BC ABAP Workbench Tools

HELP.BCDWBTOO

Release 4.6C

Copyright

© Copyright 2001 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of
Microsoft Corporation.

IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.

ORACLE® is a registered trademark of ORACLE Corporation.

INFORMIX®-OnLine for SAP and Informix® Dynamic Server™ are registered trademarks of Informix Software Incorporated.

UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of the Open Group.

HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium,
Massachusetts Institute of Technology.

JAVA® is a registered trademark of Sun Microsystems, Inc.

JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, SAP Logo, R/2, RIVA, R/3, ABAP, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax
	Tip

Contents

BC ABAP Workbench Tools

BC ABAP Workbench Tools

Topics in this Documentation

New Features in Release 4.6C

The principal new feature in Release 4.6C is the integration of Internet services in the ABAP Workbench using a new tool called the Web Application Builder. As well as this, wizards for creating table controls and tabstrip controls have been introduced into the Screen Painter. There are also additional navigation functions and new features in some of the individual tools and in Business Add-Ins.

The New Features

Integrating Internet Services in the Object Navigator [Page 51]	
Screen Painter	
	▪ Tabstrip Control Wizard [Page 297]
	▪ Table Control Wizard [Page 303]
	▪ References to the ABAP data type STRING are now allowed for input/output fields on screens. Refer to the Name attribute in the General Attributes [Page 336] section.
	▪ New screen attribute: Hold scroll position. For further information, see Creating Screens [Page 317].
Additional Functions in the Object Navigator	
	▪ Persistent worklist [Page 45] in the Object Navigator
	▪ Insertion of statement patterns using drag and drop. For further information, refer to Inserting Patterns Using Drag and Drop [Page 125].
	▪ History for object list navigation.
	▪ Showing and hiding the navigation area [Page 40].
ABAP Editor	

	Additional functions avaiable in the context menu of the frontend editor: ▪ Navigation to a particular line. For further information, refer to Navigating in the Source Code [Page 106]. ▪ Buffer and block operations. See also: Using Buffers [Page 117].
Transaction Classification	
	Transactions can be classified as *Professional User Transactions* or *Easy Web Transactions*. The *GUI Support* group box allows you to specify the GUI that is launched when a user starts the transaction from the mySAP.com workplace.
New Functions in **Business Add-Ins [Page 527]**	
	▪ Create example and default implementations For a defined Business Add-In, you can provide an example and a default implementation with the corresponding source code.
	• Extendable filter types If implementations of filter-specific definitions of Business Add-Ins were only possible for existing filter values, it is now possible to create implementations for filter values that do not yet exist.

Object Navigator

Use

The Object Navigator is a central point of entry into the ABAP Workbench. It is the successor of the Repository Browser, and you can access it using transaction code **SE80**.

You use the Object Navigator to organize your programming in an integrated development environment. It allows you to create, change, and manage objects.
Development objects are arranged together in object lists. Each object list contains all of the objects in a certain category, such as development class, program, or global class.
From the object list, you can select an object by double-clicking it. The system automatically opens the tool that you use to process that kind of object. See also Working with Development Objects [Page 23].

To help you in your work, the Object Navigator provides a comprehensive set of navigation functions. For further information, see Navigation [Page 37].

Special Features

- Use of **controls**

  In redesigning the interface of the ABAP Workbench, SAP developers have used the SAP Control Framework. The various controls that are used are all programmed using ABAP Objects. For an overview of the controls used, refer to Controls and the ABAP Workbench [Ext.].

- Separation of **navigation** and **tool** areas

  The Object Navigator has separate areas for navigation and tools. For further information, refer to Navigation Areas [Page 38].
  The navigation area display the object list. The tool area, on the other hand, displays the different development objects, each in its relevant tool.
  The following tools are integrated in the tool area in the Object Navigator:
  ABAP Dictionary, Class Builder, ABAP Editor, Function Builder, Screen Painter, Menu Painter, and text element maintenance.
  The remaining tools use the entire screen, and thus hide the navigation area when they are open.

- **Overview** of components of an application

  When you are editing a particular object in a tool, you can still see the other components belonging to the program, development class, global class, or function group.

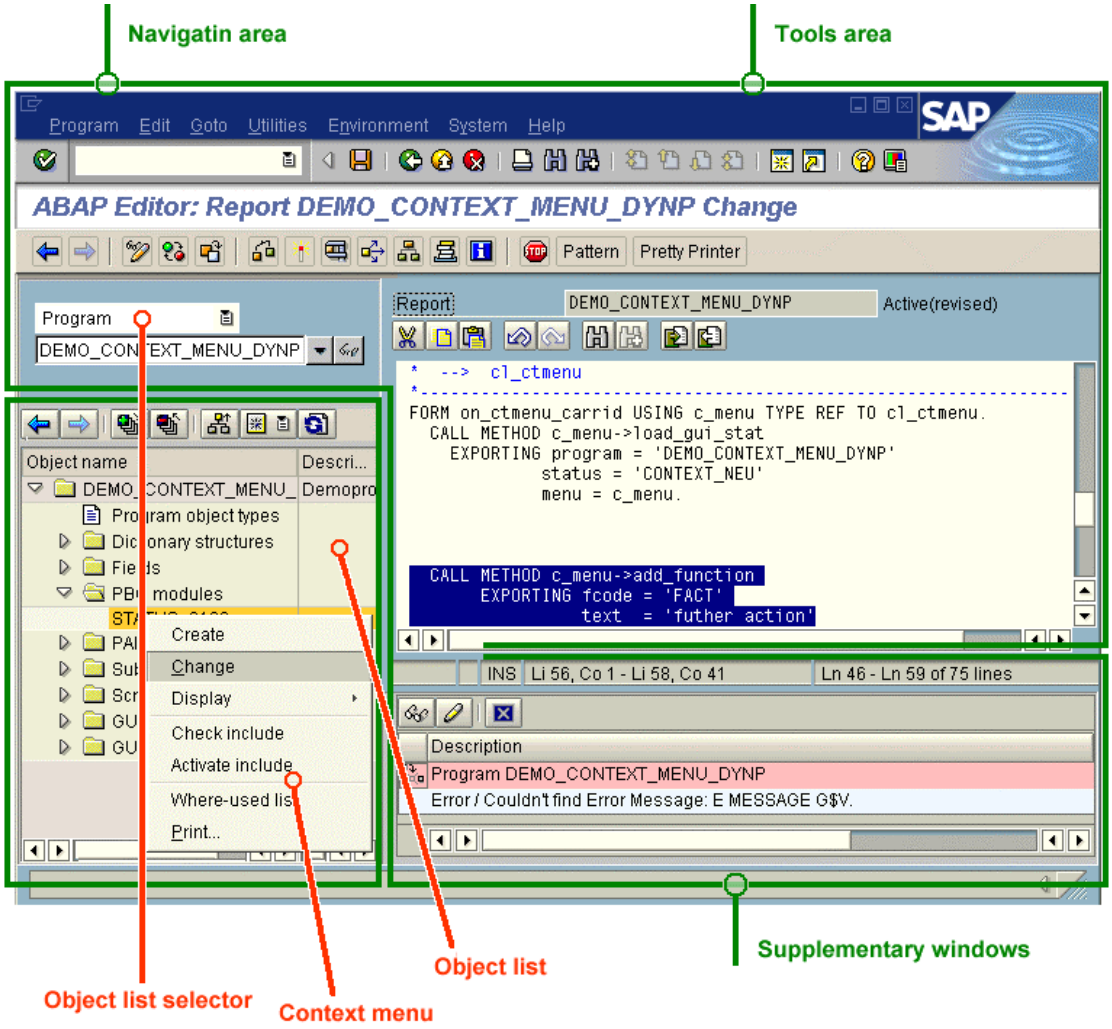
- Context-sensitive **right-click** in browser

  In the navigation area, all of the functions that apply to objects are available from a context menu (right-click). You can choose a function directly from this menu, for example, to start the appropriate tool for editing the object.

- **User-specific views**

  If you often need to work on the same objects in the ABAP Workbench, you can include them in your favorites list.
  If you need to return to the same objects frequently during a single work session, you can include them in a worklist.

Components of the Interface



Navigation Area

- Object list
- Toolbar for object list display
- Context menu

Object Navigator

Tool Area

- ABAP Workbench tools
- Tool functions
  - Menus
  - Standard toolbar
  - Application toolbar
  - Context menu (only in the ABAP Editor in *edit control mode*)

Additional window

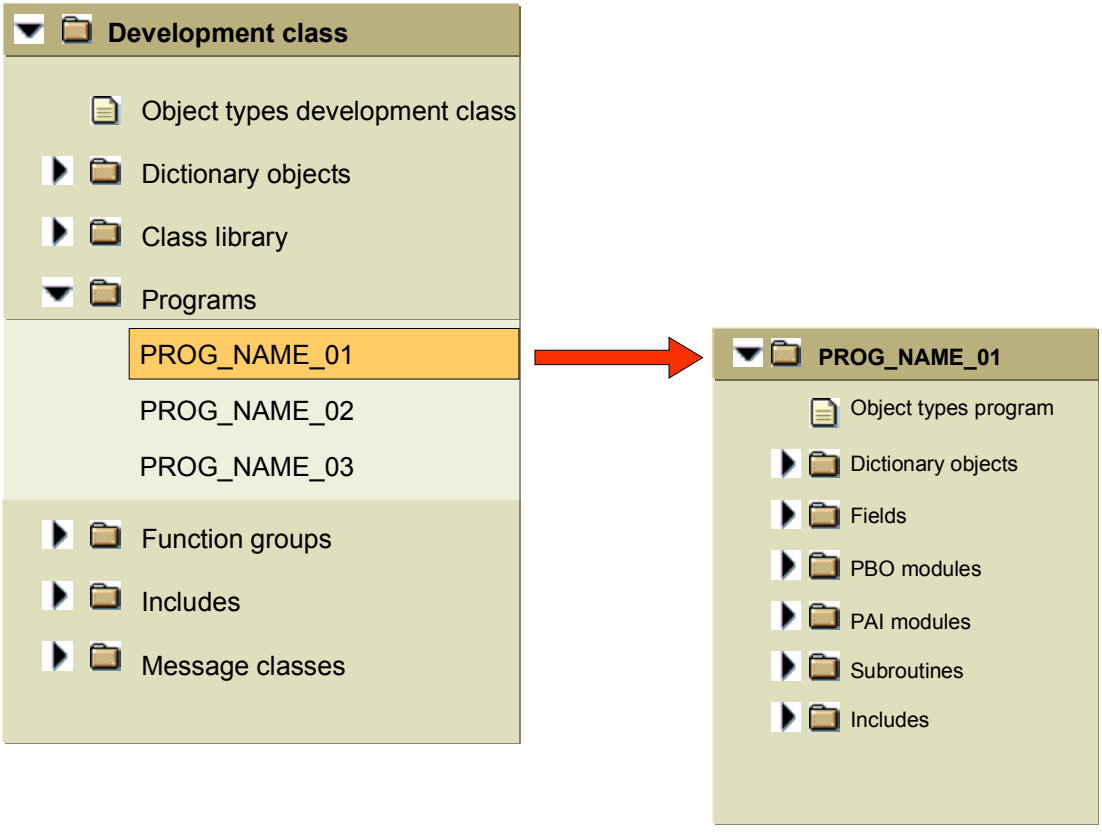
Working With Development Objects

Any component of an application program that is stored as a separate unit in the R/3 Repository is called a development object or a Repository Object. In the SAP System, all development objects that logically belong together are assigned to the same development class.

Object Lists

In the Object Navigator, development objects are displayed in object lists, which contain all of the elements in a development class, a program, global class, or function group.
Object lists show not only a hierarchical overview of the development objects in a category, but also tell you how the objects are related to each other. The Object Navigator displays object lists as a tree.
The topmost node of an object list is the development class. From here, you can navigate right down to the lowest hierarchical level of objects. If you select an object from the tree structure that itself describes an object list, the system displays just the new object list.

For example:



Selecting an Object List in the Object Navigator

To select development objects, you use a selection list in the Object Navigator. This contains the following categories:

Category	Meaning
Application hierarchy	A list of all of the development classes in the SAP System. This list is arranged hierarchically by application components, component codes, and the development classes belonging to them
Development class	A list of all of the objects in the development class
Program	A list of all of the components in an ABAP program
Function group	A list of all of the function modules and their components that are defined within a function group
Class	A list of all of the components of a global class. It also lists the *superclasses* of the class, and all of the inherited and redefined methods of the current class.
Internet service	A list of all of the componentse of an Internet service: Service description, themes, language resources, HTML templates and MIME objects. When you choose an Internet service from the tree display, the Web Application Builder is started. See also Integrating Internet Services [Page 51].
Local objects	A list of all of the local private objects of a user. Objects in this list belong to development class **$TMP** and are not transported. You can display both your own local private objects and those of other users. Local objects are used mostly for testing. If you want to transport a local object, you must assign it to another development class. For further information, refer to Changing Development Classes [Page 34]

Selecting Objects

When you start the ABAP Workbench with the Object Navigator, the system displays the last object list that you displayed. This also applies to new terminal sessions.

