ***S_DEVELOP***. The debugger can only be started if the user's authorization contains an appropriate value for the field ***ACTVT***. This may be:

- ***03***:
  - Debugger can be started.
- ***02***:
  - Debugger can be started.
  - Variable values can be changed.
  - Functionality *Goto Statement* is accessible.
- ***01***:
  - Debugger can be started.
  - Variable values can be changed.
  - Functionality *Goto Statement* is accessible.
  - System programs can be debugged
  - Kernel debugging is possible.

The following profile parameters are of importance:

- **rdisp/wpdbug_max_no**: Maximum number of dialog work processes that may used by the debugged program exclusively. Not more than 50% of the dialog work processes may be converted.
- **rdisp/max_debug_lazy_time**: Maximum idle time during debugging (no interaction). If this lazy time is exceeded, the debugger session is reset (main session closed).
- **rdisp/max_alt_modes**: Maximum number of main modes per user session (default: 6).
- **abap/ext_debugging_possible**: Restrict external debugging to certain user types.
- **rfc/ext_debugging**: Restrict external debugging for RFC scenarios.
- **rfc/enable_trfc_dbg_user_switch**: Activate tRFC/qRFC user switch for debugging LUWs.
- **rfc/disable_debugger_command_field**: Deactivate command field input n the debugger during the execution of reports and function modules in an RFC session.

Lesson Summary

You should now be able to:

- Analyze the memory consumption of variables and programs
- Analyze the programs loaded in the current internal session
- Explore screens and Web Dynpro controllers
- Analyze exceptions objects
- Start and stop SE30 and ST05 trace files from a debugger session
- Configure the ABAP Debugger
- Debug programs that access the database

Unit Summary

You should now be able to:

- Configure the layout of the ABAP Debugger
- Use breakpoints and watchpoints
- Save and load debugger session metadata
- Display and manipulate data objects
- Influence the program flow
- Analyze the memory consumption of variables and programs
- Analyze the programs loaded in the current internal session
- Explore screens and Web Dynpro controllers
- Analyze exceptions objects
- Start and stop SE30 and ST05 trace files from a debugger session
- Configure the ABAP Debugger
- Debug programs that access the database

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 2

Advanced Topics

Unit Overview

In this unit, the following aspects are covered: First, the restriction of the debugging process on predefined object sets is discussed. Then, the debugging of application that involve HTTP and RFC communication, user switches, or multiple application servers is explained. Finally, the usage of debugger scripts to simplify repetitive tasks, to create trace files while debugging, or to set sophisticated breakpoints and watchpoints is summarized in the last lesson.

Unit Objectives

After completing this unit, you will be able to:

- Define object sets and debugger profiles persistently using transaction SLAD
- Define debugger profiles transiently from a debugger session
- Use debugger profiles in a debugger session
- Define external breakpoints that are related to the terminal ID (TID)
- Use request based debugging with SAP GUI applications
- Use request based debugging with Web Dynpro for ABAP applications
- Create debugger scripts
- Execute debugger scripts
- Create and analyze debugger trace files

Unit Contents

Lesson: Layer Aware Debugging

Lesson Overview

Layer aware debugging is a new functionality available as of SAP NetWeaver 7.0 EhP 2. This concept allows the user of the new ABAP Debugger to define which sets of programming object are relevant for the debugging session. This way, source code not interesting for the debugging process is hidden. This lesson explains the definition of debugger profiles and the usage of these profiles in debugging sessions.

Lesson Objectives

After completing this lesson, you will be able to:

- Define object sets and debugger profiles persistently using transaction SLAD
- Define debugger profiles transiently from a debugger session
- Use debugger profiles in a debugger session

Business Example

You need to analyze a bug, or you want to find out how an application works. You don't know where exactly to set a breakpoint. So you start the ABAP Debugger and start stepping through the code. And you keep on stepping through the code, endlessly, without reaching any of the application logic in which you are interested. This is because your ABAP application makes use of application frameworks or technical infrastructures like ESI, Web Dynpro, or ALV. Thus you want to "turn off" the framework code and reach only your application logic in the Debugger.

Layer Aware Debugging

In modern software architectures, multiple software layers are involved in the processing of an application. Each of these layers defines an own framework, which is used by the following software layer. A good example is Web Dynpro:

A Web Dynpro application is based on the Web Dynpro framework, while the Web Dynpro framework is based on the Internet Communication Framework (ICF). Thus between the application layer and the ABAP kernel layer (containing ABAP and dynpro core functions), at least two additional software layers are involved.

In reality, each complex application involves multiple additional application layers. Thus, a lot of code that is not of interest to understand the current application is processed by the debugger.

It was always possible to exclude the ABAP kernel layer from debugging (system debugging switched off). However, in the past there was no mean to exclude selectively programming objects not part of the ABAP kernel from debugging.

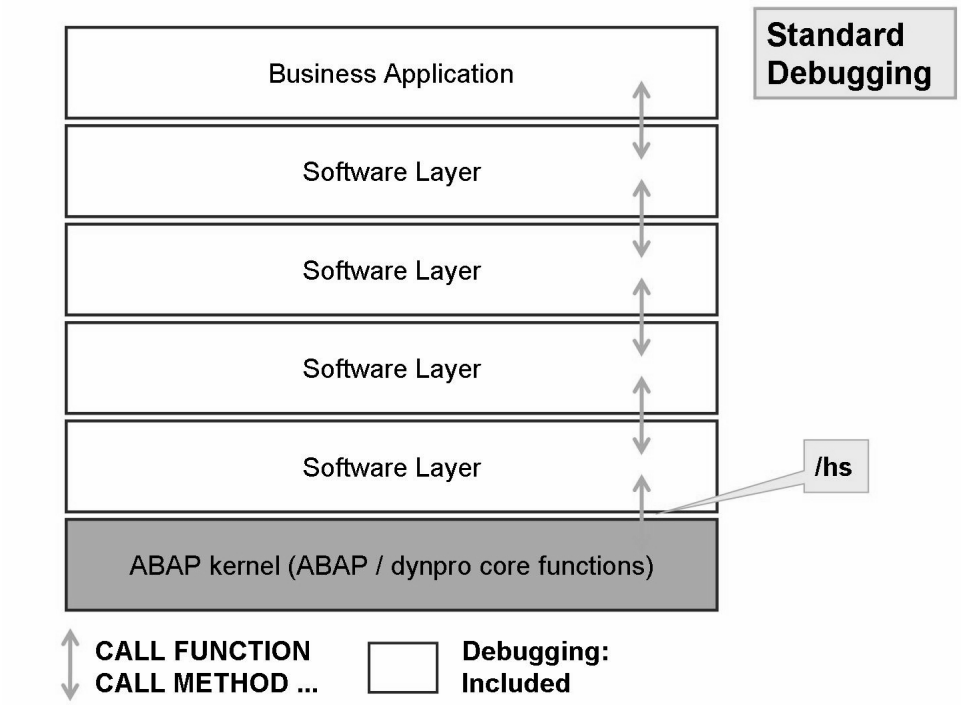


Figure 45: Debugging - all software layers are included

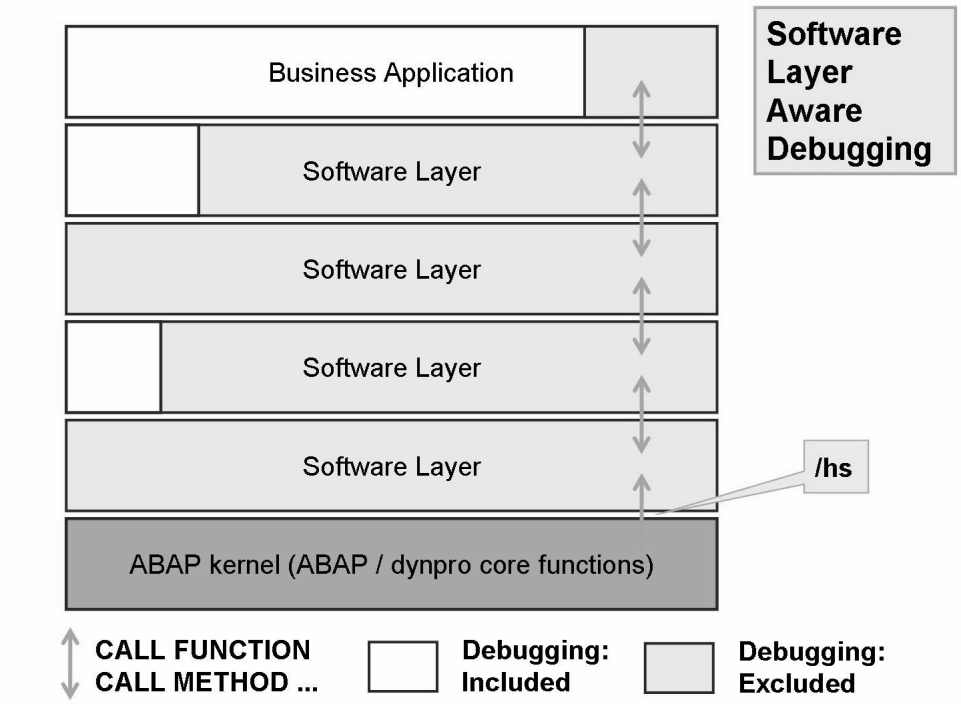


Figure 46: Software layer aware debugging - restrict debugging on object sets

This drawback is now addressed by the software layer aware debugging concept. In a debugger session, the user may switch on this debugging feature. The debugging process is guided by a debugging profile. The debugger profile references one or multiple object sets, each object set incorporating a selection of programming objects. The debugger profile defines settings for each of the object sets. These settings influence the debugging process (e.g. is object set visible in the dubbing process, or does the debugger stop when the next object does not belong to object set).

Object Sets

Object sets can be defined persistently. This way object sets may be reused in multiple debugger profiles. To create new object sets, or to display, change, copy, or delete existing object sets, the transaction SLAD is used.

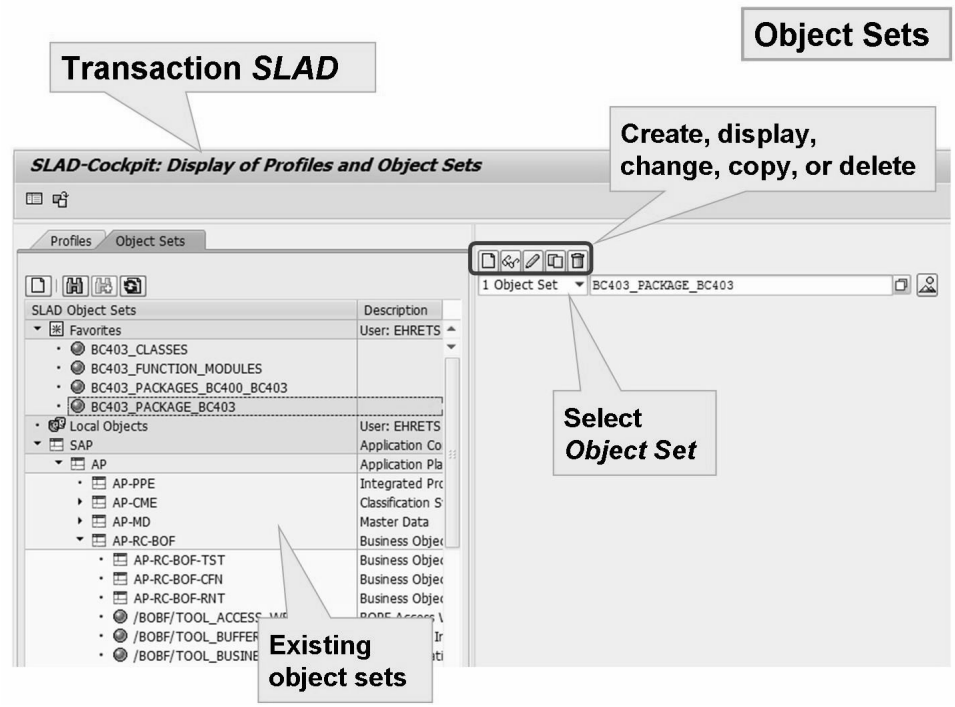


Figure 47: Object sets - transaction SLAD

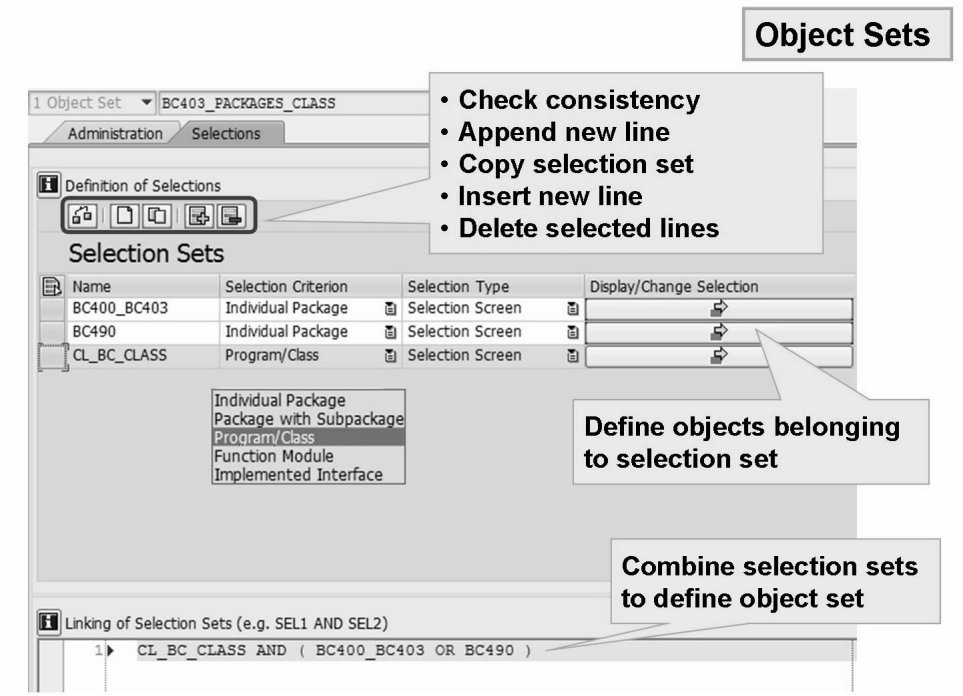


Figure 48: Combine selection sets to define object set

An object set combines one or multiple selection sets of programming objects using logical AND, OR, and brackets. Each selection set contains repository objects that are selected by the name of an individual package, a main package including sub packages, a program / class, a function module, or an implemented interface. Object names are entered using a standard select options dialog. The names may also be derived from the application hierarchy.

Example: An object set contains three selection sets.

- The first selection set (*A*) contains packages between BC400 and BC403.
- The second selection set (*B*) contains classes having a name beginning with CL_BC*.
- The third selection set (*C*) contains the package BC490.
- The selection sets are combined using the condition *B AND (A OR C)*.

This means the resulting object set consists of all classes having a name CL_BC* and belonging to the packages BC400 to BC403 or to the package BC490.

Note: Object sets are repository objects.

Debugger Profiles

Debugger profiles combine multiple object sets that are of interest in a debugger session. In addition, the profile defines settings for each of the included object sets. The settings are used to determine debugging behavior as follows:

- **Visibility**: The debugger stops when the relevant code is entered. If the code is visible, it can be debugged in normal steps (for example single steps). If the code is hidden, the debugger does not step in the code (comparable to system programs).
- **Point of Entry**: The debugger stops when the code of the specified object set is entered. This function is useful to debug the code in large steps, jumping from layer to layer or from component to component.
- **Point of Exit**: The debugger stops right after the code of the specified object set when the object set is exited. This function is useful to debug the code in large steps, jumping from layer to layer or from component to component.
- **Including System Code**: This function is used to include all the system programs of the object set.

Each debugger profile contains a predefined object set called ***<<%REST%>>***. This object set includes all ABAP code that is not part of the self-defined object sets of the profile. The debugger behavior concerning the ***<<%REST%>>*** object set may be specified or the ***<<%REST%>>*** object set may be deleted. This is especially useful if only the calls between the self-defined object sets or of interest.

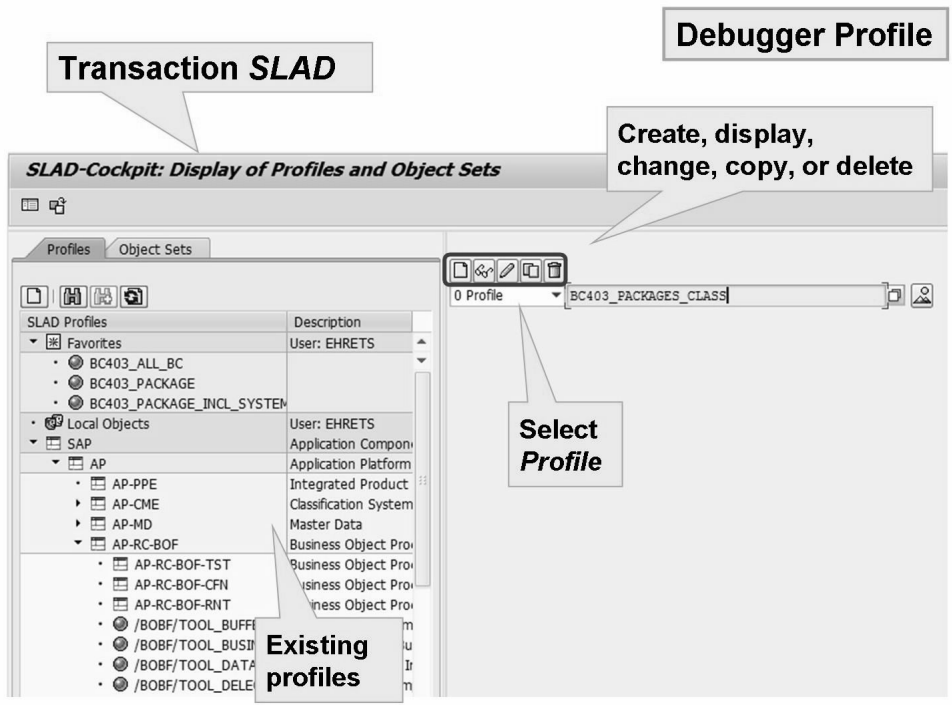


Figure 49: Debugger profiles - transaction SLAD

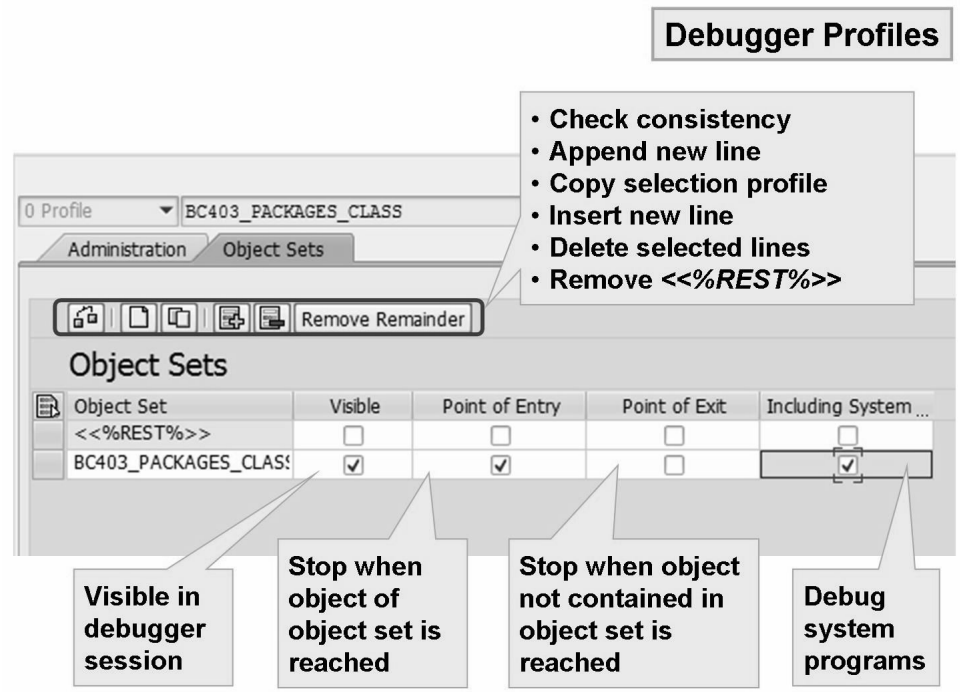


Figure 50: Debugger profiles - set object set attributes

Debugger profiles can be defined persistently. This is done via transaction SLAD. This transaction may also be used to display, change, copy, or delete existing profiles. In a debugger session, persistent debugger profiles can be loaded. Persistent debugger profiles cannot be altered in the ABAP Debugger. However, temporary profiles may be derived from persistent profiles. These profiles are dropped at the end of the debugger session.

Note: Persistent debugger profiles are repository objects.

Use Debugger Profiles in Debugger Session

Debugger profiles can only be loaded when using the new debugger. Proceed as follows:

- Click toolbar button labeled *Configure Debugger Layer* (STRG + SHIFT + F4).
- Mark checkbox labeled *Layer-Aware Debugger Active*.
- Select radio button labeled *Use of Predefined Object Set using Debugger Profile* if you want to load a persistent profile.
- Select the debugger profile:

  Enter a profile name in the *Profl.* field. Use the F4 help or select the button with the quick info text *Profile Library* to browse through the profile navigation tree.

- Activate the profile:

  If you click the *Continue* button, the profile is activated for the current session only. If you select the *Save Settings* button, the profile is activated each time the debugger starts.

When layer aware debugging is switched on, an additional toolbar button is displayed in the debugger session to activate / deactivate layer aware debugging. However, when starting a new debugger session, layer aware debugging is always active.

To switch of layer aware debugging, proceed as follows:

- Click toolbar button labeled *Configure Debugger Layer* (STRG + SHIFT + F4).
- Clear checkbox labeled *Layer-Aware Debugger Active*.
- Activate the profile:

  If you click the *Continue* button, the profile is deactivated for the current session only. If you click the *Save Settings* button, the profile is deactivated each time the debugger starts.

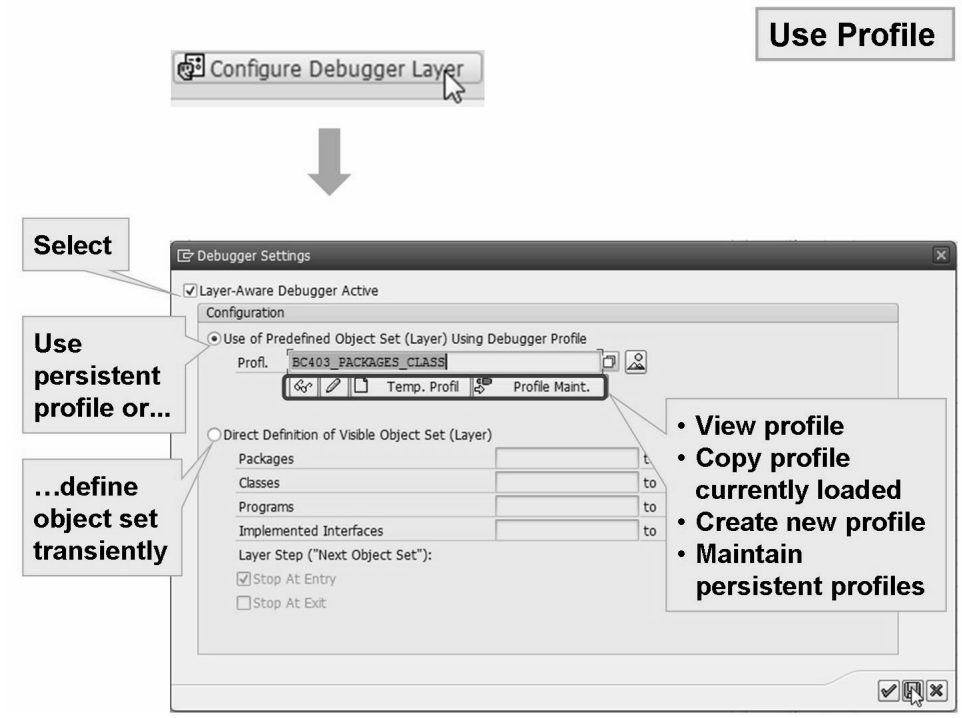


Figure 51: Use profiles in debugging sessions

To create a debugger profile in a debugger session, multiple options exist:

In the configuration dialog for layer aware debugging, one object set may be defined in the section related to the radio button *Direct Definition of Visible Object Set*. However, these settings are dropped at the end of each debugger session.

It is also possible to define a new debugger profile by using an the currently loaded persistent profile as a template (button labeled *Change*). Here the transaction SLAD is opened in a dialog screen. The new profile may be saved on the database persistently (*Save*), or it may be defined temporarily (*Continue*). In the latter case, the profile is dropped at the end of the debugger session.

Finally, the button labeled *Temp. Profile* is used to create a new profile either from the scratch or by copying any existing persistent profile. Again, the new profile may be saved on the database persistently (*Save*), or it may be defined temporarily (*Continue*).

Note: The configuration dialog for layer aware debugging is also included in the general debugger settings dialog.

Lesson Summary

You should now be able to:

- Define object sets and debugger profiles persistently using transaction SLAD
- Define debugger profiles transiently from a debugger session
- Use debugger profiles in a debugger session

Internal Use SAP Partner Only

Lesson: Request Based Debugging

Lesson Overview

In SAP Systems based on SAP Web Application Server 6.20 and later, external debugging can be used to debug applications that run in a user session different from the developer's user session. All application types that use the HTTP protocol have to be debugged that way (e.g. Web Dynpro, Business Server Pages, Web services, ICF services). Classically, external breakpoints are related to user names and they are defined on all application servers.

However, this breakpoint type does not fit all needs. While processing the application, the user name may be switched. This may happen if the application connects to another system to collect data via RFC.

A new flavor of external breakpoints - available as of SAP NetWeaver 7.02 - may be used to overcome these restrictions. In this lesson, the definition and usage of external breakpoints in request based debugging scenarios is discussed thoroughly.

Lesson Objectives

After completing this lesson, you will be able to:

- Define external breakpoints that are related to the terminal ID (TID)
- Use request based debugging with SAP GUI applications
- Use request based debugging with Web Dynpro for ABAP applications

Business Example

Example 1 (Web Dynpro applications):

An online store is implemented using a Web Dynpro application. You want to buy a book. You put it into the shopping basket and then confirm your online order. The confirmation is executed in a back-end system via RFC. However, during the confirmation, an error occurs. Thus you need to debug the function module triggered via RFC. Therefore you set a user breakpoint in this function module and restart the application. However, the application processing is not interrupted at the breakpoint. The reason for this is a generic user or user mapping. In this case the user breakpoint doesn’t work because it is set for your user.