Procedure

To select any existing development object for processing in the Object Navigator:

1. On the initial screen, choose the type of object list from the object list selector.
2. Enter the name of the object in the input field.

   This field is normally filled with the name of the last object that you processed.

   If you want to see a list of the objects you have recently used, choose  and select the required entry.

   Step 2 does not apply if you choose *Application hierarchy* in step 1.

3. Choose  or press **ENTER**.

   The requested object is displayed in tree form.

4. Select the required object by double-clicking.

   If the object you selected is itself an object list (for example, a program within a development class), the tree display changes so that only the selected object is displayed.
   In this case, you need to select the object again by double-clicking it.

In step 4, you can also open the required object by choosing *Change* from the context menu. The system then opens the object in change mode.

Alternatively, you can open the object in a separate session in step 4 by choosing *Display* → *In new window*.

Result

The Object Navigator automatically starts the tool with which the object was created and opens the object in display mode. To switch to change mode, choose  in the application toolbar.
If you choose an include, the Object Navigator starts the ABAP Editor and opens the include. If, on the other hand, you choose a GUI status, the Object Navigator starts the Menu Painter in which to open the status.

In each tool, you can use the Other object function to select another object from the same or another category. For example, if you have opened an include in the ABAP Editor, choose *Program* → *Other object* to choose another include or a further object or component.

See also:

Selecting Objects

Working with Development Objects [Page 23]

Creating New Objects [Page 27]

Creating New Objects

The procedure for creating new development objects in the Object Navigator varies slightly depending on whether there an object with the same category already exists in the object list or not. If one exists, a node for that type of object is visible in the tree display. In this case, the corresponding context menu is available.

Procedure

To create a new development object where the object node already exists in the object list:

1. Select the corresponding object node.
2. Choose *Create* from the context menu (right-click).

   A dialog box appears.
3. Enter the name of the new object.

   Remember that all Repository objects in the customer namespace must begin with Y or Z.
4. Choose *Continue*.

   The system checks whether there is already an object with the same name in the system.
5. Enter the required object details and choose *Save*.

   The *Create Object Directory Entry* dialog box appears.
6. Assign a development class to the object.
7. Choose  to confirm your entries.

To create a new development object where the object node does not yet exist in the object list:

1. Choose Other object ().
2. In the *Object Selection* dialog box, choose the appropriate object type.
3. Enter the name of the object you want to create.

   Note that all Repository objects in the customer namespace must begin with Y or Z.
4. Choose  .

   The system checks whether there is already an object with the same name in the system.
5. Enter the required object details and choose *Save*.

   The *Create Object Directory Entry* dialog box appears.
6. Assign a development class to the object.
7. Choose  to confirm your entries.

Result

The system creates the object in the R/3 Repository in an inactive version. When you assign it to a development class, it is automatically linked to the transport and correction system.

You can also create development objects from the initial screen of the corresponding tool.

See also:

Example: Creating a Program [Page 103]

Creating a Program

Prerequisites

This description assumes that you are creating your new ABAP program in the Object Navigator. It is also based on the most general case, and you can use it even if there is not yet a *Programs* node in your object list.
You can also use the *Create* function from the context menu of the object node. See also Creating New Objects [Page 27].

Procedure

1. Choose (*Other object*).

In the *Object Selection* dialog box, chose *Program*.

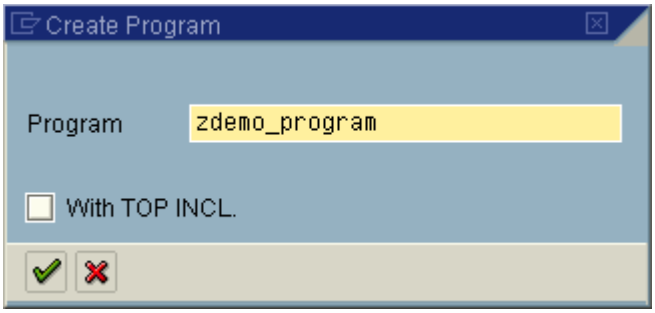
2. Enter the name of the new program.

   Remember that all programs in the customer namespace must begin with Y or Z.

3. Choose .

   The *Create Program* dialog box appears.

4. If you want your program to be an *executable program*, deselect the *With TOP include* option. If, on the other hand, you want to create a *module pool*, select the option.



5. Choose to confirm your entries.
6. If you created a program with a TOP include, a dialog box appears in which you have to enter the name of the include.
7. Choose to confirm your entries.

   Another dialog box appears in which you must set other program attributes.

   For details of what each attribute means, refer to Maintaining Program Attributes [Ext.].

8. Enter the program attributes and choose *Save.*

   The *Create Object Catalog Entry* dialog box appears.

Assign a development class to the program.

9. Choose to confirm your entries.

The program is added to the object list of the relevant development class and displayed under the *Programs* object node. The system starts the ABAP Editor and opens the program in change mode.

Result

The program has been created in the R/3 Repository, and its inactive version is displayed in the ABAP Editor. You have assigned a development class, and it is thus attached to the transport and correction system.

You can also create programs from the initial screen of the ABAP Editor (SE38).

Copying Objects

You may be able to simplify the process of creating new objects by copying existing programs or their components.

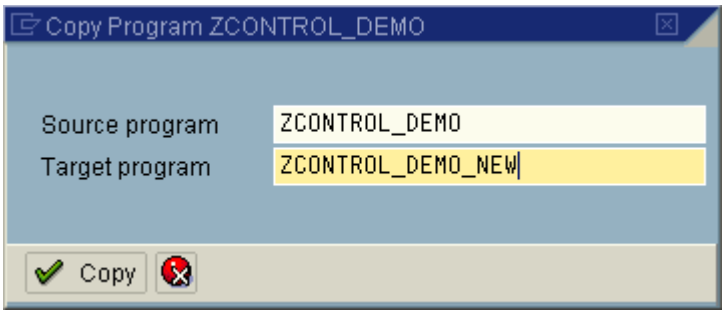
When you copy a program, the system uses the **active** version of the source object. Only in the case of function groups and function modules does the system ask whether you want to use the active or the inactive version.

Prerequisites

If the source object contains components, decide which of the components you want to copy.

Procedure

1. Select the desired object.
2. Choose *Copy* from the context menu.
3. Specify a name for the target object.



4. Choose *Copy*.

   A dialog box appears in which you can enter further components.
5. Select the components that you want to copy.

   If you want to copy a program's includes, another dialog box appears in which you can select the includes individually and assign them new names.
6. Choose *Copy* again.
7. Assign a development class to your object.

   You do not have to do this with local objects.
8. Choose to confirm.

Result

The new object is created in the R/3 Repository and its inactive version is included in the object list.

Deleting Objects

Prerequisites

You can only delete development objects if they are not used by other objects. To find out if they are still in use, use the where-used list function.

Procedure

To delete an object from the object list:

9. Select the object.
10. Choose Delete from the context menu.
11. Confirm the action by choosing *Delete.*
12. Enter a transport request, and confirm by choosing .

If you want to delete several objects from a single object list, select them by holding down the CTRL or SHIFT key and clicking the objects, then follow steps 2 – 4 as described above. You will need to repeat step 3 for each individual object. See also Using Set Mode [Page 36].

Result

Both the active and inactive versions of the object or objects are deleted.

Assigning an Object to a Different Development Class

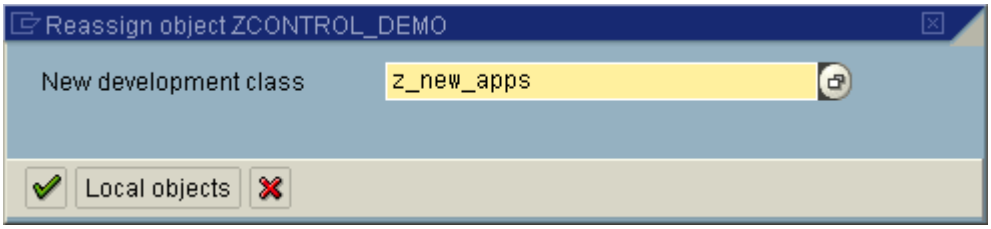
Use

You can reassign objects from one development class to another. This allows you, for example, to create and test a program in your own private objects (development class $TMP) and then assign it to another development class to transport it.

Procedure

1. Select the required object in the object list.
2. Choose *Other functions* → *Change development class* from the context menu (right-click).

   The *Reassign Object* dialog box appears.
3. Enter the name of the new development class.



4. Choose ✔ to confirm.

   If the object was not already assigned to a change request, you must enter one now.
5. Enter a valid change request and choose ✔.

Result

The system assigns the object to the change request and moves it to the new development class.

You can reassign more than one object from the same object list by selecting them and using set mode [Page 36]. The system then reassigns all of the selected objects in a single operation.

Activating Objects

Use

You can activate either your entire worklist, selected objects, or just components of one object (classes in ABAP Objects).

Prerequisites

Before activating an object, the system checks the syntax of the entire object (main program, function group, or class). Any syntax errors are displayed in a list. However, it is still possible to activate objects even if they contain errors. This can be useful if you want to create templates for coding generators.

Procedure

1. Select the relevant object in the object list.
2. Choose *Activate* from the context menu or the icon.
   Your worklist appears. The selected object is highlighted.
3. Choose  to confirm your selection.
   If you are activating an include that cannot be assigned to a single main program, the system asks you for a main program. Choose one main program from the list of programs that use the include.
   A message in the status bar informs you that the object has been successfully activated.

Result

When you activate an object, its syntax is checked. The check uses the inactive versions of the components selected for activation, but the active versions of all other components.
The inactive versions are used to create active versions of the objects. A new runtime version is then generated. Finally, the inactive version is deleted and removed from the list of inactive objects.

Special Features

When you activate an entire object from the object list, only the inactive objects belonging to that object are displayed in the worklist. However, you can display all of your inactive objects by choosing *All inactive objects*.

Using Set Mode

Use

In the Object Navigator, there are certain functions that you can apply to a set of development objects in a single operation:

- Delete
- Display
- Change
- Print
- Write to transport request
- Change development class

Procedure

To apply a function to a set of objects in the Object Navigator:

1. Select a group of objects from an object list (by holding down the CTRL or SHIFT key and clicking the required entries in the tree).
2. Display the context menu (right-click) and choose the required operation.

   If you choose *Display* or *Change*, the system places the selected entries in your **worklist**. The worklist remains available to you until the next time you log off.
3. Carry on processing the objects as normal for the operation you chose.

See also:

Deleting Objects [Page 33]

Assigning a New Development Class [Page 34]

Worklist [Page 45]

Navigation

Separation of Navigation and Tool Areas

You can navigate separately in the navigation and tool areas. When you navigate in one area, the other is not automatically updated.

For example, suppose you choose an include from an object list. This is displayed in the ABAP Editor. If you then double-click the name of a global class referenced in the include, the system starts the Class Builder and displays the relevant class there. The navigation area, however, still shows the object list from the program to which the include belongs.
Similarly, if you choose a different object list, there is no automatic adjustment in the tool area.

See also:

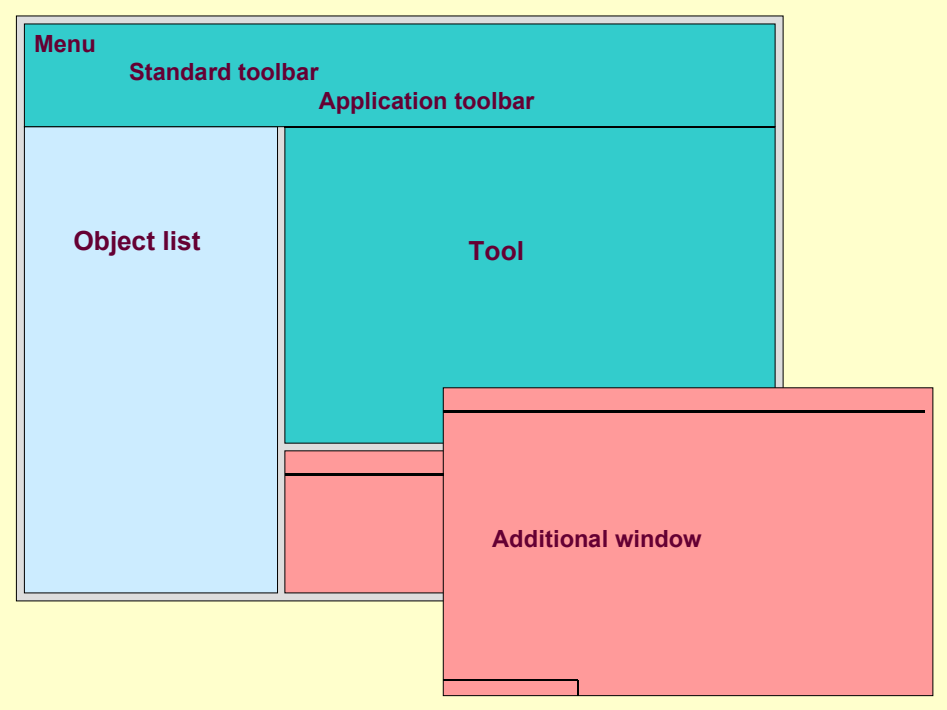
Navigation Areas in the Object Navigator [Page 38]

Navigating in the Tool Area [Page 41]

Navigation Areas

There are three independent navigation areas in the Object Navigator:

- Navigation in the object list
- Navigation in the tools
- Navigation in an additional window
  - Integrated window in the Object Navigator (Syntax check, navigation stack, worklist...)
  - Separate window (for example, when you open an object in a new session).