Example 2 (customer support):

There are situations in which the user who sets a breakpoint and the user who sends an external request are two different end users. In addition, the request may be send from one computer (PC or laptop) while the breakpoint is set from another computer. You can experience such situations for example in support environments: A customer runs a Web Dynpro application that calls a remote

function module, which you in turn wish to debug locally on your own PC. It could also be that you need to logon into the customer network (with SAP GUI remote or with a Web browser via WTS) and initiate an HTTP or RFC request, which you need to debug locally on your computer.

Request Based Debugging

Request based debugging makes it possible to stop and debug specific HTTP or RFC requests that a specific end user (an operating system user at a PC) sends. In contrast to user based debugging, the following is assured:

- It does not make any difference, which application server processes the request.
- The user name is not of importance.
- Processing requests from other end users are not affected - which is essential in a productive system.

Request based debugging is therefore intended for debugging applications that are based on client-server technologies where a general system user processes the external requests of different clients. Requests can be isolated from each other and analyzed separately.

Example situation: A support employee debugs HTTP requests. Breakpoints are set in the remote user session (SAP GUI), while the browser is started on the WTS in the customer network.

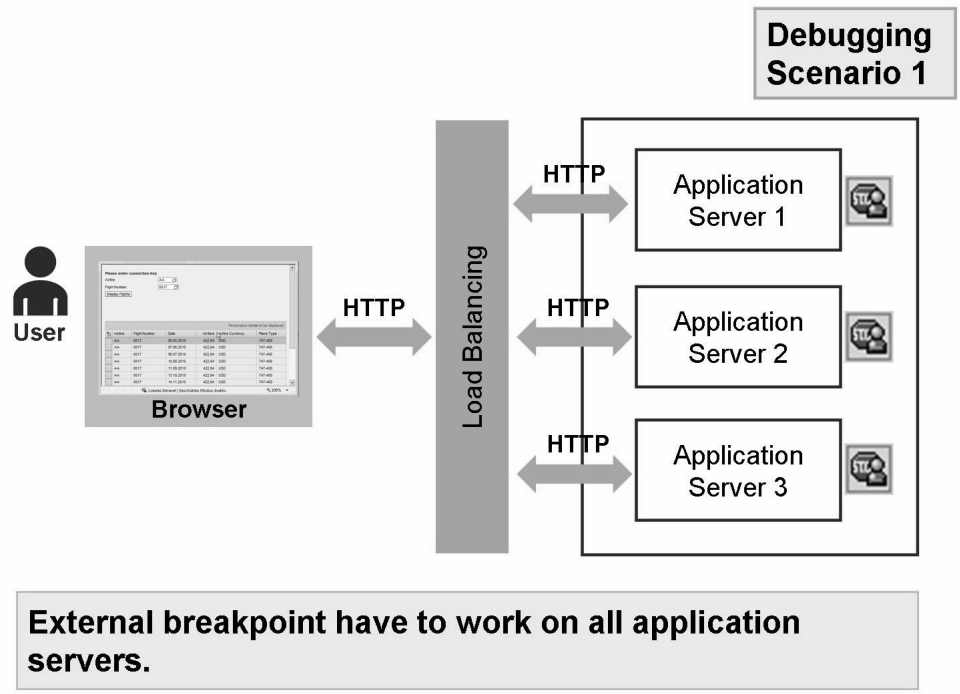


Figure 52: Request based debugging - requirements 1

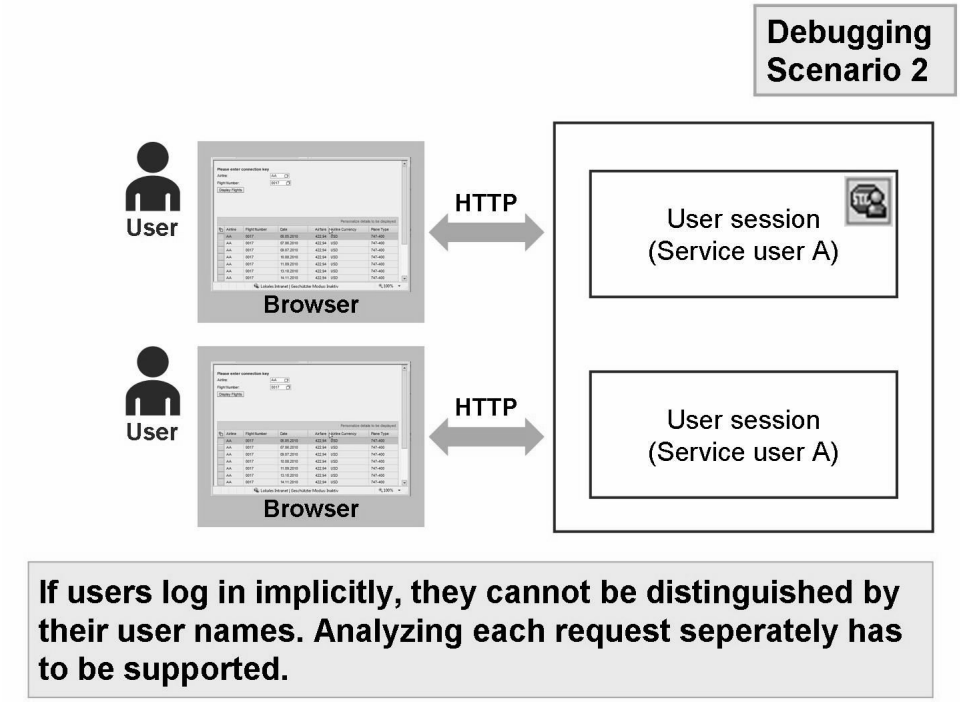


Figure 53: Request based debugging - requirements 2

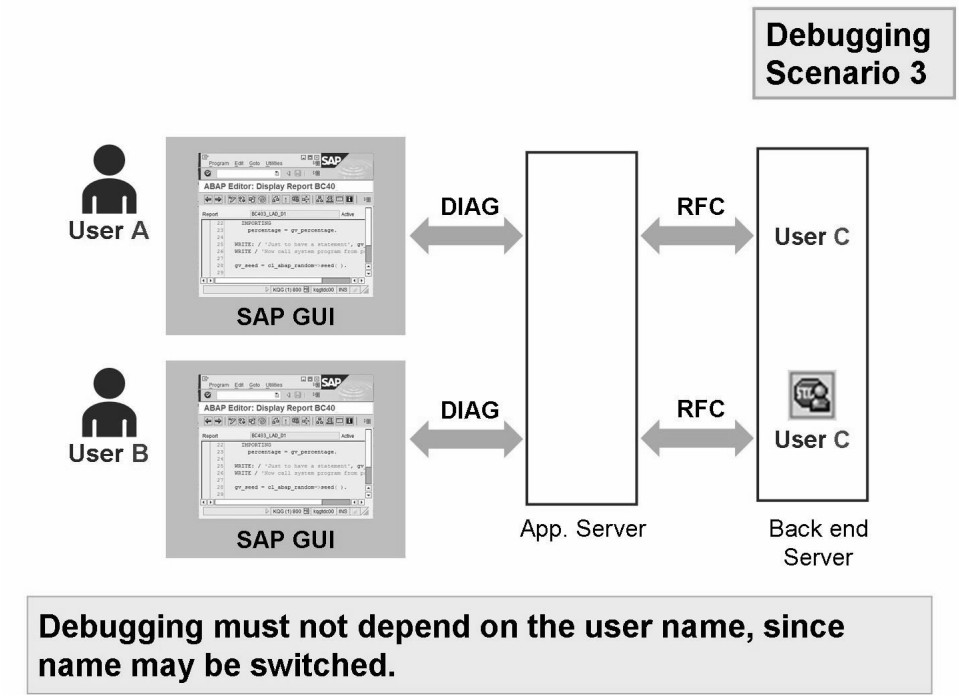


Figure 54: Request based debugging - requirements 3

Concept

Both, the external breakpoints and the requests to be tested are given a special identifier. This identifier is called ***terminal ID***. The ABAP Debugger only starts when the terminal ID of the current request matches the terminal ID of the external breakpoint that has been reached.

Switch on Request Based Debugging

To assign the generated terminal ID to external breakpoints, proceed as follows:

- In the Object Navigator, edit the user-specific settings for debugging (menu path *Utilities* → *Settings*, tab *ABAP Editor*, sub tab *Debugging*.
- Select radio button labeled *Terminal ID*. The generated terminal ID will be displayed right of the radio button

In the following, all external breakpoints are automatically identified with the terminal ID of the current **end user**. The end user who sends requests must actively associate the outbound requests with the terminal ID! If a request that is flagged with a terminal ID is processed over multiple ABAP systems, then the associated terminal ID is automatically forwarded on. By already setting an external breakpoint with the same terminal ID, the initial request to be tested can be processed over multiple systems.

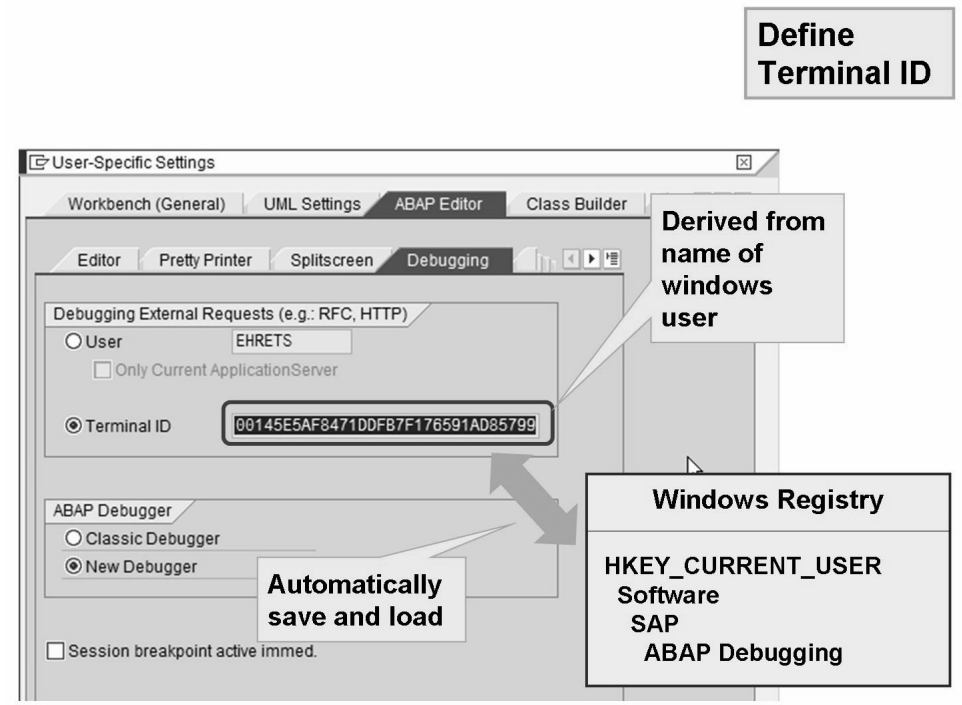


Figure 55: Switch on Request Based Debugging

Hint: The terminal ID represents the logon of the **Windows user** (end user) to the Windows operating system. It is generated and stored in the Windows Registry when request-based debugging is activated the first time (registry key: **HKEY_CURRENT_USER\Software\SAP\ABAP Debugging**). It remains valid until it is deleted manually from the Windows Registry.

Debug SAP GUI based Applications

If the end user who sets the breakpoints also sends off the request to be tested (from the same PC), only one action is required to send the terminal ID with all requests:

In the window of the SAP GUI that initiates the request, enter ***/htid*** in the command field. From now on, all requests initiated from this window (that is, from this external mode) are automatically given the terminal ID of the current end user. The system displays an appropriate message in the status bar.

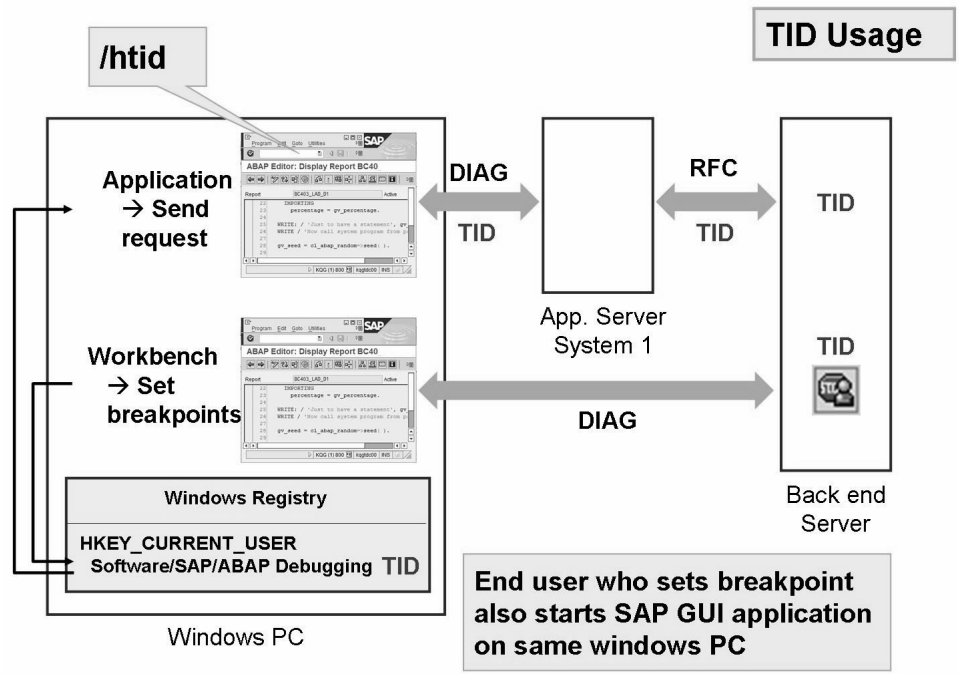


Figure 56: Set breakpoints and start application from same PC

If the end user who sets the break point and the end user who sends off the request are different, then the terminal ID related to the external breakpoints has to be attached to the request manually. This is done by entering ***/hset tid=<TID>*** in the SAP GUI command field (which will overwrite the terminal ID read implicitly from the Windows Registry). Here *<TID>* is the terminal ID associated with the external breakpoints.

From this point all requests initiated from this window of the SAP GUI (that is, this external mode) are associated with the terminal ID that is entered.

Note: To transfer the terminal ID to the person who starts the application, e-mail, phone, or other means may be used.

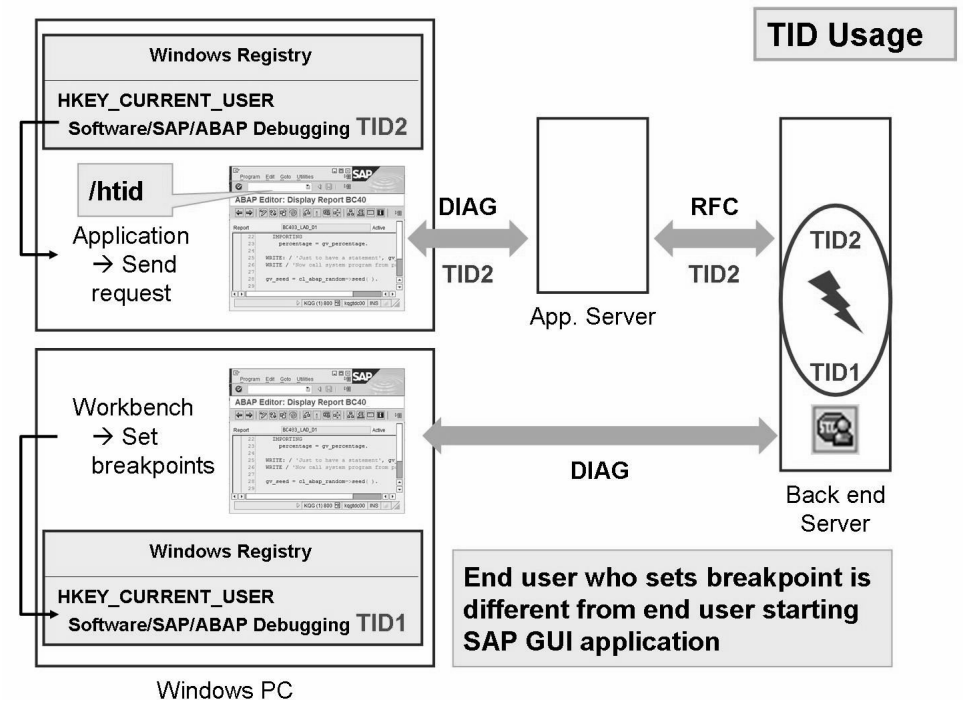


Figure 57: Set breakpoints on one PC and start application from other PC - problem

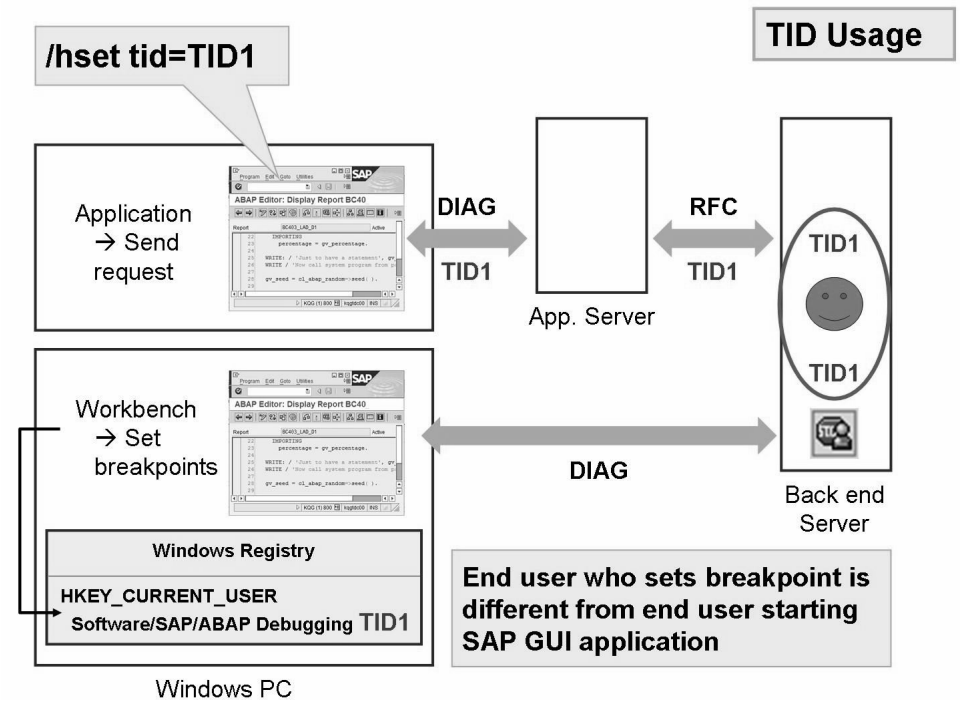


Figure 58: Set breakpoints on one PC and start application from other PC - solution

Note: The manual assignment of a terminal ID (*/hset tid=TID*) can be cleared by entering ***/hdel tid*** in the command field.

The current assignment of the terminal ID can be checked by entering ***/hget tid*** in the command field.

Debug HTTP based Applications

If an HTTP based application is to be debugged by the concept of request based debugging, the following steps need to be performed:

- Get the SAP HTTP plug-in for Microsoft Internet Explorer. For details, see **SAP Note 1041556**.
- Start Microsoft Internet Explorer using the SAP HTTP plug-in. A new browser window opens. Another window of the plug-in appears in the foreground.
- In the field labeled *Terminal-ID* enter the terminal ID related to the external breakpoints. This is not necessary, if you set the breakpoints on the same PC.
- Choose *Start Transaction* with a double-click. Do not close the window of the plug-in.
- In the browser window, start the request as normal.

As soon as an external breakpoint assigned to the same terminal ID is reached, the ABAP Debugger is opened in a new window on the developer's client.

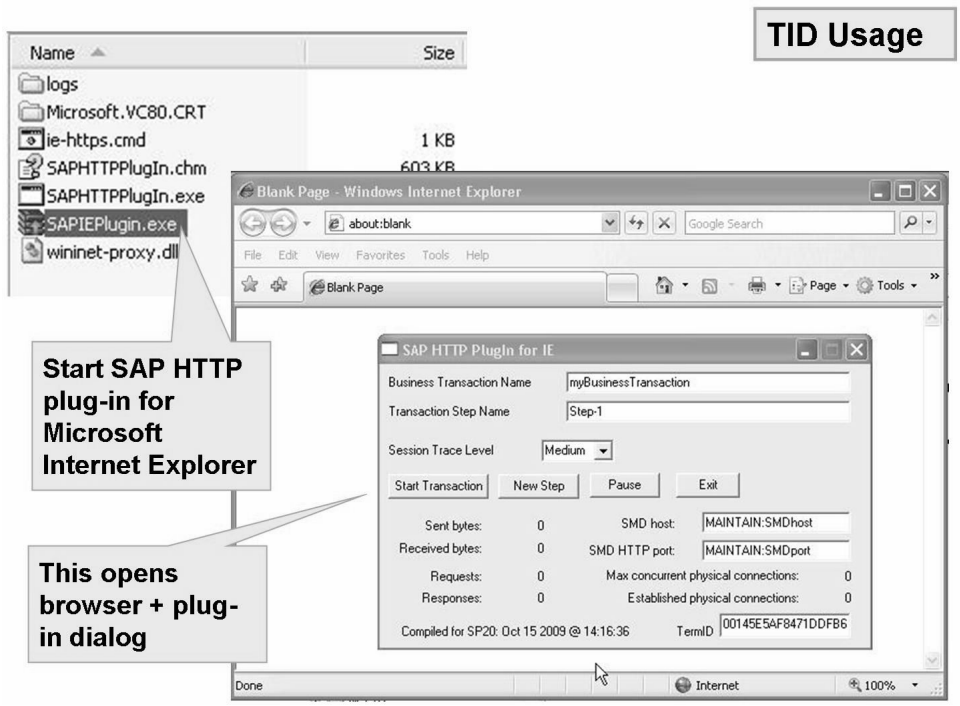


Figure 59: Start browser via SAP HTTP plug-in for Microsoft Internet Explorer

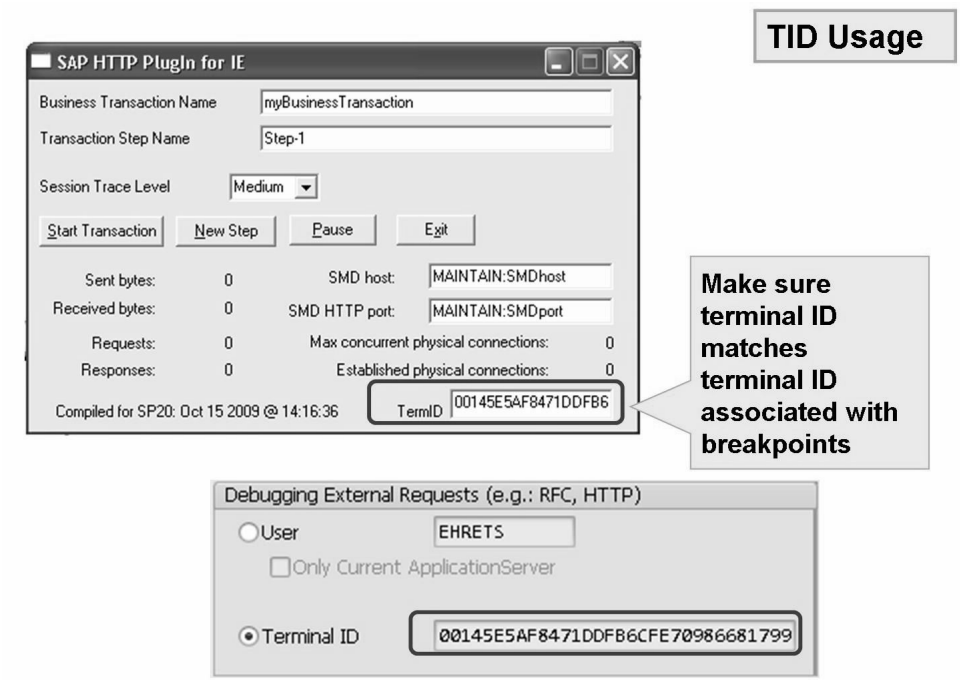


Figure 60: Set terminal ID to be attached to HTTP request

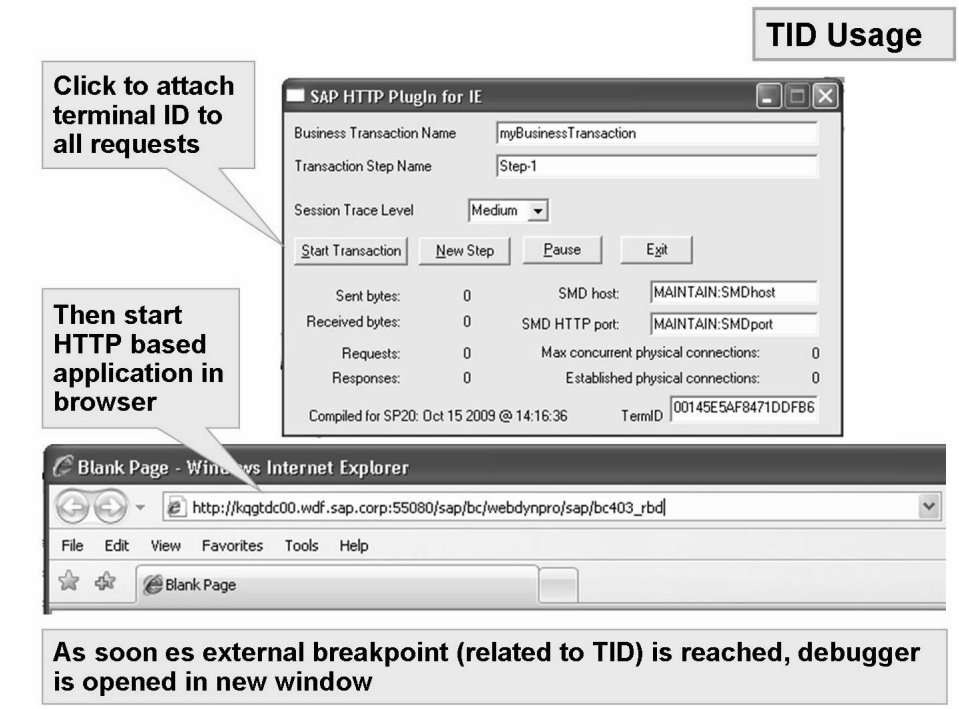


Figure 61: Start HTTP application

Lesson Summary

You should now be able to:

- Define external breakpoints that are related to the terminal ID (TID)
- Use request based debugging with SAP GUI applications
- Use request based debugging with Web Dynpro for ABAP applications

Lesson: Debugger Scripting

Lesson Overview

Debugger scripting is a new debugging feature available as of SAP NetWeaver 7.0 EhP 2. It may be used to automate many actions. Examples: Change the application context or change the execution of an application (e.g. simulate an authority issue), analyze complex application context and present it in an understandable manner, create custom traces, create custom breakpoints and watchpoints.