Functions of the Tools

The menu, standard toolbar, and application toolbar (apart from *Exit*) apply to the work area. In particular, the *Back* function in the standard toolbar only applies to the tool work area.

Synchronizing the Object List

When you navigate to a new object (possibly to different tool as well), you can synchronize the object list.
All of the tools integrated in the tool area of the Object Navigator contain a function that updates the object list.
To do this, choose *Display object list* ( ).

Suppose you are editing a program in the ABAP Editor and double-click the name of a global class. The system displays the class definition in the Class Builder. If you then update the object list, the object list for the global class (with all of its components) appears.

Hiding the Object List

For further information, refer to Hiding the Navigation Area [Page 40].

See also:

Functions in the Navigation Area [Page 42]

Hiding the Navigation Area

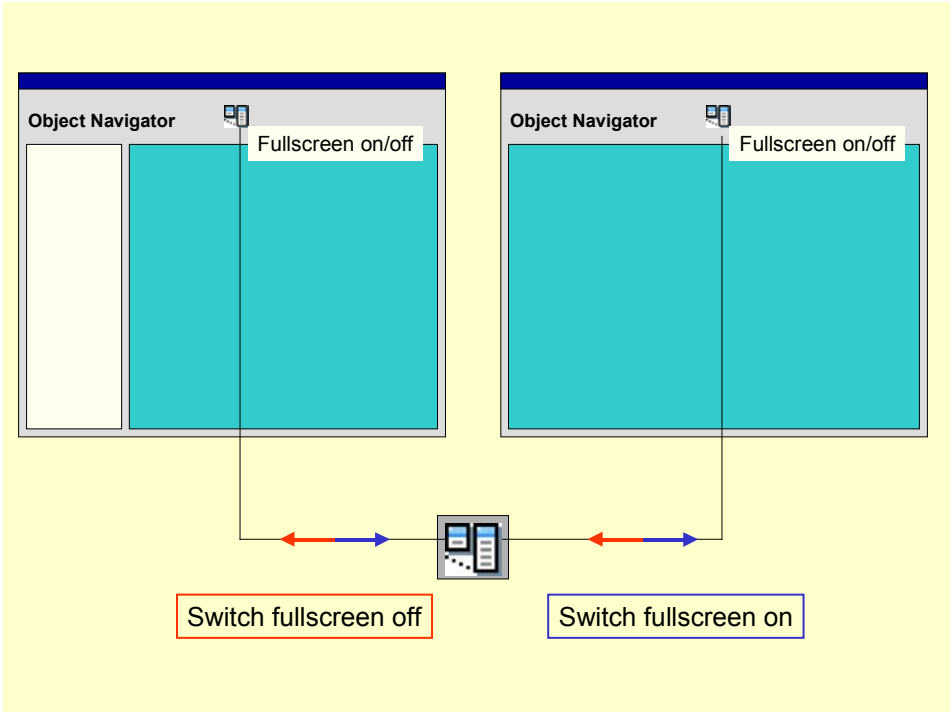
Use

The Fullscreen on/off function in the Object Navigator allows you to hide the navigation area to make the tool area larger. This can be particulary useful if you have a small screen.

Procedure

Choose  from the application toolbar to switch to the fullscreen display.

(If you click the same icon again, the navigation area is displayed again).



See also:

Navigation Areas [Page 38]

Navigating in the Tool Area

There is a series of navigation functions that you can use for the tool area:

Single step navigation (*previous object* and *next object*)

You can navigate separately in the navigation and tool areas of the Object Navigator. Navigation in one area does not automatically update the other.

The  *previous object* and  *Next object* functions allows you to navigate forwards and backwards in the tools. You can access the functions from the application toolbar and the navigation stack. They are useful for switching quickly between two navigation targets in a stack.

Displaying Lists of Navigation Targets

Navigation Stack [Page 43]

Worklist [Page 45]

Finding Out the Navigation Context

Navigation Context [Page 44]

Forward Navigation in the Tools

Navigation by Double-Click [Page 108]

Object List Functions

The object list tree display contains special navigation functions that are arranged in a separate toolbar. They allow you to switch quickly between navigation targets and between hierarchy levels in the object list. There is also a favorites list.

Function	Icon	Description
Previous object list		Navigates back one step in the object list display with history.
Next object list		Navigates forward one step in the object list display with history
Expand all nodes		Expands the object list display fully
Collapse all nodes		Collapses the object list display fully
Higher-level object list in hierarchy		Displays the next-highest object list in the hierarchy. For example, navigating upwards from the object list of a program displays the development class to which the program belongs.
Favorites		Allows you to maintain a favorites list. You can include the current object list in your favorites directly. You can also edit your existing favorites.
Refresh object list		Displays the current object list (including changes). This allows you to update the list. If you are editing in the same session as the object list display, the system refreshes the list automatically.
Close		Closes the navigation area. The tool area then occupies the whole screen.

Including an Object List in the Favorites

1. Display the relevant object list in the navigation area.
2. Choose  *Favorites*
3. Choose *Add.*
   The system confirms that the object list has been added to your favorites (as long as it was not already in the list).

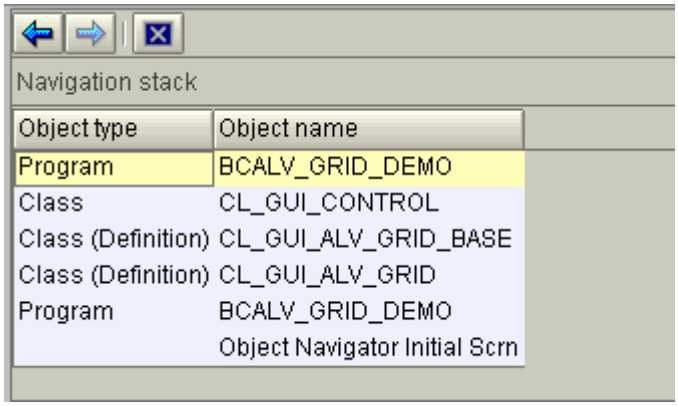
Navigation Stack

All of the navigation steps that you make in an Object Navigator session are automatically recorded by the system on a navigation stack. You can use this stack to return to particular navigation targets within the same session.

You can display the stack in a separate window. This allows you a constant overview of where you are and what you have done during your work.

Choosing an Object from the Navigation Stack

Choose  from the application toolbar in the Object Navigator to display the navigation stack. The system displays a list with the navigation steps in a separate window area.



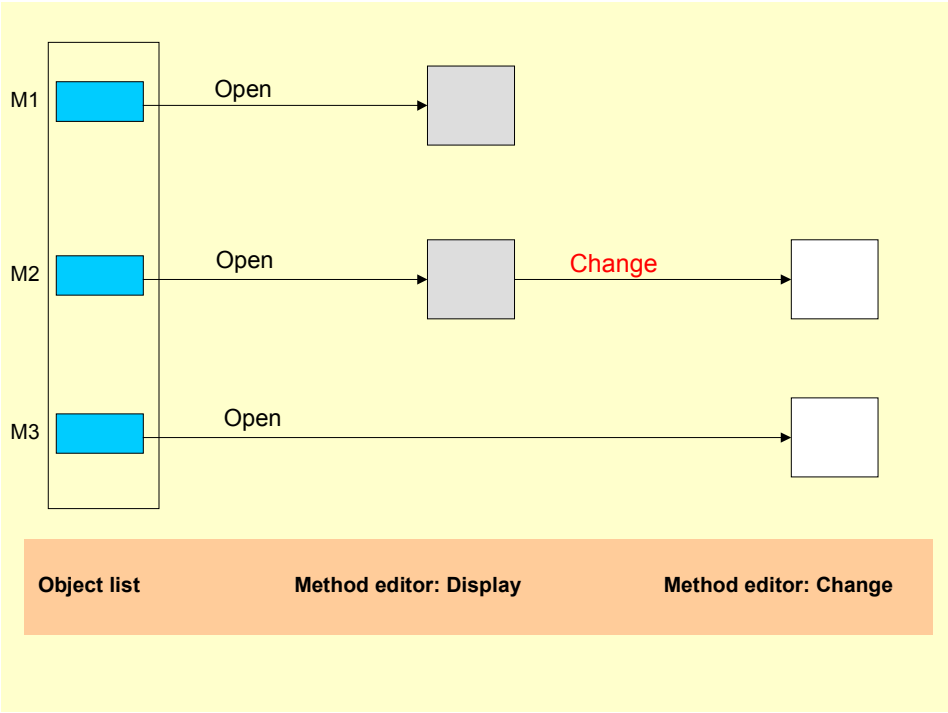
As well as the single-step operations *Previous object* and *Next object*, you can choose any entry by double-clicking it to return to the corresponding object or component. This allows you to skip several steps in the stack to reach the target you want.

The navigation stack is not stored permanently, and is deleted when you leave the Object Navigator. Furthermore, there is a separate navigation stack for each internal session.

Navigation Context

When you navigate between development objects, the navigation context is retained, as long as you are switching between components of the same global class, program, or function group. The effects of this are best illustrated in an example.

Suppose we have a method (already implemented) M1 of a global class that we select from the object list by double-clicking. The system starts the ABAP Editor and displays the method in display mode. If we then call another method M2 of the same class, it is also opened in display mode. Suppose we then switch to change mode and open a third method M3 – its definition will also be displayed in change mode.



Worklist

You can put together a list of navigation targets that you want to use by placing them in a worklist.
A worklist is a user-specific list of objects. You must fill it yourself.
From Release 4.6C, you can save your worklist and return to it in a later terminal session.

Suppose, for example, you use the Repository Information System to search for an object. You might then place it in your worklist so that you can return to it later to copy a section into your own program.

You can also use the worklist to set markers in the Editor. For further details, refer to Setting Markers [Page 47].

Adding Objects to Your Worklist

You can fill your worklist as follows:

- By adding the object on which you are currently working

  Open the required object in the appropriate tool and choose *Utilities* → *Worklist* → *Add current object*.

- By choosing *Display* or *Change* in set mode

  Select a set of objects from the object list, display the context menu (right-click) and choose *Display* or *Change*. See also Using Set Mode [Page 36].

- By dragging and dropping the object from the object list

  Display the worklist (*Utilities* → *Worklist* → *Display*).

  Select one or more objects or components from the object list and drag them from the list to the worklist.

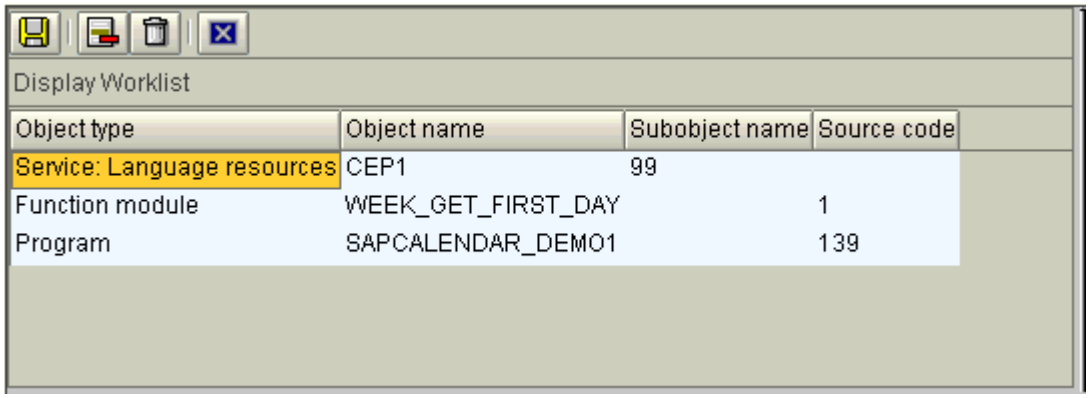
Saving the Worklist

1. If the worklist is not currently displayed, choose *Utilities* → *Worklist* → *Display*.

   The worklist appears in a separate dialog box.

2. Check that you want to keep all of the entries in your persistent worklist.
3. Choose the *Save* icon to save the worklist.

Worklist



You can now return to the worklist in a subsequent terminal session.

Entries in your worklist that are global classes or methods of global classes, function modules, or subroutines can be used to insert source code segments into your programs using drag and drop. For further information, refer to Inserting Patterns Using Drag and Drop [Page 125].

Setting Markers

Use

There are no implicit markers in the ABAP Editor. However, you can set explicit markers to allow you to navigate to a particular line within a program or to a different program altogether. To do this, you just need to include the relevant line in your **worklist**.

Procedure

Creating a Marker:

1. Position the cursor on the required line in the source code.
2. Choose *Utilities* → *Worklist* → *Include current object*.

Navigating to a Marker:

4. Choose *Utilities* → *Worklist* → *Display*.

   The worklist appears in a separate window.

5. Double-click the required entry.

   The source code appears in the Editor, with the cursor positioned on the line at which the marker is defined.

You can also save markers. For further information, refer to the "Saving Entries in the Worklist" section under Worklists [Page 45].

User-Specific Settings

You can preset many of the user-specific settings for the different tools in the ABAP Workbench. These are arranged in a tab that you can call from any of the tools.

Displaying the Settings

You can display your personal ABAP Workbench settings from the Object Navigator or any other tool in the ABAP Workbench. Choose *Utilities* → *Settings*.

ABAP Workbench Tools Settings

Tool	Setting	Description
Workbench (general)	Display background picture	If you set this option, a background picture is displayed in the Object Navigator. This setting has no effect in a WAN.
	No picture	If you set this option, the picture is never displayed
ABAP Editor	Frontend editor	Enables the textedit control mode in the ABAP Editor. When you work in this mode, the source code of your program is loaded at the frontend and can be edited locally. See also Frontend Editor [Page 94]
	Backend editor	The conventional line-based backend editor. See also: Backend Editor [Page 97]
	Numbering	The *With* and *Without line numbering* options only apply to table control mode.
	Compression	In table control mode, you can use *the With compression logic* option to compress logical blocks of ABAP coding. This provides a readable display of complex ABAP programs. For further information, refer to Using Compression Logic [Page 110].
	Path for local editing	Determines the path for local editing. This is the path to which the program is downloaded or stored temporarily, or under which it is started in a local editor. This option only applies to table control mode.
	Upper-/lowercase conversion in display mode	Setting this flag only applies to display mode. The conversion then applies each time you display source code. The display form is based on the settings you make in the Pretty Printer under the *Convert upper/lowercase* option. The *Keyword uppercase* option is not supported for performance reasons. If this flag is not set, there is no special formatting in display mode. In this case, the display is the same in both change and display modes. The source code is displayed exactly as it is stored in the database.

Pretty Printer	Indent	Indents lines in the ABAP source code. For example, all statements belonging to an event are indented by two characters. See also: Improving the Layout [Page 129].
	Convert upper-/lowercase	Option allowing you to standardize the source code display. ▪ Lowercase (for the entire program apart from literals and comments). ▪ Uppercase (for the entire program apart from literals and comments). ▪ Keyword uppercase (highlights ABAP keywords). Note that the *Keyword uppercase* option can be very runtime-intensive in long programs.
Splitscreen	Window arrangement	You can arrange the programs either next to each other or one above the other in the splitscreen editor.
	Comparison operations	If you choose *Ignore indentations*, the system recognizes program lines as identical as long as they have the same contents, even if they are indented differently. If you choose Ignore comments, the system recognizes program lines as identical as long as they have the same contents apart from comments appended using <">. The *Ignore upper-/lowercase* setting is initial. For further information, refer to Special Splitscreen Functions [Page 199].
Class Builder	Display filter	Options for displaying the components of global classes or interfaces in the Class Builder. They allow you to extend or restrict the standard display.
	Scope Filter	Restricts the display of class components to instance or static components.
Screen Painter	Graphical layout editor	If you set this option, the system uses the graphical layout editor in the Screen Painter. If the option is not set, the alphanumeric editor is used instead.

Menu Painter	Output length in status maintenance	*Function code* option: The length of the function code field in the Menu Painter can be set to between 4 and 20 characters. If the function code you want to enter is longer than the length specified in this option, the field cannot accept input beyond its defined length. *Text length* option: The length of the text field for a function text in the Menu Painter can be set to between 10 and 40 characters. If the text you want to enter is longer than the length specified in this option, the field cannot accept input beyond its defined length. Note that these user-defined settings are only valid in the status maintenance screen of the Menu Painter.
	Frontend platform	*Frontend* option: The function key settings depend on the platform you are using. All function codes maintained in the Menu Painter are platform-independent, but the function key names are not. By selecting a platform, you ensure that the function keys are labeled using the convention of the selected platform.
Function Builder	Check syntax in test	When you test a function module, the system can check the syntax of the function group to which the function module belongs. The default setting for this option is inactive to improve performance in the Function Builder test environment.

Integrating Internet Services

As part of the mySAP.com initiative, SAP has started to integrate Internet services in the ABAP Workbench in Release 4.6C. This means that it is now possible to create Web objects as Repository objects and publish them on an Internet Transaction Server (ITS).

Contents

Further Documentation

The HTMLbusiness Language [Ext.]

Developing Internet Applications With the SAP Web Studio [Ext.]

Web Application Builder

Purpose

The Web Application Builder allows you to create Web development objects within the ABAP Workbench. Existing R/3 transactions requre these objects to allow them to run as Web transactions in a Web Browser. You can also use the Web Application Builder as an integrated environment for creating MiniApps.

Integration

The Web Application Builder is a fully integrated tool within the ABAP Workbench. Objects that you create with it, such as service files, HTML templates, and MIME objects, are stored in the R/3 Repository and are connected to the R/3 Change and Transport System.

Features

- Creating Internet services for existing R/3 transactions or MiniApps.
- Implementing the dialog logic.
- Generating the HTML templates for the screens of a transaction. These contain standard HTML and HTML^{Business} statements that map the screen layout.
- Editing the generated HTML templates using HTML and HTML^{Business} to develop them further.
- Including MIME objects (icons, graphics, Java applets, animation...) to improve the layout further.
- Creating language-specific texts (language resources).
- Publishing the services or individual service ocmponents on the Internet Transaction Server (ITS)
- Executing the complete Web transaction from the ABAP Workbench.
- Connection to the Change and Transport System (CTS).
- Connection to Version Management.

Constraints

Certain functions are not yet available:

- There is no syntax check
- HTML^{Business} and the flow logic are not yet integrated with the Debugger.

Creating an Internet Service

Use

In order for you to log onto the R/3 System and start a Web application, the Internet Transaction Server requires a relevant Internet service.
Each service consists of a set of components:

- Service description: Parameters that describe how the service should be executed, particularly logon data and details of the transaction that is to be executed.
- HTML templates: You can create an HTML template for each screen of an Easy Web Transaction (EWT).
- Language resources: A language resource contains all the texts that are required to execute a service in a particular languge. They ensure that the service is language-independent.
- MIME objects: These can be icons, graphics, Java applets, or sound/video components that you use to extend the user interface in the Web environment.

A concrete instance of a service is defined by a theme. A theme has its own set of HTML templates, language resources, and MIME objects, and gives a service a particular appearance. The actual function of the service remains unchanged.

Prerequisites

If you want to create a service for an R/3 transaction, you should first check its classification and change it if necessary. The default classification of a transaction is *Professional User Transaction*.

Procedure

8. Start the Object Navigator (SE80).
9. In the object list selection, choose *Internet Service*.
10. Enter the name of the service you want to create.

    Note that all Repository objects in the customer namespace begin with Y or Z.

11. Choose  or **ENTER**.

    The system checks whether there is already a service with the same name in the system. If there is not, the *Create Object* dialog box appears.

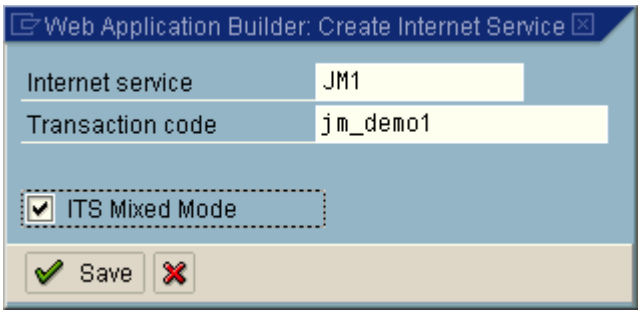
12. Confirm the creation by choosing *Yes*.

    The *Create Service* dialog box appears.

13. If your Web application is a Web transaction, enter the transaction code of the corresponding R/3 transaction.

    If you only want to generate HTML templates for some of the screens in the R/3 tranaction, select the ITS mixed mode option.
    For further information, refer to Using Mixed Mode [Page 57].

Creating an Internet Service



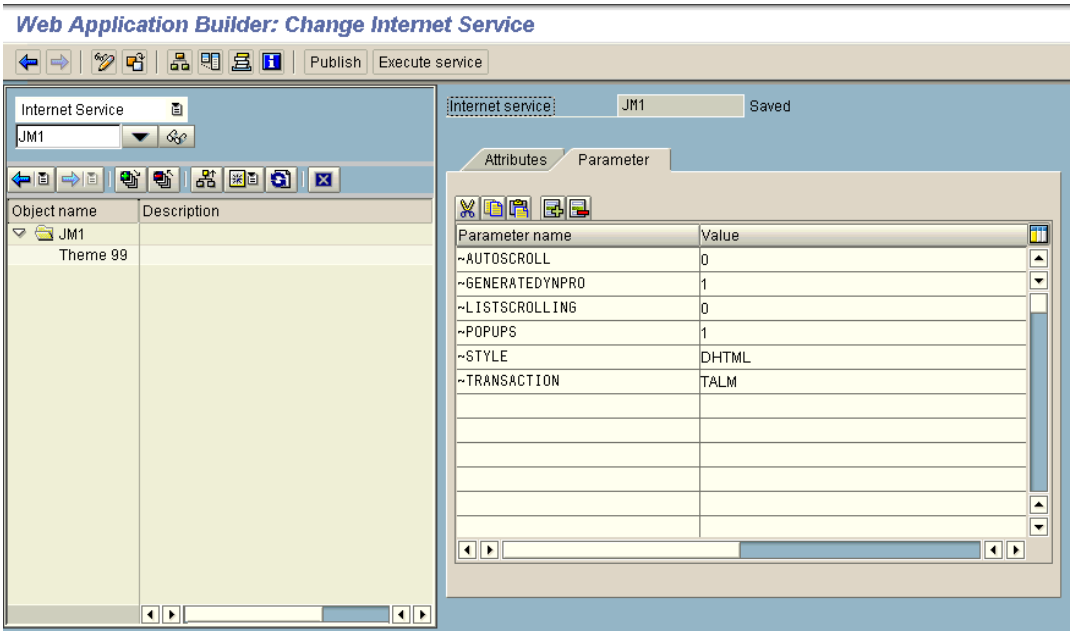
1. Choose .

   The Create Object Catalog Entry dialog box appears.

2. Assign the service to a development class.

Result

The Internet service has now been created as a development object in the R/3 Repository, and appears in the object list. The service has been assigned the theme *99*, which is the current theme.
If you entered an R/3 Transaction in step 6, the parameter *~TRANSACTION* will have been generated for the service. if you set the *ITS mixed mode* flag, the corresponding parameters will have been filled with appropriate values.



See also:

Using Mixed Mode [Page 57

Generating HTML Tempaltes [Page 58

Publishing a Web Application [Page 68]

Using Mixed Mode

Use

Mixed mode allows you to use **both** templates **and** the automatic WebGUI generation within one Web transaction. Screens that do not have templates are generated automatically by the WebGUI.
If a template exists for a screen, the ITS will use it and use HTML^{Business} functions to generate an HTML document for the screen before sending it to the Web browser.

Mixed mode allows you to create templates for selected screens within a transaction. This is particularly useful if particular screens or transactions cannot be reproduced in the WebGUI without errors, or where the layout is inappropriate to your requirements. You can improve the layout of the screens by hand.

Prerequisites

To enable screens to be generated by the WebGUI mechanism, you need to add certain parameters to the Internet Service. The automatic generation is not supported by default for compatibility reasons.

If you did not set the ITS mixed mode flag when you created the service, add the following parameters and values manually. In other cases, they are generated automatically when you create the service and filled with valid values.