This lesson explains in detail how to create debugger scripts, how to use debugger scripts, and how to analyze trace files created by debugger scripts.

Lesson Objectives

After completing this lesson, you will be able to:

- Create debugger scripts
- Execute debugger scripts
- Create and analyze debugger trace files

Business Example

You would like to use debugger scripting to automate repetitive tasks. In addition you would like to create trace files documenting the debugging activities and the program flow or the call stack. Finally you would like to create sophisticated breakpoints or watchpoints that depend on complex requirements.

Overview

Debugger scripting is a new debugging feature available as of SAP NetWeaver 7.0 EhP 2. Debugger scripting lets you automate anything that you can do by hand in the new ABAP Debugger. In addition, debugger scripts allows you to create all kinds of trace files. The following tasks are covered:

- Analyze and modify the contents of variables and objects in the program.

  Example: Skip over failed `AUTHORITY-CHECK` statements with a script by stopping at each `AUTHORITY-CHECK`, running it, and resetting SY-SUBRC to 0.

- Perform any imaginable tracing that you might want to do:
  - Create the first automated complete trace of executed ABAP statements possible.
  - Trace the call stack in an ABAP application or in any part of an application. Tracing the call stack is very helpful to find out what to include in a profile for Layer Aware Debugging.
  - Define own traces with any trace message content (e.g. trace the value of a parameter that is passed up or down the call stack in your application to find out where it goes bad).
- Implement sophisticated breakpoints and watchpoints.

  Example: Combine the powers of the new ABAP breakpoints and watchpoints with the intelligence of scripts. Some system returns trash in an RFC data structure? No problem:
  - Run a script whenever the program in the debugger calls a function module via RFC.
  - Execute the RFC call.
  - Check the returned data in your script.
  - Write a trace or break execution when the bad data is returned.
- Control the execution of the program that is debugged.

  Example: Execute a statement at which you have stopped. Call a breakpoint to stop execution when your script finds something that should be analyzed interactively.

- Analyze and display data from complex data structures.

  Example: For tracing data generation or checking data consistency as your application runs, a script is the suitable tool. A script can read those nested attributes (e.g. an internal table row by row), check or trace them, and even stop execution if something goes wrong.

Debugger scripts for standard tasks, such as changing variable values to test error handling, are already available.

Debugger scripting uses the functions within the ABAP Debugger interface (ADI). The Debugger script is running on the debugger side. Therefore, it can only use the ADI functionality to manipulate the application, but it cannot directly influence it.

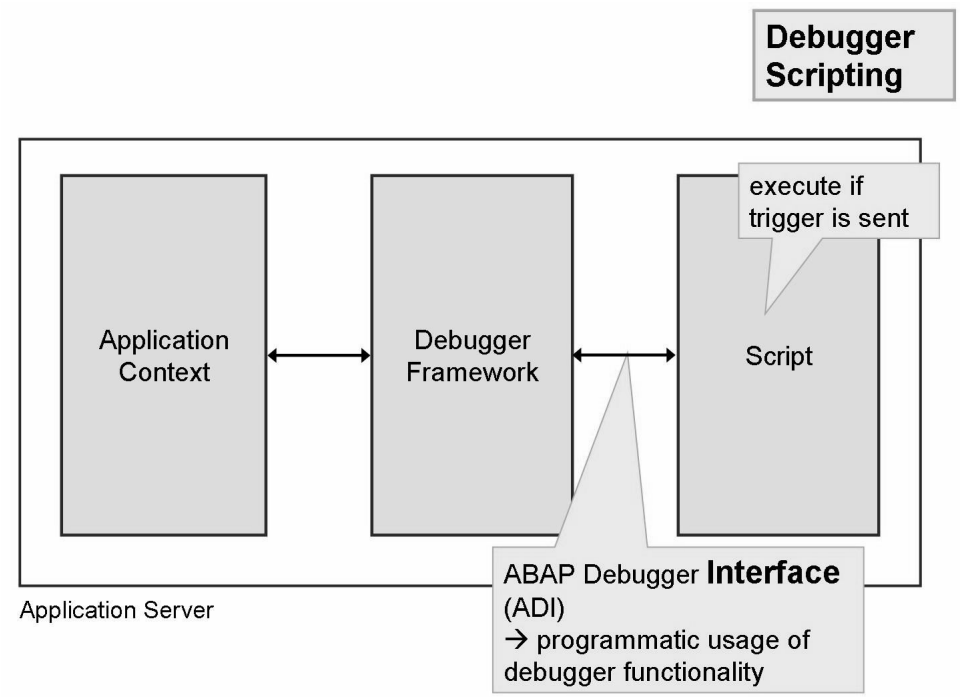


Figure 62: Debugger Scripting

A debugger script is a subroutine pool containing only the local debugger script class **LCL_DEBUGGER_SCRIPT**. This class contains the code of the debugger script. While debugging, the script code can be processed on request or it may be processed after predefined debugger events.

The local debugger class can be adapted by adding attributes and methods. Four public methods - inherited by the super class **CL_TPDA_SCRIPT_CLASS_SUPER** and redefined in the local class - are called by the framework:

- **prologue**: This method is called once, when starting the script processing. It is used by the framework to initialize the script technically.
- **init**: This method is called once, when starting the script processing (directly after the *prologue* method). It is used to initialize the script (e.g. create a user dialog to ask for data that is needed in the script).
- **script**: This method is called each time the script is triggered.
- **end**: This method is called once, at the end of the script processing. At this point, data could be displayed to the user, resources could be closed, or data could be aggregated before storing it.

Note: By default, the script is not paused at any time. However, a breakpoint may be defined in the script code via the statement `me->break( )`. In addition, the statement `me->raise_error( )` may be used to stop the script processing if needed.

Create and execute a Debugger Script

A debugger script can be created quickly right in the debugger. The *Debugger Scripting* debugger tool is available directly in the new ABAP debugger on the *Scripts* tab. However, to prepare scripts in advance, the transaction SAS (*ABAP Debugger Scripting and Tracing*) can also be used (tab *Script Editor*).

Note: Preparing a script with transaction SAS has the advantage that the debugger work process is not blocked unnecessarily long.

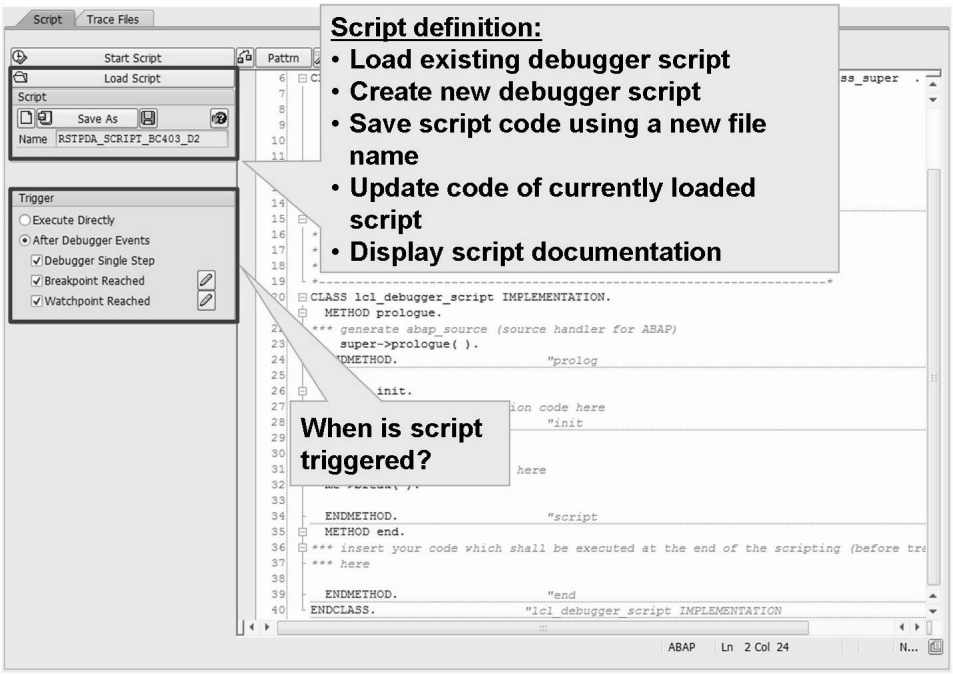


Figure 63: Debugger Scripting tool

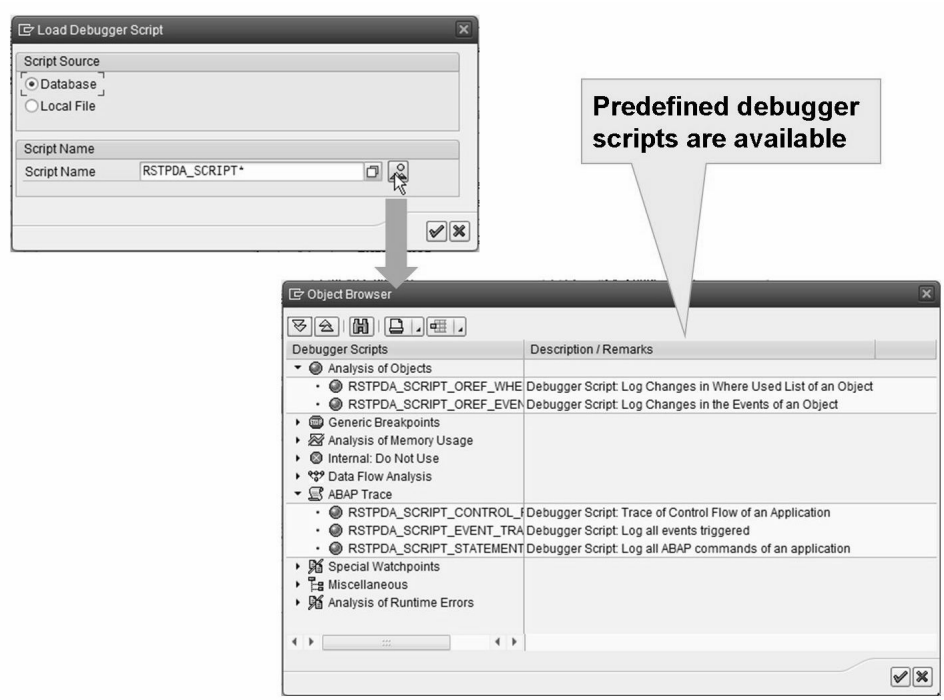


Figure 64: Load predefined debugger scripts

Initially, the *Debugger Scripting* tool displays the source code of a template script. This template contains empty implementations of the basic script methods. A new script may be developed based on this template or by loading an existing script, changing it, and subsequently saving it using a different name. The script may be saved on the database or in the file system.

Note: Technically, a debugger script is a subroutine pool (however, a new program type *X* has been created for subroutine pools containing debugger scripts). Thus, it may be displayed, changed, copied, or deleted in the Object Navigator. In addition, it may be documented. This program documentation may then be displayed in the *Debugger Scripting* tool by clicking the button displaying a hand and a question mark and which is labeled *Help for Script*.

The group labeled *Trigger* is used to define when the script code is executed. The following settings are possible:

- **Execute Directly**: If the button *Start Script* is pressed, the script runs immediately and is executed only once.
- **After Debugging Events**: The script is processed each time a certain debugger event is reached.
  - **Single Step in Debugger**: after each debugger step
  - **Breakpoint Reached**: after any script breakpoint is reached.
  - **Watchpoint Reached**: after any script watchpoint is reached.

To define which breakpoints trigger the execution of the debugger script, the button displayed right of the corresponding checkbox has to be clicked. A dialog opens below the *Debugger Scripting* tool. All breakpoints that should trigger the script execution have to be defined on the tab *Breakpoints*.

To define which watchpoints trigger the execution of a debugger script, proceed as described above. However, in the dialog displayed below the *Debugger Scripting* tool, choose the tab *Watchpoints*.

Note: Script breakpoints and script watchpoint trigger the script. The application processing is not interrupted.

Execution of the Debugger Script

Debugger scripts can only be started in a debugger session by clicking the *Start Script* button in the *Debugger Scripting* tool. The program will stop at all breakpoints or watchpoints set in the source code of the debugged program or set in the debugger script via the statement `me->break( )`. In this case, all of the features of the debugger can be used to analyze the debugged program.

Note: To debug the script code, the `break-point` statement is used.

The trigger setting influences the execution of the script as follows:

If the trigger is set to *Execute Directly*, the script runs once. If trace files functions are used in the script code, then each script execution defines a new trace file. The statement `me->break( )` in any of the script methods does not interrupt the processing of the script. After the execution of the script, debugging is continued without using scripts. However, if *Continue (F8)* is selected and the *Script* tab is the active tab, then a dialog screen pops up. Here, the user may decide, if the script is processed one more time, or if debugging is continued without using scripts.