Parameter	Value	Description
~generateDynpro	1	Switches on the automatic generation mode for screens that have no template
~listscrolling	0	Simulation of downward scrolling in list reports
~popups	1	Displays dialog boxes instead of suppressing them
~style	DHTML	Specifies which generator should be used
~autoscroll	0	Simulates downward scrolling for step loops and table controls

Creating HTML Templates

Use

When you implement a MiniApp, you must create HTML templates. The dialog logic of a MiniApp runs on the ITS, not in R/3.

For each transaction, you can choose whether you want to generated HTML templates for all screens, for some screens (mixed mode [Page 57]), or at all. Templates that you create explicitly are identical to the HTML documents that are generated automatically by the WebGUI.

Generating templates explicitly is useful if the WebGUI features are insufficient for your needs and you would need to adapt the standard generated template anyway. This will particularly be the case if you are trying to improve the layout of a screen or if you want to include hyperlinks.

Standard template generation from the WebGUI should be sufficient for most transactions. The WebGUI can display the screen elements of a simple transaction (text fields, input/output fields, checkboxes, radio buttons, tabstrip controls, table contorls, subscreens...) without you having to go to the effort of creating a template.

Prerequisites

- You must already have created the service.
- You have sufficient knowledge of HTML and HTML^{Business} to take advantage of the template-based approach.

Procedure

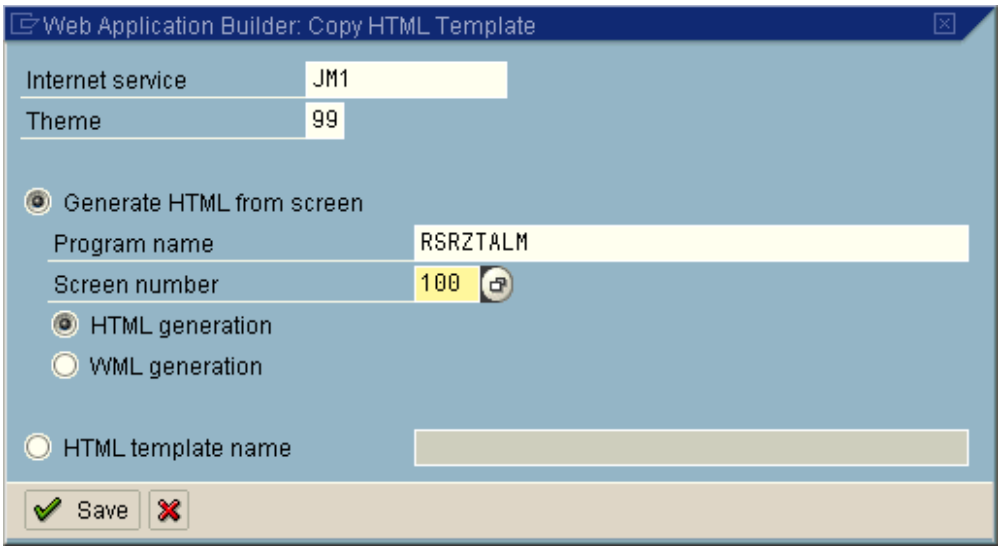
To create an HTML template from the tree display in the object list:

8. Right-click the name of the service.
9. From the context menu, choose *Create* → *Template*.

   The *Create Template* dialog box appears.
10. Enter the theme for the service and fill out the remaining fields.

    If the Web application is a Web transaction and you want to generate a template for a particular screen, select *Generate HTML from screen* and enter the program name and screen number.

    If the application has no corresponding R/3 screen (MiniApps), select *Name of template* and enter the name.



1. Confirm by choosing *Save.*

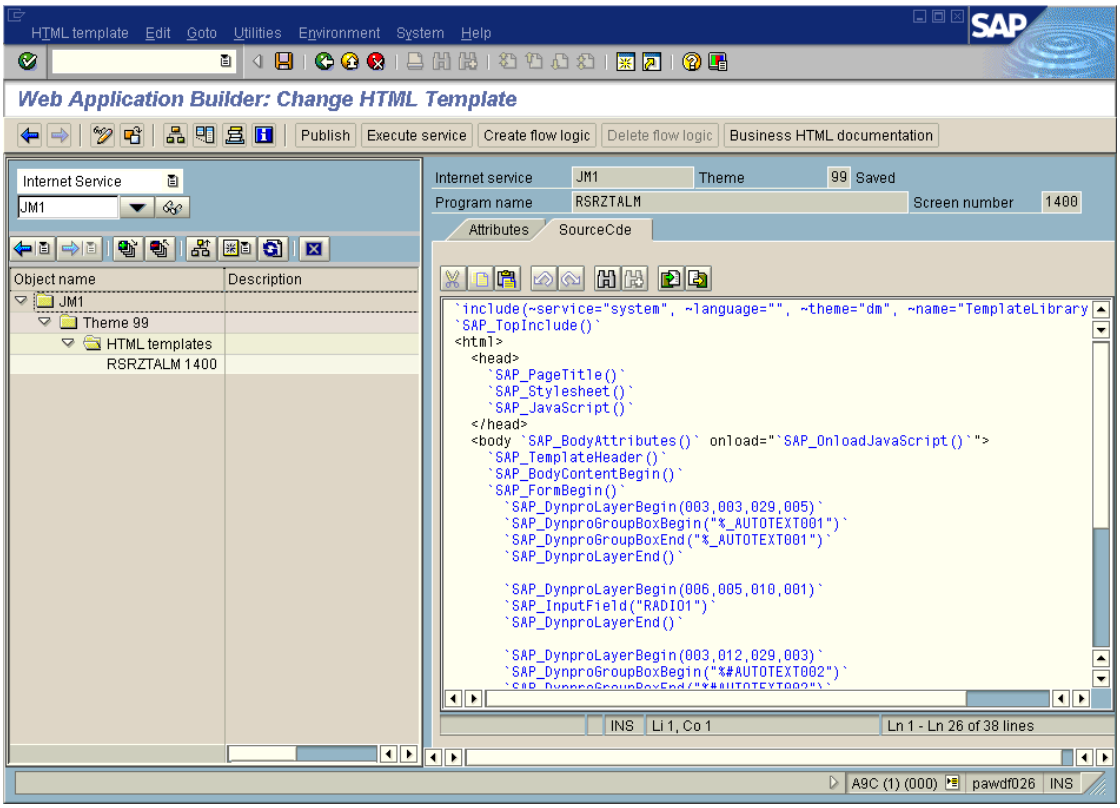
   The *Create Object Catalog Entry* dialog box appears.

2. Assign the template to a development class and choose .

Result

The generated template appears in the object list under *Templates*. The generated contents of the tempalte are displayed in the Editor. Only the static screen information is evaluated - an HTML^{Business} function is inserted in the template for each screen element. These are highlighted in blue. You can now change the contents of the template using standard HTML and HTML^{Business}.

Creating HTML Templates



See also:

Extending HTML Templates [Page 62]

Adding MIME Objects [Page 64]

Publishing Services [Page 68]

Extending HTML Templates

Once you have created an HTML template, you can change the generated source code.

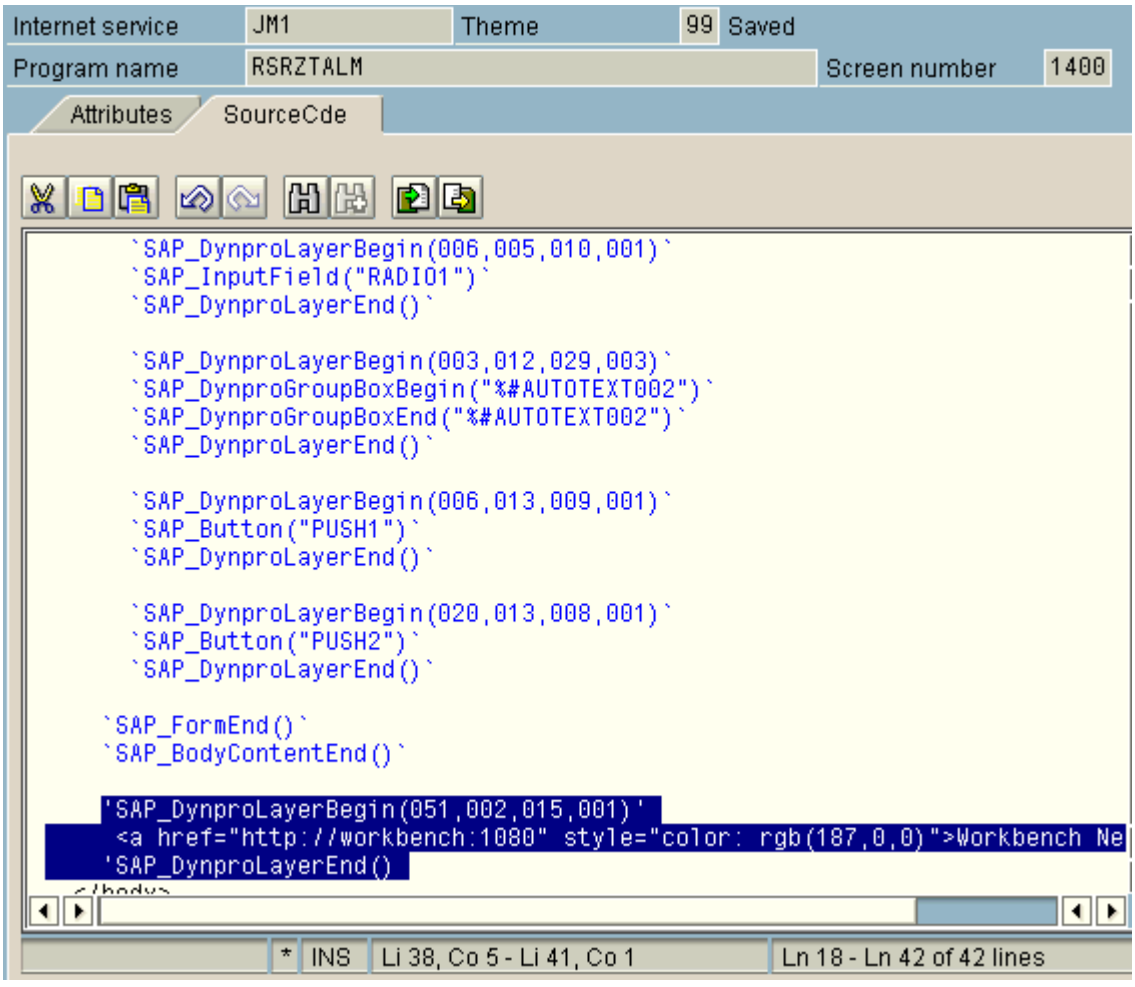
To do this, you must be familiar with the basics of HTML and HTML[Business].

HTML[Business] is an extension of standard HTML developed by SAP to allow R/3 screen data to be merged dynamically with information on HTML templates and to make it easier for the ITS to exchange data between the R/3 System and the Web Server.

For further information, refer to HTMLBusiness Reference [Ext.].

Example

This example sets a hyperlink to a particular position on an HTML page:



See also:

Publishing a Service [Page 68]
Adding MIME Objects [Page 64]

Adding MIME Objects

Use

You can use MIME objects (icons, graphics, audio files, animations...) to improve the layout of your Web applications.

Prerequisites

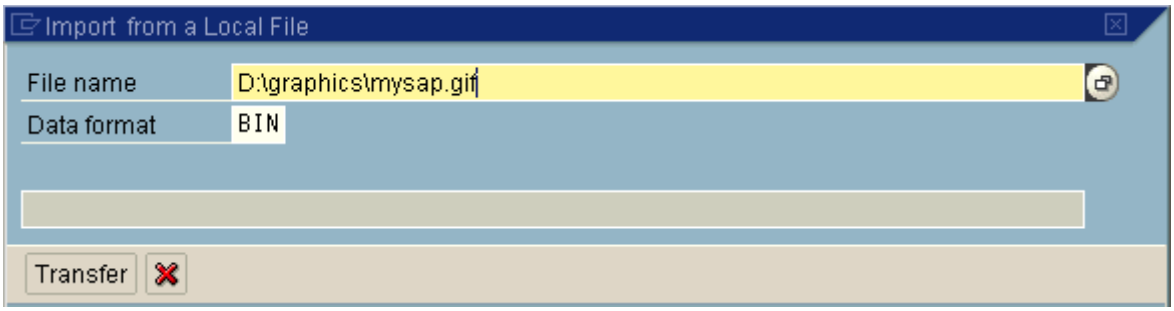
You must already have created an Internet service.

Procedure

To add a MIME object to an Internet service from the object list:

13. Right-click the relevant service.
14. In the context menu, choose *Create* → *Mime*.

    The *Read from Local File* dialog box appears:



15. Enter the path name of the file you want to import, and ensure that the file format is correct.
16. Choose *Import*.

    The *Create Mime* dialog box appears.
17. Enter the theme and the name for the MIME object.
18. In the Name field, you can create a subdirectory, separated from the name of the MIME object by a forward slash ("/").

19. Choose to continue.

    The *Create Object Catalog Entry* dialog box appears.

20. Assign the MIME object to a development class and choose .

Result

The MIME object has been inserted in the R/3 Repository as a standalone object. It appears under Mimes in the object list display, and, if it is a graphic, its contents are displayed.

You can now use this object in your interface design.

When you publish the service, the MIME objects are not stored in an ITS directory. Instead, they are stored on the HTTP server under the name and subdirectory you specified in step 5 above.

See also:

Publishing Web Applications [Page 68]

Creating Language Resources

Use

A language resource contains all of the language-specific elements of an Internet service and enables you to make your application multilingual. Compared with hard-coded text in the HTML template, it also makes it easier for you to understand and maintain your source code.

For each language-specific text in your Web interface, you insert a placeholder into the HTML template (similarly to text elements in an ABAP program). The actual texts are maintained through the theme parameters with the same name. At runtime, the ITS recognizes the placeholders for each template and replaces them with texts in the appropriate language.

Prerequisites

You must already have created the HTML templates for your Internet service.

Procedure

Adding Placeholders to an HTML Template

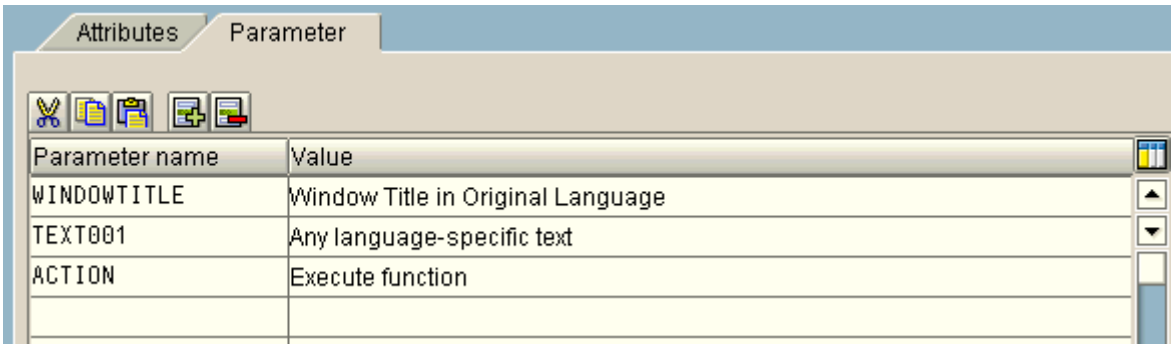
1. Open the relevant template.
2. Enter the placeholder for the language-specific texts in the HTML source code.

   Use the following syntax: `` `#Placeholder` ``

   Suppose we defined three placeholders (`` `#windowtitle` ``, `` `#text001` `` und `` `#action` ``) in the source code.

Entering Language-Specific Texts

1. Double-click the theme.
2. In the Parameter name column, enter the name of the placeholder (without #) and enter the text in the original language as its *Value*.



Result

The theme parameters are now part of the service. They are translation-relevant parts of the R/3 Repository object, and as such will enter the translation workflow when you release the service.

When you start the service in the original language, the texts appear in the relevant language.

If there is no translation of the language-specific texts in the logon language, no text is displayed when the user executes the service.

Publishing a Service

Use

In order for an Internet service to be executed by the ITS, it must be stored in the ITS file system. This is known as publishing the service. You can choose to publish the entire service or just parts of it. When you publish the whole service, the corresponding Internet service and its HTML templates are placed in the file system of the AGate server, and the MIME objects are placed in the file system of the WGate server.

Note that by default, the Internet service is published on all Internet Transaction Servers. However, you can choose to restrict the publication to an ITS assigned to your particular R/3 System. To do this, choose *Utilities* → *Settings* and then, under *ITS*, enter the required server. For further information, refer to User Settings [Page 72].

Prerequisites

At least one ITS must have been assigned to the R/3 System, and it must be active.

Procedure

To publish an entire Internet service from the object list:

21. Right-click the relevant service.
22. Choose *Publish* → *Entire service* from the context menu.

    If an error occurs while the system is publishing the service, a log is generated containing the relevant message texts.

    If no errors occur, the system displays the message *The object has been published successfully*.

Result

Once you have published the entire service, you can start your Web application.

See also:

User Settings for Internet Services [Page 72]

Executing a Service [Page 69]

Executing a Service

Use

Use this function to test a Web transaction or MiniApp from the ABAP Workbench.

Prerequisites

You must already have published the entire service on the ITS. The ITS must be active.

Procedures

To start the Web application from the Object Navigator:

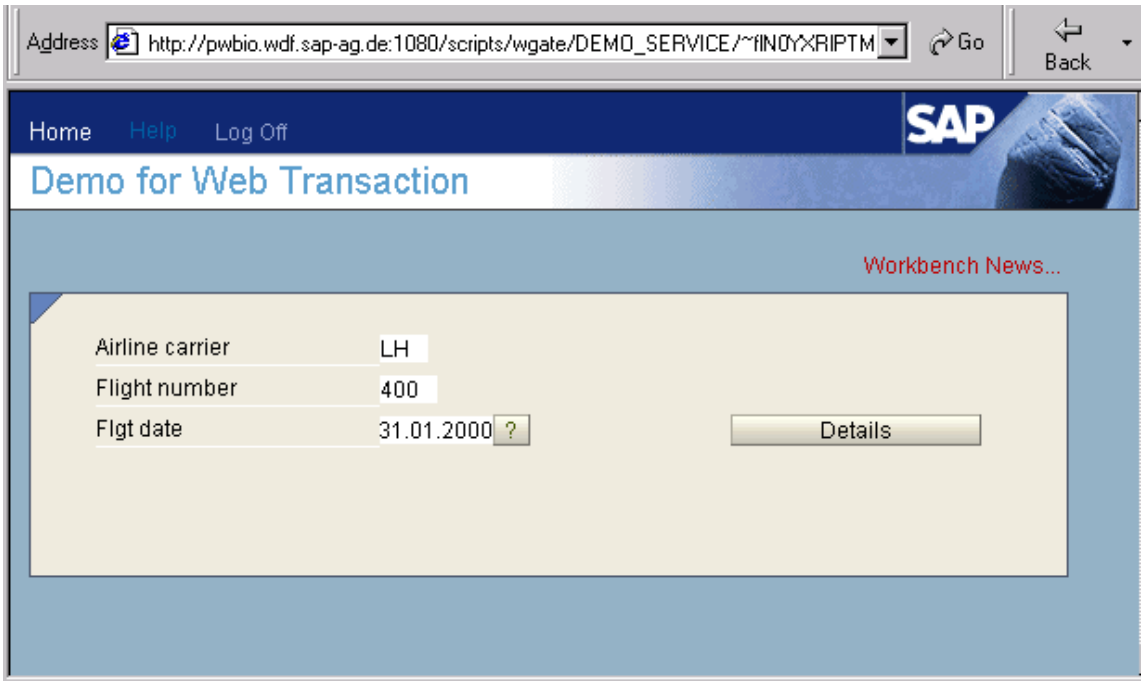
1. Select the relevant service.
2. Choose *Execute*.

   The system starts the Web browser and displays a logon window.
3. Check the logon language, and log onto the ITS by choosing *Logon*.
4. Run the Web application.

You can suppress the logon window by specifiying a logon language in the ~LANGUAGE parameter.

Result

The service is started using the HTTP address **http://<ITS>:<Port>/scripts/wgate/<service>/!**



The way in which a Web transaction is displayed depends on the classification of the underlying R/3 transaction. *Professional User Transactions* are displayed with the normal R/3 menus, command field, standard toolbar, and application toolbar in the Web browser. *Easy Web Transactions* (EWTs), on the other hand, are displayed with a special EWT header.

User Settings for Internet Services

Use

You can use the ITS settings to

- Find out which ITS instances are assigned to your SAP System.
- Restrict the ITS instances on which services are published to a single instance.
- Find out the name of the default Web server for starting services.
- Specify a Web server other than the default for starting services.

Activities

To change the ITS settings:

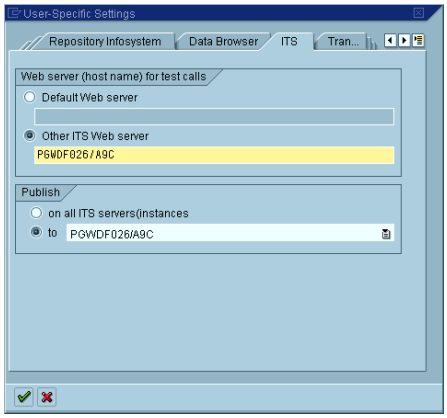
1. Choose *Utilities* → *Settings*.
2. Under *ITS*, change the required filter settings.
3. Choose ✔ to confirm.

The new settings will apply to all subsequent service publications.

Filter settings

Setting	Description	
Web server (Hostname) *for tests*	*Default web server*	The name of the default web server used to start services in the current SAP System. This name is entered in the Customizing table **TWPURLSVR** (transaction **SM30**). If there is **no entry** for this setting, the name of the web server has not yet been entered. In this case, you should inform your system adminstrator (unless you have authorization to maintain the table yourself).
	Other ITS web server	You can enter the name of any other web server on which you want to run services.  **Note**, however, the assignment of the web server to an SAP System. If you specify a web server belonging to the ITS instance of a different SAP System, the service will be started on a different SAP System to the one from which it was published.

Publish	*On all ITS instances*	The service is published on all ITS instances assigned to the SAP System. The message that follows successful publication is only displayed if the service could be published successfully on all ITS instances. Errors occurring on any instance are logged.
	on <individual instances>	If you choose an ITS instance from the list box, services will only be published on that instance. The name of the ITS instance is derived from the RFC destination details as maintained in transaction **SM30**. If the list box contains no entries, no destinations have been created. To maintain destinations, start transaction **SM30**, enter the view **IACORDES** and create the RFC destination. If you do not have authorization for this, inform your system administrator.



Documentation Not Available in Release 4.6C

Documentation Not Available in Release 4.6C

Documentation Not Available in Release 4.6C

Documentation Not Available in Release 4.6C

Documentation Not Available in Release 4.6C

Documentation Not Available in Release 4.6C

Documentation Not Available in Release 4.6C

Documentation Not Available in Release 4.6C

Documentation Not Available in Release 4.6C

ABAP Workbench: Tools

Overview and Navigation

Object Navigator [Page 20]

ABAP Workbench Overview [Page 86]

Information About Repository Objects [Page 496]

Developing Web Objects

Web Application Builder [Page 52]

ABAP Development

ABAP Dictionary [Ext.]

ABAP Editor [Page 92]

Class Builder [Page 202]

Function Builder [Page 408]

Screen Painter [Page 262]

Menu Painter [Page 353]

Context Builder [Ext.]

Programming Environment

Text Element Maintenance [Page 140]

Variant Maintenance [Page 167]

Splitscreen Editor [Page 195]

Maintaining Message Classes [Page 190]

Test Tools in the ABAP Workbench

ABAP Debugger [Page 444]

Runtime Analysis [Page 447]

Performance Trace [Page 449]

Other Tools

Area Menu Maintenance (From Release 4.6A) [Page 396]

Transaction Maintenance [Ext.]

Modifications

Modification Assistant [Ext.]

See also:

Further Reading: ABAP Programming [Page 91]

Overview of the Workbench

This section introduces you to the ABAP Workbench and the concepts you need to know before you start to use the Workbench.

Topics:

Tool Integration and Working Methods

The tools in the Workbench are integrated. For example, when you are working on a program, the ABAP Editor will also recognize objects created using other tools. This integration also means if you double-click an object to select it, the Workbench automatically launches the tool that was used to create the object.

SAP has developed the Object Navigator to help you to organize your application development in this integrated environment. It provides a context that makes it easier for you to trace the relationships between objects in a program. Rather than working with tools and recalling development objects, you work with objects and allow the Workbench to launch the appropriate tool for an object.

We recommend that you use the Object Navigator to develop your applications. For this reason, this documentation is written from the perspective of an Object Navigator user.

See also:

Development Objects and Development Classes [Page 88]

Object Navigator [Page 20]

Development Objects and Development Classes

When you work with the Workbench, you work with development objects and development classes.

Development objects are the individual parts of an ABAP application. Some examples of development objects are programs like reports, transactions, and function modules. Program components such as events, screens, menus, and function modules are also development objects. Finally, objects that programs can share are development objects as well. These shareable objects include database fields, field definitions, and program messages.

A development class is a container for objects that logically belong together; for example, all of the objects in an application. A development class is also a type of development object. An example of a development class might be `General Ledger Accounting`.