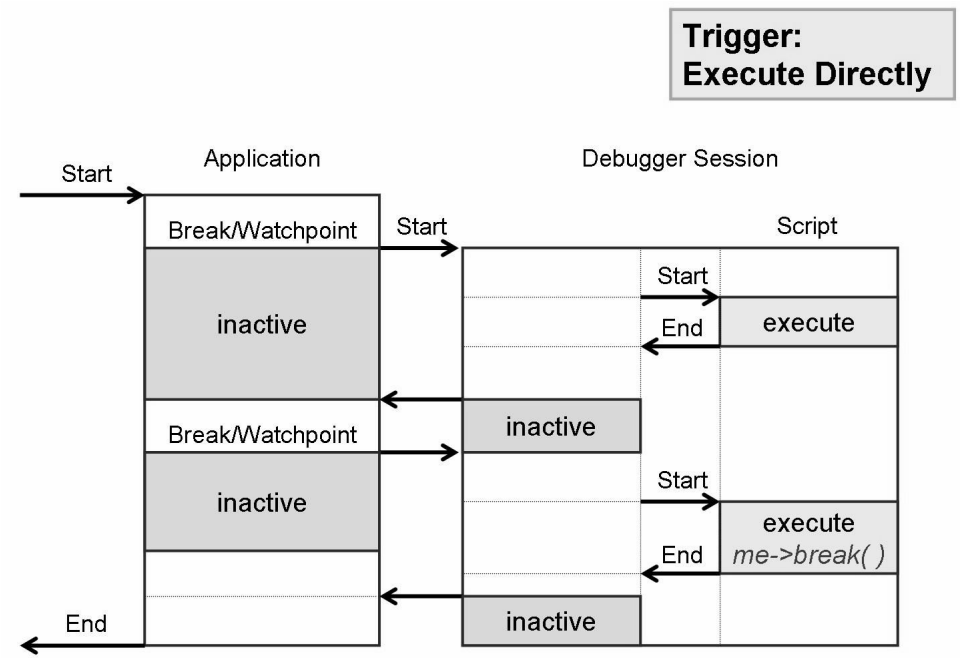


Figure 65: Program flow - execute script directly

If the trigger is set to *After Debugging Event* and the script is started, the script is executed each time the trigger event is reached. If trace files functions are used in the script code, then each script execution writes in the same trace file. The script is not stopped automatically before the debugger session is closed.

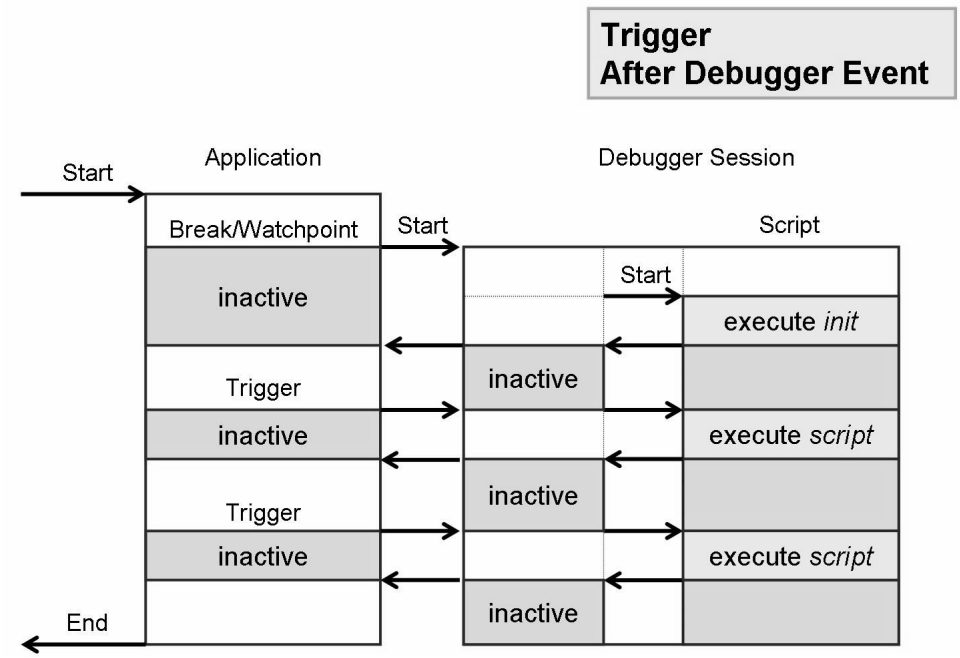


Figure 66: Program flow - execute script when trigger is reached

Stopping the script on request is only possible, if the debugger session gets the control. This is the case, if a breakpoint or watchpoint defined in the application is reached. This is also the case, if a breakpoint defined in the script code - by the statement `me->break( )` - is reached. In addition, the debugger is displayed, if */h* is entered in the command field of the application and if the application is then continued.

If a script is running, the content of the *Script* tab in the debugger changes. Now only two buttons are displayed, to stop the script and to continue the script. If the script is stopped (*Exit Script*), debugging is continued without using scripts. If *Continue Script* is selected, the script is executed again each time a trigger is reached. If one of the standard debugger functions (*Single Step (F5)*, *Execute (F6)*, *Return (F7)*, or *Continue (F8)*) is selected, debugging is continued but the script is not executed even if a trigger is reached. However, the script is not stopped, so it may be continued later on.

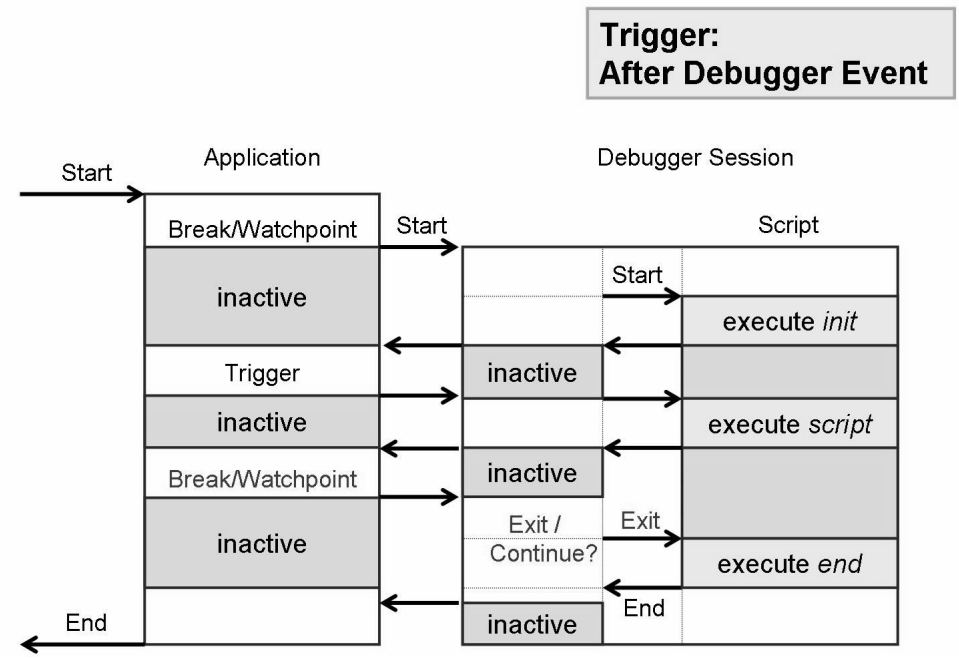


Figure 67: Program flow - end scripting on request (1)

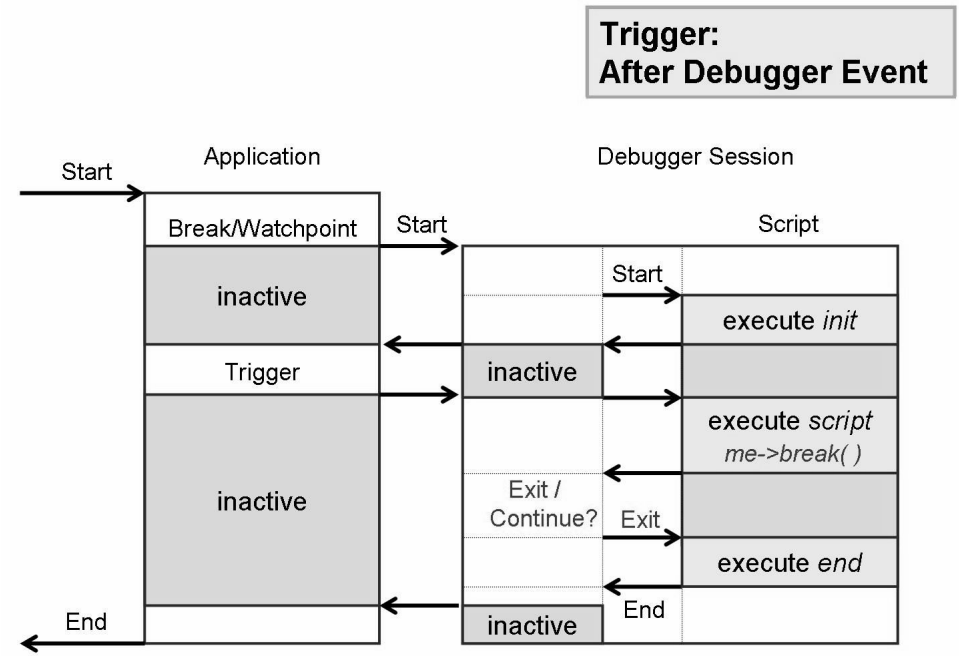


Figure 68: Program flow - end scripting on request (2)

Note: To terminate the script without user interaction, the method `me->raise_error( )` may be called in the debugger script.

Access the ABAP Debugger Interface (ADI)

The complete functionality available in the debugger interface can be accessed by the script code as follows:

1. Most of the ADI functionality can be accessed by static components defined in classes starting with **CL_TPDA_SCRIPT_**.
2. Some methods and attributes are defined as instance components. These components can be accessed via the instance of the local script class (*me->*). The local script class inherits from the super class CL_TPDA_SCRIPT_CLASS_SUPER. At runtime, the local class is instantiated by the debugger framework.

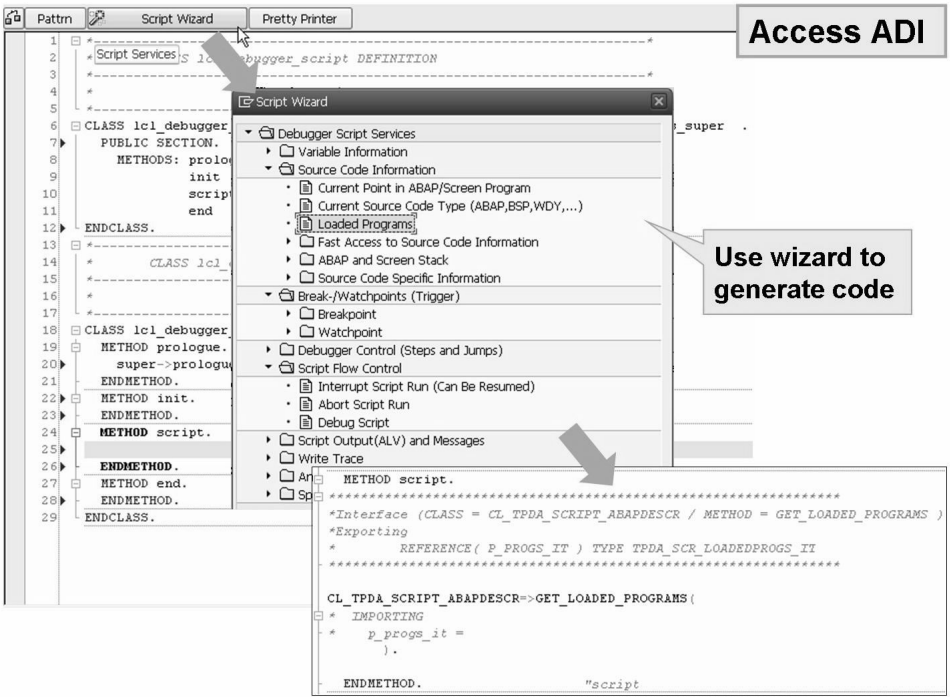


Figure 69: Generate code to access the ABAP Debugger Interface (ADI)

The developer of the script does not need to know all of these classes. A wizard may be used to generate all important method calls.

Create and analyze a Debugger Trace File

The scripting functionality can be used to produce traces. Functions defined in the class **IF_TPDA_SCRIPT_TRACE_WRITE** may be used to write the last ABAP statement, the call hierarchy, or a user defined content to a trace file.