When you create a new object or change an existing object, the system asks you to assign the object to a development class.

Storage of Development Objects

The SAP system stores development objects in the R/3 Repository, which is a part of the database.

When you complete work on a development object like a program, screen, or menu, you generate a runtime version of the object. This runtime version is stored, along with the object, in the Repository. An application consists of several runtime objects that are processed by the work processes in the R/3 System.

In a standard SAP installation, development and live operation take place in separate systems. New applications are created in the development system and transported to the production system. Daily work takes place in the production system which uses run-time versions created in the development system.

The division between production and development systems is recommended because changes to an existing ABAP application take immediate effect. To prevent disturbances in daily work flow in the production system, all developments are carried out in development systems designed especially for this purpose.

The Workbench Organizer

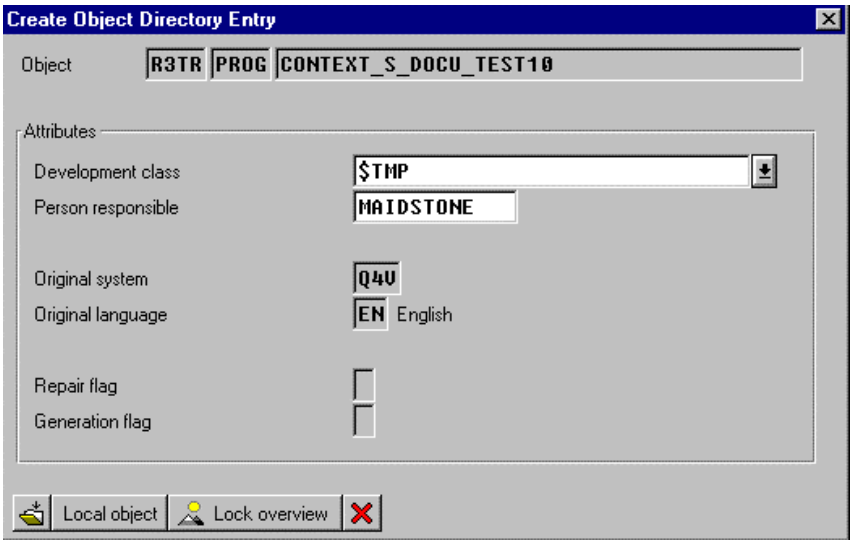
You use the Workbench Organizer and the transportation system to move applications from the development system to the production system. The Workbench Organizer also provides version control and tracking. When you work with the Workbench, you will encounter safeguards provided by the Workbench Organizer. A brief overview of these checks and how they affect the development process is provided in Development in a Team Environment [Page 89].

For further details, see the Workbench Organizer [Ext.] documentation.

Development in a Team Environment

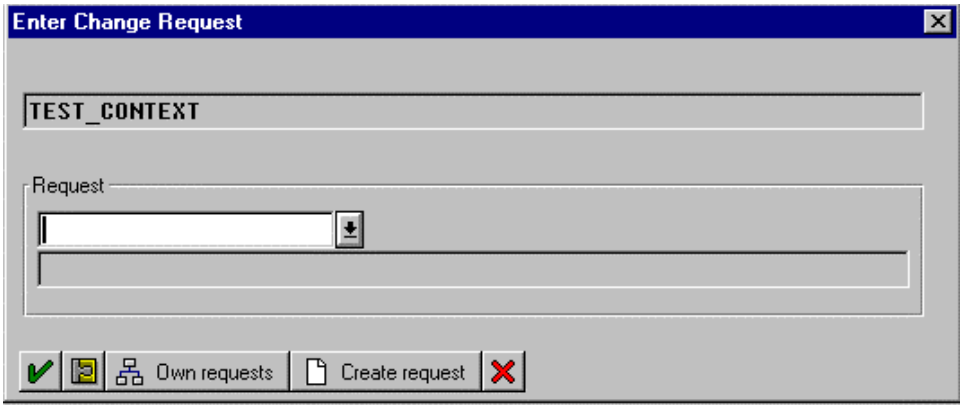
ABAP allows you to divide work on large projects among several programmers. Consider an accounting application project with an accounts payable module and an accounts receivable module. The ABAP environment helps you to create a work area in the system for the project. You can then assign tasks to each programmer and follow their work as it progresses.

The tool you use for tracking development projects is called the Workbench Organizer. The Organizer tracks changes to existing SAP development objects and the creation of new objects. If you create a new object, the Organizer asks you for a development class when you try to *Save* the object:



The Organizer uses the development class to determine whether a change request is required. A change request records the changes made to one or more development objects. The $TMP development class contains local objects. Local objects are not transported and so the Organizer does not monitor them.

If you specify a non-local development class, the system prompts you to enter a change request. The system also queries you for a change request the first time you attempt to change an existing non-local object. The query dialog appears as follows:



Development in a Team Environment

When you associate a change with a request, the system creates a task for you under that change request. The organizer creates a task for each programmer making a change under a change request. You can think of a change request as a container of change tasks.

Once you associate an object with a change request, the system views the objects as under development. The object is locked, and cannot be changed by other users. When you have finished creating or changing an object, you release your task. To transport your changes to a production system, you release the change request that held your task.

You can change the development class and change request associated with an object. For more information about changing the development class of an object, refer to Reassigning Objects to Another Development Class [Page 34] For more information about the Workbench Organizer and the transport system, see the Workbench Organizer [Ext.] documentation.

Further Reading

The following documentation contains more information about the ABAP Workbench and ABAP programming:

- ABAP Workbench Tutorial [Ext.]. An introduction to the most important aspects of the ABAP Workbench and simple ABAP programming.
- ABAP User's Guide [Ext.]. A comprehensive guide to the individual parts of an ABAP program. It contains information about both basic and advanced ABAP programming, including the techniques of list and transaction processing.
- ABAP Dictionary [Ext.]. An introduction to ABAP data, covering basic and aggregated objects, how to maintain ABAP Dictionary objects, and other special areas.
- Workbench Organizer [Ext.]. A guide to organizing large development projects in the R/3 System. It covers how to set up the Workbench Organizer and transport system, version management, and the modification concept.
- Extended Function Library [Ext.]. Information about a range of standardized dialogs for address management, application logs, and archiving.
- Basis Programming Interfaces [Ext.] A description of the programming interfaces for SAP components, including background processing and batch input.

ABAP Editor

You use the ABAP Editor to create and edit your programs and their components.

If you want to use the ABAP Editor together with the Modification Assistant, read the Modifications in programs [Ext.] documentation.

Topics in this Documentation

Introduction to the ABAP Editor

The ABAP Editor is a tool that you use to write ABAP programs, class methods, function modules, screen flow logic, type groups, and logical databases.

Editor Modes

The ABAP Editor has two different modes:

- Frontend editor
- Table control mode

The frontend editor uses the SAP Textedit Control from the SAP Control Framework. It loads your source code onto the frontend and allows you to perform many tasks without any communication with the application server.

Please note that **command mode** is no longer supported in the new version of the ABAP Workbench and is therefore no longer available in Release 4.6B.

Integration

Table control mode and frontend editor are fully compatible and interchangeable – source code that you have created using one mode is properly reproduced by the system in the other without you having to do anything yourself. In particular, the line lengths are the same. If you exceed the maximum length, the system automatically inserts a line break.

Both editor modes offer the same source code layout. The contents of the editor are displayed exactly as they are stored in the database. There is no automatic conversion (for example, into uppercase) in either mode.

Differences

There are differences between the two modes in respect of how they are used, the system requirements, and (to a small extent) the functions that they contain. Other sections of this documentation explain the differences in more detail.

See also:

The Frontend Editor [Page 94]

The Backend Editor [Page 101]

Changing the Editor Mode [Page 100]

Local Editing [Page 101]

The Frontend Editor

Use

In the frontend editor, the ABAP source code is loaded onto the frontend and edited locally. The advantage of this is that all editing functions that do not require communication with the backend can be performed very quickly.

However, the communication channel between the frontend and backend can be overloaded when you use the frontend editor. If you have a large program, this can be a problem even in a LAN environment, but in a WAN, it can become critical.

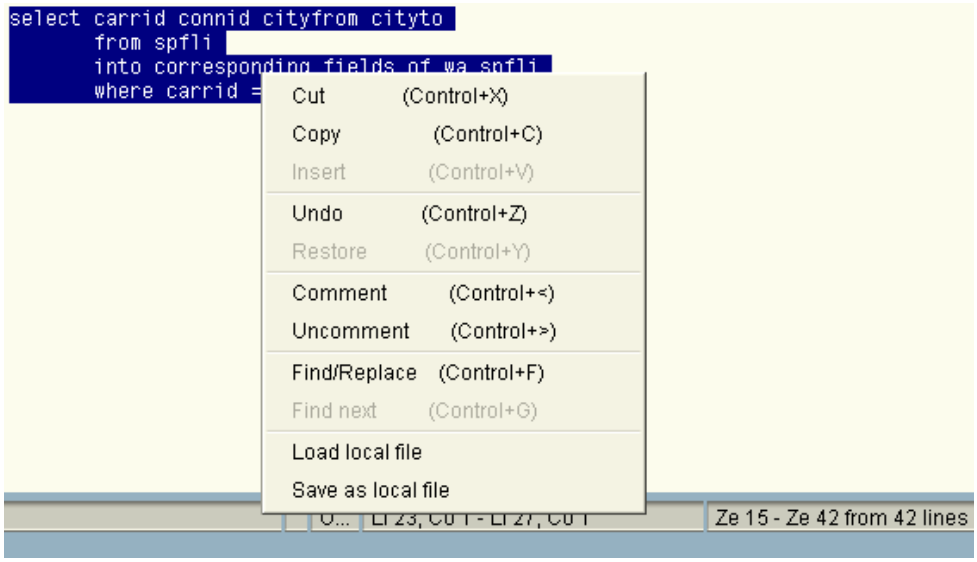
You can edit the following development objects in the frontend editor:

- ABAP programs
- Method implementations (Class Builder)
- Function module implementations (Function Builder)
- Screen flow logic (Screen Painter)
- Type groups (ABAP Dictionary)

Edit control mode has the same range of features in each situation in which you can use it.

Edit control mode is not yet supported in the splitscreen editor, the BOR editor, or the logical database editor.

Editing an ABAP program in edit control menu using the context menu:



Prerequisites

Operating system

You must be using a 32 bit operating system (Windows NT 4.0 or Windows 95). Other operating systems are not currently supported.

Controls and DLLs

Various DLL and OCX files are required at the frontend. They are installed and registered automatically when you install the SAPgui.

Features

The frontend editor of the ABAP Editor contains the following features:

- Local scrolling (only available in this mode)
- Cut, copy, and paste for selected text areas (only available in this mode)
- Drag and drop (only available in this mode)
- Context menu (right-click) for accessing editor functions (only available in this mode)
- Local find and replace function
- Navigation to a selected line (using the context menu)
- Access to the buffer and block operations (using the context menu)
- Commenting out text blocks
- Working with blocks and clipboards
- Navigation functions (forward navigation)
- Syntax check, displaying error messages and warnings in a separate window
- Colored highlighting for comment lines
- Automatic line feed when the maximum line width is reached (only available in this mode)
- *Insert statement* function.
- Multiple-step undo and redo functions (only available in this mode)
- Displays current cursor position
- Pretty Printer for standardizing the layout
- Import and export for local files.

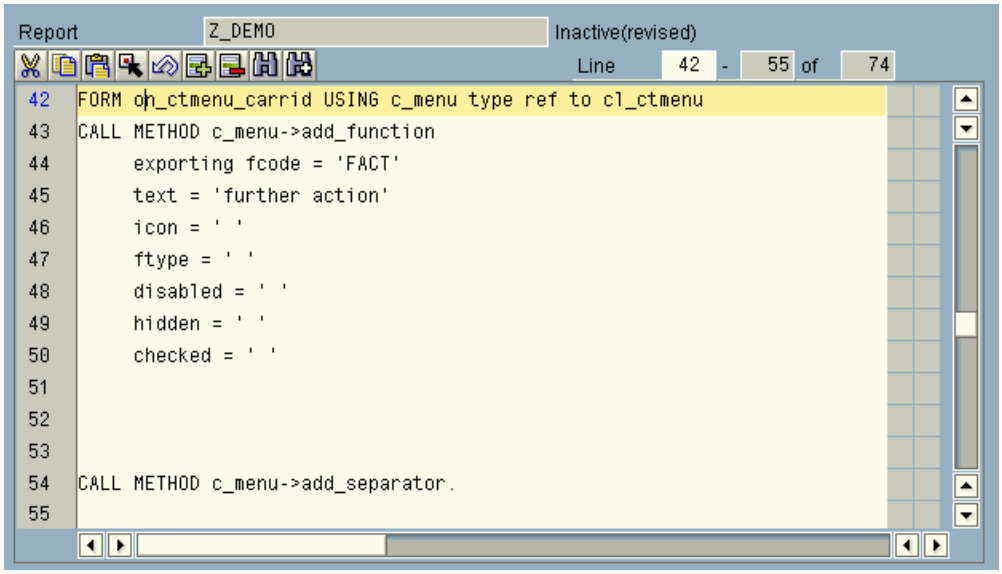
Table Control Mode

Use

The backend editor allows you to use the traditional backend editor for editing your ABAP coding. The editor is line-based, and to use normal editor functions such as cut, copy, and paste, you must first select a block of lines. It is often useful to use the clipboards in this mode.