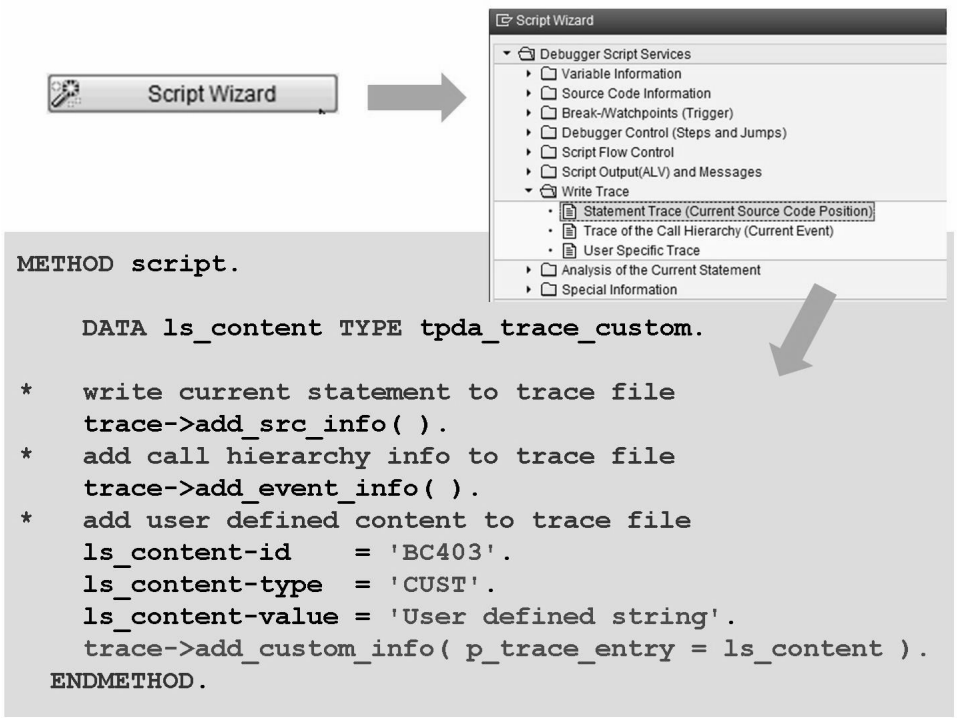


Figure 70: Create a debugger trace file

As soon as the script creating the trace file is ended, the file appears in the list of trace files on the *Trace File* tab page. The list is sorted according to time stamp. The newest trace appears on top. For each trace, the following functions are available:

- **Display:**

  The administrative data of the selected trace file is displayed in a dialog screen. If the dialog screen is closed by clicking the *Continue* button, trace file details are displayed on a follow up screen. The details to be displayed can be selected via the radio buttons located in the group labeled *Components*.

  In addition, the button labeled *Start Standalone Trace Analys.* may be used to analyze the trace file with transaction SAS (started in a new main session).

- **Delete:**

  The selected trace file is deleted.

- **Start analysis in a new session:**

  Transaction SAS is started in a new main session.

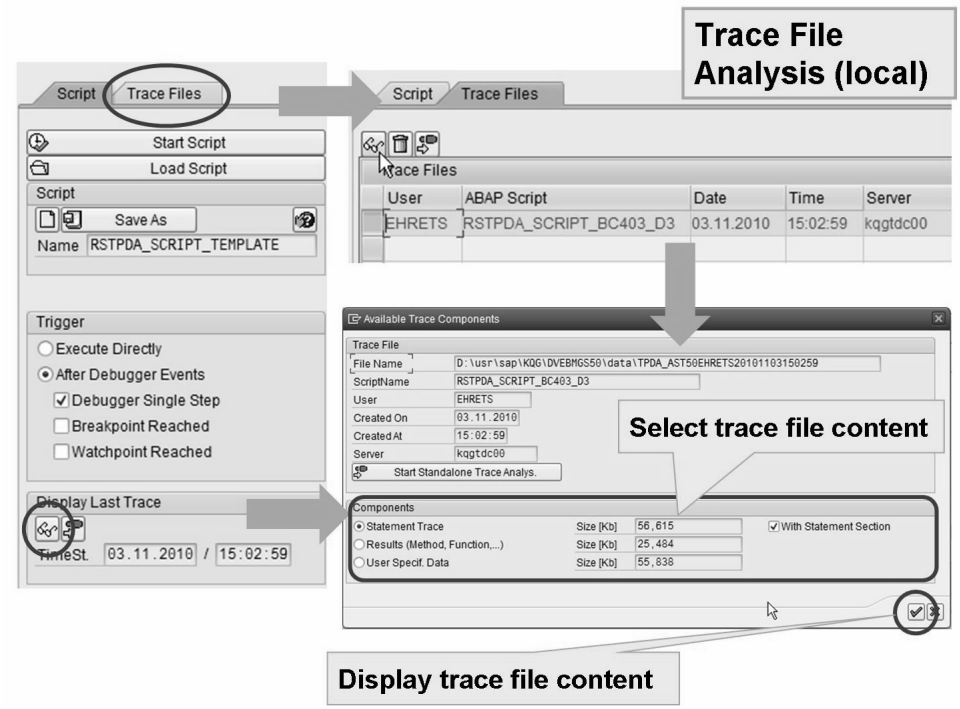


Figure 71: Trace file analysis - in Debugger Scripting tool

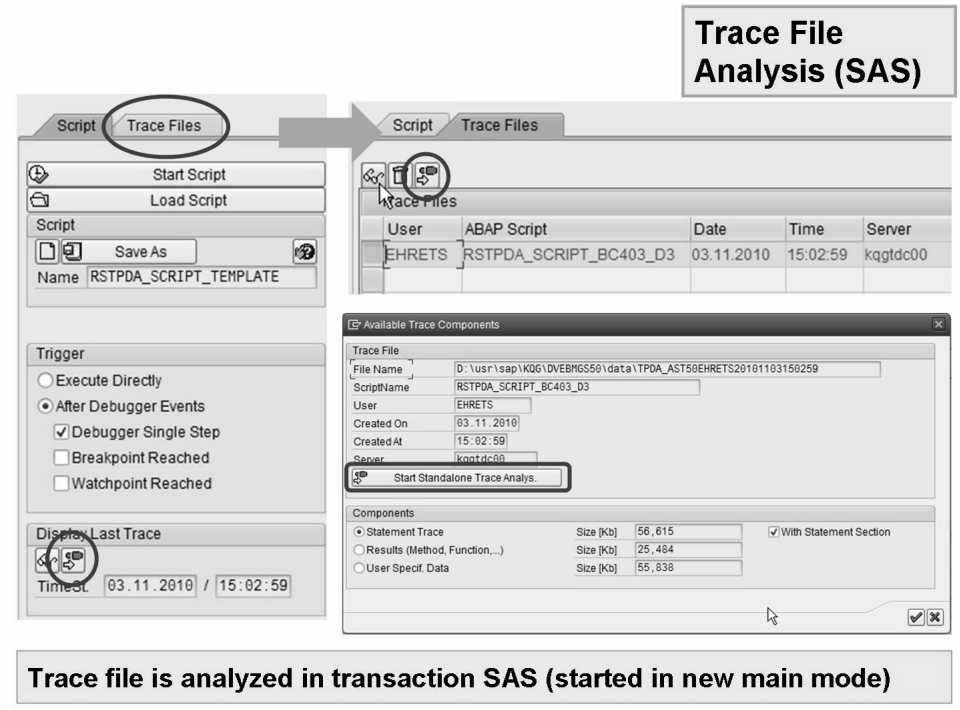


Figure 72: Trace file analysis - start transaction SAS

The transaction SAS is used to analyze in detail trace files produced with the *Debugger Scripting* debugger tool. The transaction is an auxiliary tool to the *Debugger Scripting* tool.

The main screen displays a tabstrip with three tab pages:

- **Trace Files**: The tab contains two screen areas - *Own Trace Files* and *Trace Files of Other Users*. Each screen area contains a list of trace files and related administrative information. The files are sorted according to time stamp, with the newest on top. Double-clicking a file from the list opens the trace file on the *Trace Display* tab.
- **Trace Display**: On this tab, information about the trace file is displayed. The *Trace Selection* screen area contains the *Statements*, *Hierarchy*, and *Custom* pushbuttons. Clicking a pushbutton displays the respective type of trace file. If the trace file is a combined trace, more than one pushbutton is active.
- **Script Editor**: This tab contains all the functionality of the Script tab of the *Debugger Scripting* tool, except for the Save and Start Script functions.

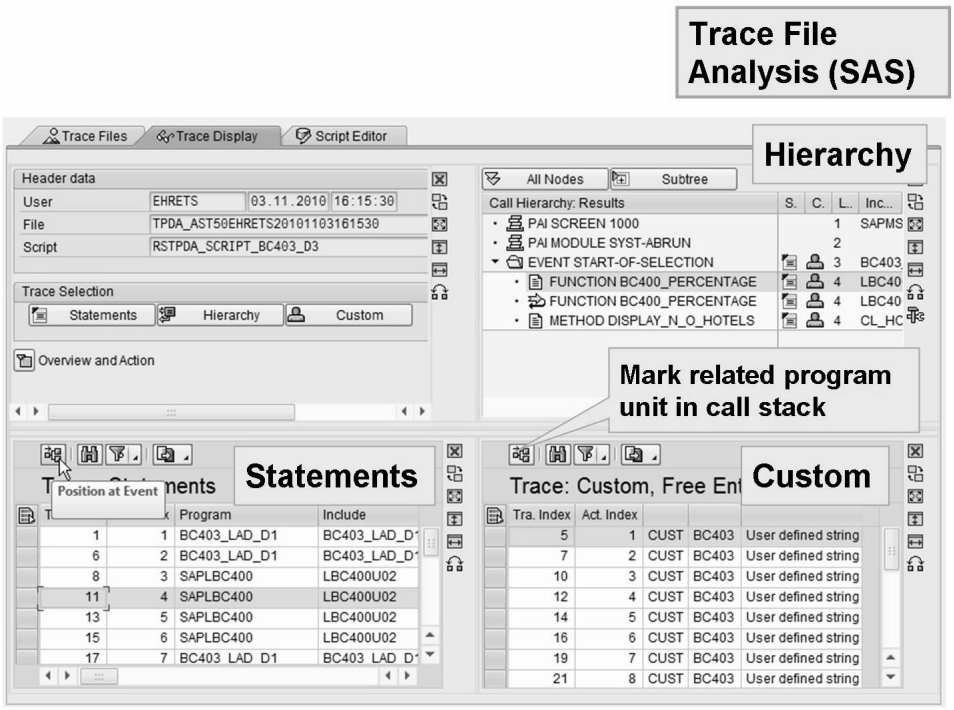


Figure 73: Trace file analysis - transaction SAS

Script Examples

In this section typical examples of how to use debugger scripting are discussed.

Automation

Debugger scripting may be used to automate repetitive tasks observed in the debugging process. In this case, script breakpoints or script watchpoints can be defined in the *Debugger Script* tool, which serve as a trigger for executing the script. These breakpoints do not interrupt the processing of the application. However, every time the script is triggered, the program status can be checked, variable values can be read and changed, the next program statement can be executed, and so on.

In the following example, each row of an internal table containing flight data is checked. If the occupation of a flight is too low, an information message is displayed. When debugging the program, the debugger stops each time a message is displayed. Thus the user who is debugging the program has to press *Continue* as often as messages are displayed.

This can be overcome by a script. A trigger breakpoint is defined for the line containing the calculation of the occupation. The script checks the current value of the occupation. If the logical condition for sending an info message is fulfilled, the script changes the occupation in a way that the logical condition is not fulfilled any more. In addition, the message text is reconstructed and written in a user trace file.

The following methods are used in this script:

- **debugger_controller->debug_step**: Process debugger command *Single Step (F5)*, *Execute (F6)*, *Return (F7)*, or *Continue (F8)*.
- **cl_tpda_script_data_descr=>get_simple_value**: Get value of simple variable (as string).
- **cl_tpda_script_data_descr=>change_value**: Change value of simple variable.
- **trace->add_custom_info**: Write customer defined data in user trace file.

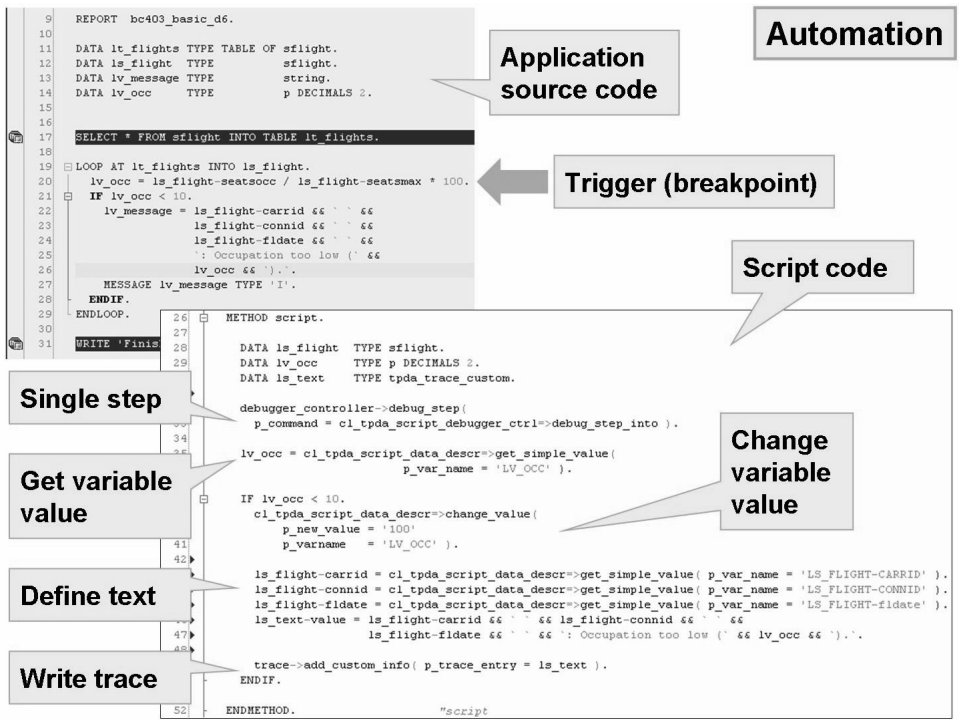


Figure 74: Example - automation task (change variable value to continue)

To use the script, two breakpoints need to be set in the source code. The first breakpoint has to be defined anywhere before the trigger. This breakpoint is used to start the debugger session, to load the script, and to start the script. The second breakpoint has to be defined anywhere after the loop. This breakpoint is needed to end the script and thus to finish the trace file.

Interactive Scripts

A script is defined by the source code of a local class. Thus, the statements used in the script code are not limited to the method calls offered by the ABAP Debugging Interface. Almost any of the capabilities offered by ABAP may be embedded in the code.

In the following, the example discussed above is extended. When starting the script, the user should have the opportunity to decide whether the messages are displayed and whether a trace file is created. This dialog should only be displayed once. Thus, the code is defined in the method **init** of the local script class.

To generate the dialog, the function module POPUP_GET_VALUES may be used. This function module creates a dialog containing as many parameter fields as lines are contained in the table parameter FIELDS. For each field the field length and the field type is obtained from a structure field or from a transparent table field defined in the ABAP Dictionary. It is not possible to define this information independent from the Dictionary. Thus, to create two simple boolean fields, we

have to refer to any structure field or transparent table field having an appropriate type. The user input can be read from the table parameter FIELDS after the dialog has been closed.

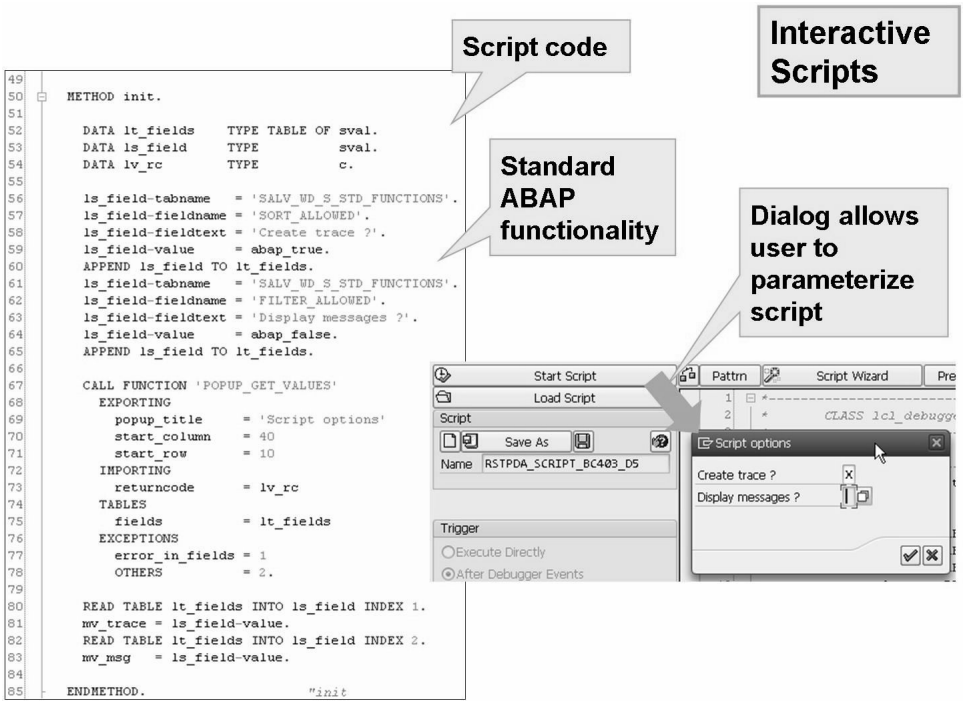


Figure 75: Interactive scripts - definition of dialog

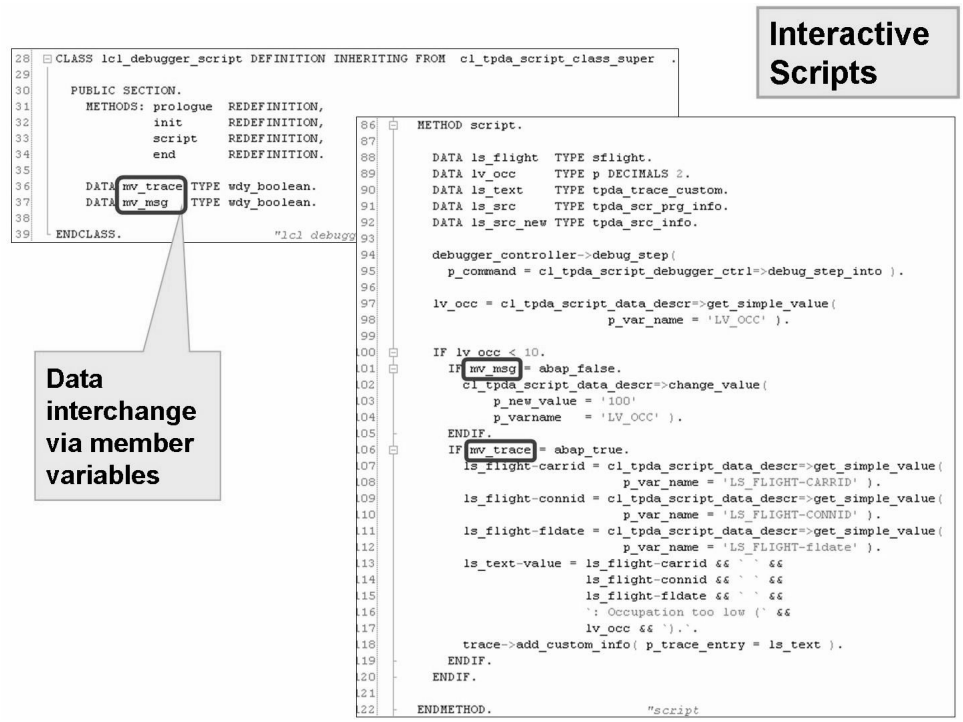


Figure 76: Interactive scripts - interchange data between methods

The user input has to be accessed in the method **script**. Thus, this information has to be stored in member variables.

Define Breakpoints at Runtime

ABAP already offers an extensive set of standard breakpoints that can be flexibly applied. Even so, there is always a need for an even more intelligent breakpoint if a difficult problem has to be analyzed. With debugger scripting, any combination of breakpoint functionality can quickly be implemented.

In the following example, the processing of an application should be stopped each time a `LOOP...ENDLOOP` statement is reached and if the internal table processed in the loop has a certain name (e.g. GT_FLIGHT). In addition, the application should only be interrupted once per find spot. This task cannot be fulfilled with standard breakpoints (breakpoint at ABAP statement...). However, with debugger scripting this is no problem.

All code is defined in the script method. The code contains the following sections:

First, the current line of the application source code is scanned. Here, the factory method **scan** of class **CL_TPDA_SCRIPT_SCAN** is called. This method returns an instance of the scan class. If the current line contains a `LOOP` statement, the reference may be casted to the respective class **CL_TPDA_SCRIPT_SCAN_ITAB_LOOP**. If the cast is successful, the method **it_tables** is used to get back the names of all internal tables related to the loops

of the current line. Next, the names of the internal tables are compared with the search pattern (e.g. GT_FLIGHT). If the search pattern is contained, the processing of the application is interrupted.

An additional coding section assures that the application is interrupted only once per find spot. To implement this requirement, the line number of each find spot is stored in an internal table (member variable). Before the processing is stopped, the line number of the current find spot is compared with the line numbers found before. The program is only interrupted, if a new find spot is reached.

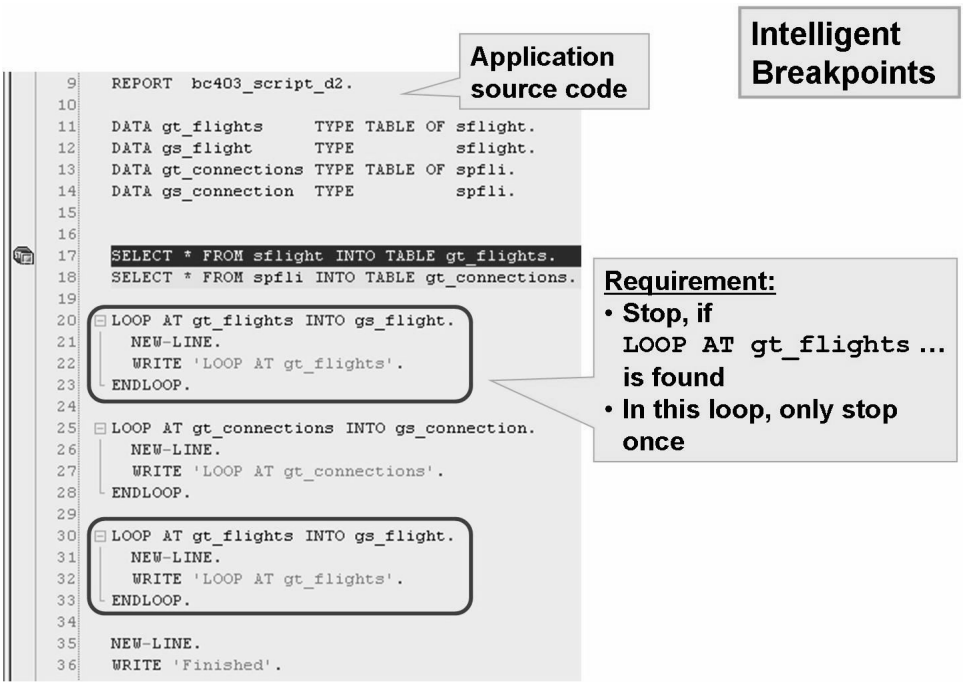


Figure 77: Intelligent breakpoints (1)

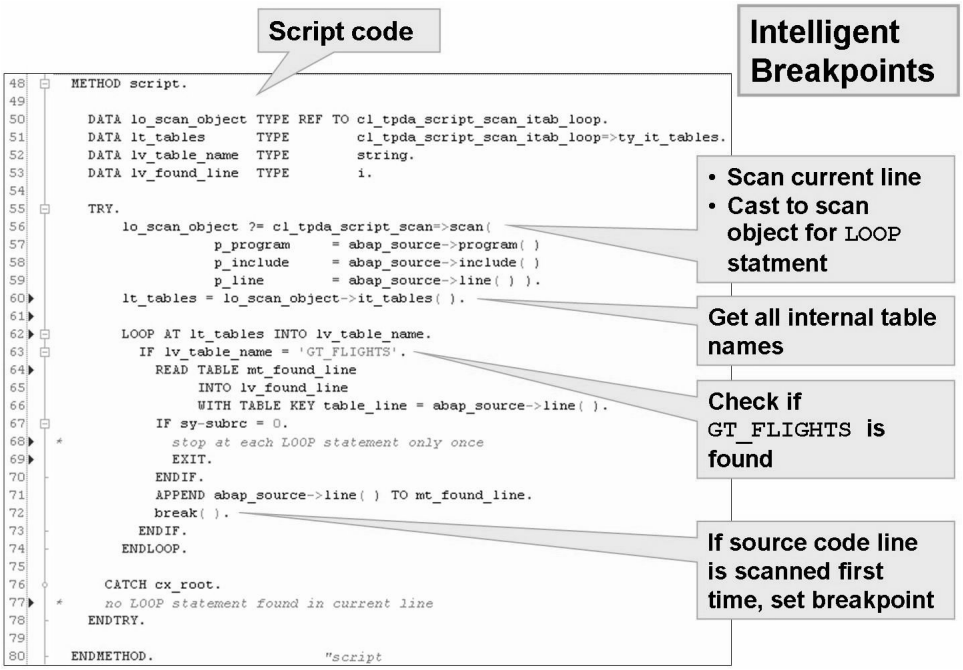


Figure 78: Intelligent breakpoints (2)

To start the script, a breakpoint needs to be set at the beginning of the application source code. Different kinds of triggers may be used to enter the script: Either a trigger of type *Debugger Single Step*, or (better) a script breakpoint at the statement `LOOP AT`.

Lesson Summary

You should now be able to:

- Create debugger scripts
- Execute debugger scripts
- Create and analyze debugger trace files

Unit Summary

You should now be able to:

- Define object sets and debugger profiles persistently using transaction SLAD
- Define debugger profiles transiently from a debugger session
- Use debugger profiles in a debugger session
- Define external breakpoints that are related to the terminal ID (TID)
- Use request based debugging with SAP GUI applications
- Use request based debugging with Web Dynpro for ABAP applications
- Create debugger scripts
- Execute debugger scripts
- Create and analyze debugger trace files

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Course Summary

You should now be able to:

- Start and end a debugger session
- Configure the debugger layout
- Use all kinds breakpoints and watchpoints
- Explore data objects
- Analyze the memory consumption of data objects and programs
- Use all kinds of debugger tools
- Restrict debugging on predefined software layers
- Debug applications involving HTTP and RFC calls and user switches
- Create and use debugger scripts

Index

Internal Use SAP Partner Only

Feedback

SAP AG has made every effort in the preparation of this course to ensure the accuracy and completeness of the materials. If you have any corrections or suggestions for improvement, please record them in the appropriate place in the course evaluation.

Internal Use SAP Partner Only