If you have a very long program (and especially if you are working in a WAN environment), the backend editor can produce better performance than the frontend editor. Furthermore, backend editor allows you to edit any development object that is based on the ABAP Editor. The splitscreen editor, the BOR Editor, and the Logical Database Editor are only available in the backend editor.

Editing an ABAP program in the backend editor:



Prerequisites

Unlike the frontend editor, there are no operating system restrictions for the backend editor.

Features

The backend editor of the ABAP Editor provides the following functions:

- Compression logic (only available in this mode).
- Line numbering (only available in this mode).
- Find and replace functions.
- Colored highlighting for comment lines.
- Insert statement function.
- Include expansion (only available in this mode).
- Single-step undo function.
- Conversion of a text block to comment lines.
- Pretty printer for standardizing program layout.
- Syntax check.
- Upload and download of local files.

Changing the Editor Mode

When you start the ABAP Editor, the system displays the source code in the editor mode saved in your user-specific settings [Page 48].

Procedure

To change the Editor mode from anywhere in the ABAP Workbench:

1. Choose *Utilities* → *Settings* .

   The *User-Specific Settings* dialog box appears.

2. Choose *ABAP Editor*.
3. Set the new editor mode.
4. If you are going to use *Table control mode*, you can choose whether to switch the *Line numbering* and *Compression logic* on or off.
5. Choose  to leave the dialog box.

Result

The changed settings are saved, and will be retained even after you log off from the SAP System.

See also:

Using Compression Logic [Page 110]

Local Editing

Both modes in the ABAP Editor allow you to transfer an ABAP program to a local file on your frontend machine. You can then edit the source code using an editor of your choice, before loading the file into the ABAP Editor again.

Backend Editor

Uploading and Downloading Source Code

To upload a file into the ABAP Editor, choose *Utilties* → *More utilities* → *Upload/download* → *Upload*. Enter the path and filename of the file you want to read in the dialog box, then choose *Copy*.

Note that when you copy the file, its contents overwrite all of the previous contents of the ABAP Editor.

To download a file from the ABAP Editor to a local file, choose *Utilities* → *More utilities* → *Upload/download* → *Download*. In the dialog box, enter a path and name for the file (with an appropriate file extension). Then choose *Copy*.

In the backend editor, the ABAP Editor supports the following file format:

Extension	Upload/Download
ASC	ASCII
BIN	Binary
DAT	ASCII data table with column tab
DBF	DBASE format (available for download only).
IBM	ASCII with IBM code page conversion (DOS)
WK1	Spreadsheet format (available for download only).

Starting the Local Editor

Specify a *Path for local editing* in the ABAP Editor Setting [Page 48] with the same name. The source code is buffered under this path. Then choose *Utilities* → *More utilities* → *Edit locally*. The system downloads the program source code to a local editor. When you leave the editor, the source code is reloaded into the ABAP Editor.

Frontend Editor Mode

Uploading Source Code

To upload a local file into the ABAP Editor, choose Load local file (). Then select the required local ASCII file.

Local Editing

Note that the entire contents of the ABAP Editor are overwritten with the uploaded file.

To download the contents of the ABAP Editor to a local text file, choose *Save to local file* ().
Enter the path name for the local text file.
Note that this function does not apply to selected lines, but always to the entire contents of the editor.

Creating a Program

Prerequisites

This description assumes that you are creating your new ABAP program in the Object Navigator. It is also based on the most general case, and you can use it even if there is not yet a *Programs* node in your object list.
You can also use the *Create* function from the context menu of the object node. See also Creating New Objects [Page 27].

Procedure

10. Choose (*Other object*).

In the *Object Selection* dialog box, chose *Program*.

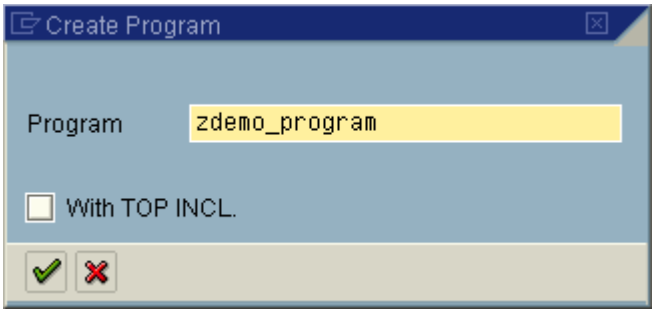
11. Enter the name of the new program.

    Remember that all programs in the customer namespace must begin with Y or Z.

12. Choose .

    The *Create Program* dialog box appears.

13. If you want your program to be an *executable program*, deselect the *With TOP include* option. If, on the other hand, you want to create a *module pool*, select the option.



14. Choose to confirm your entries.
15. If you created a program with a TOP include, a dialog box appears in which you have to enter the name of the include.
16. Choose to confirm your entries.

    Another dialog box appears in which you must set other program attributes.

    For details of what each attribute means, refer to Maintaining Program Attributes [Ext.].

17. Enter the program attributes and choose *Save.*

    The *Create Object Catalog Entry* dialog box appears.

Assign a development class to the program.

18. Choose to confirm your entries.

The program is added to the object list of the relevant development class and displayed under the *Programs* object node. The system starts the ABAP Editor and opens the program in change mode.

Result

The program has been created in the R/3 Repository, and its inactive version is displayed in the ABAP Editor. You have assigned a development class, and it is thus attached to the transport and correction system.

You can also create programs from the initial screen of the ABAP Editor (SE38).

Editing the Source Code

The precise procedure for navigation and editing source code varies according to the editor mode that you are using. The following documentation explains these differences in detail.

Topics

Navigating in the Source Code

To position the cursor within the visible area on the screen, you can simply click the left mouse button at the required point.
The editor also has scrollbars that you can use to scroll through the source code.

There is also a range of key combinations with which you can move the cursor. These key combinations may have different effects in the frontend and backend editor modes:

Key combination	Moves the cursor in the frontend editor	Moves the cursor in the backend editor
↑	Up one line, possibly beyond the visible area	Up one line, always within the visible area
↓	Down one line, possibly beyond the visible area	Down one line, always within the visible area
←	One character to the left	As in the frontend editor
→	One character to the right	As in the frontend editor
PgUp	One page up	As in the frontend editor
PgDn	One page down	As in the frontend editor
Home	To the beginning of the current line	As in the frontend editor
End	To the end of the current line	As in the frontend editor
Ctrl + ↑	To the beginning of the current paragraph	Up one line
Ctrl + ↓	To the end of the current paragraph	Down one line
Ctrl + ←	One word to the left	As in the frontend editor
Ctrl + →	One word to the right	As in the frontend editor
Ctrl + PgUp	To before the first character in the visible area	To the beginning of the source code
Ctrl + PgDn	To after the last character in the visible area	To the end of the source code
Ctrl + Home	To the beginning of the source code	To the beginning of the current line
Ctrl + End	To the end of the source code	To the end of the current line

Navigieren zu einer Zeile

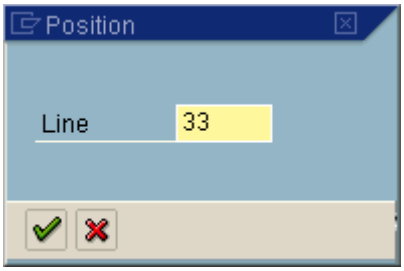
Backend Editor

In the backend editor, you can navigate to a particular line by entering the line number in the input field on the right above the editor pane. The line you entered becomes the current line, and is displayed as the first visible line of code.

Frontend Editor

To navigate to a particular line in the source code:

1. Right-click in the Editor to open the context menu.
2. Choose *Goto line*.
3. In the dialog box, enter the required line number.



4. Choose ✔ to confirm.

The line is scrolled to the top of the display.

See also:

Navigating by Double-Click [Page 10]

Using Compression Logic [Page 110]

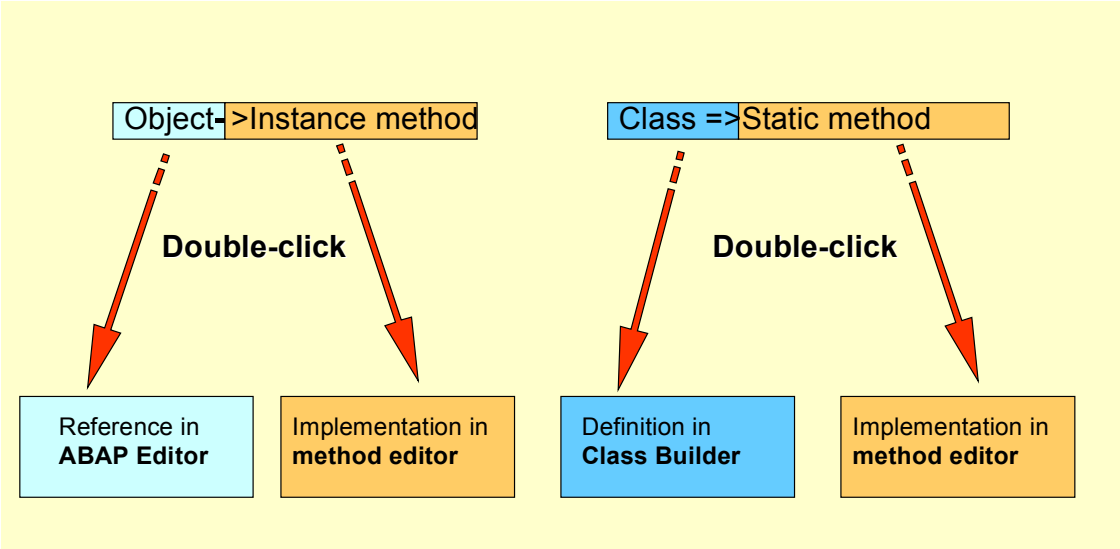
Navigating By Double-Click

In order for a double-click in the ABAP Editor to have the desired effect, you must place the cursor either directly to the left of or within the required string.

Double-clicking can trigger the following navigation steps:

Double-click	To
A development object	Navigate to the object definition. If the object has not yet been defined, the system asks whether you want to create it. If you select an object that is defined using another tool, the system closes the ABAP Editor and starts the corresponding tool.
The definition of an object	Display a where-used list for the object. A dialog box may appear first, allowing you to specify extra search options for the where-used list.
An ABAP keyword	Trigger an error message. Double-clicking on an ABAP keyword is not generally supported by the navigation.
A compound ABAP structure (IF ... ENDIF...)	Navigate to the “other half” of the structure (for example, from METHOD to ENDMETHOD, or from ENDIF to IF).
A WHEN statement	Navigate to the next WHEN statement (or ENDCASE if you double-click the last WHEN).
An ELSEIF statement	Navigate to the next ELSEIF, ELSO, or ENDIF statement.
An include name	Jump to the definition of the include.
A space in the line	Make the line the top line displayed on the screen
A line number	Make the line the top line displayed on the screen (only works in table control mode with line numbering).

Special case: Double-click on a method call in a global class:



Using Compression Logic

If you want an overview of how your source code is structured in a program, you can use compression logic. Certain parts of a program (subroutines, modules, nested ABAP statements) form logical blocks that you can compress. As well as these predefined blocks, you can define your own logical blocks in the source code.

Prerequisites

You can only use compression logic in the backend editor. The With compression logic option must be active in the ABAP Editor section of your settings [Page 48].

Features

- Compression of logical blocks.

  Choose  .

- Expanding logical blocks.

  Choose  .

- Inserting your own blocks.

1. Insert a comment line beginning with `*{` at the start of the block you want to define.
2. Insert a comment line beginning with `*}` at the end of the block.
3. Press **ENTER** to confirm.

The compression is not saved when you leave the ABAP Editor. When you next open the program, all blocks will be expanded.

Editing Source Code (Frontend Editor)

Selecting Text

Using the mouse

To select	Procedure
A block of text	Click and hold the left-hand mouse button, drag the cursor across the block
A line	Click the left-hand mouse button to the left of the line
A set of lines	Click the left-hand mouse button to the left of the first line then drag the cursor
The entire source code	Ctrl + left-hand mouse button to the left of a line.

Using the keyboard

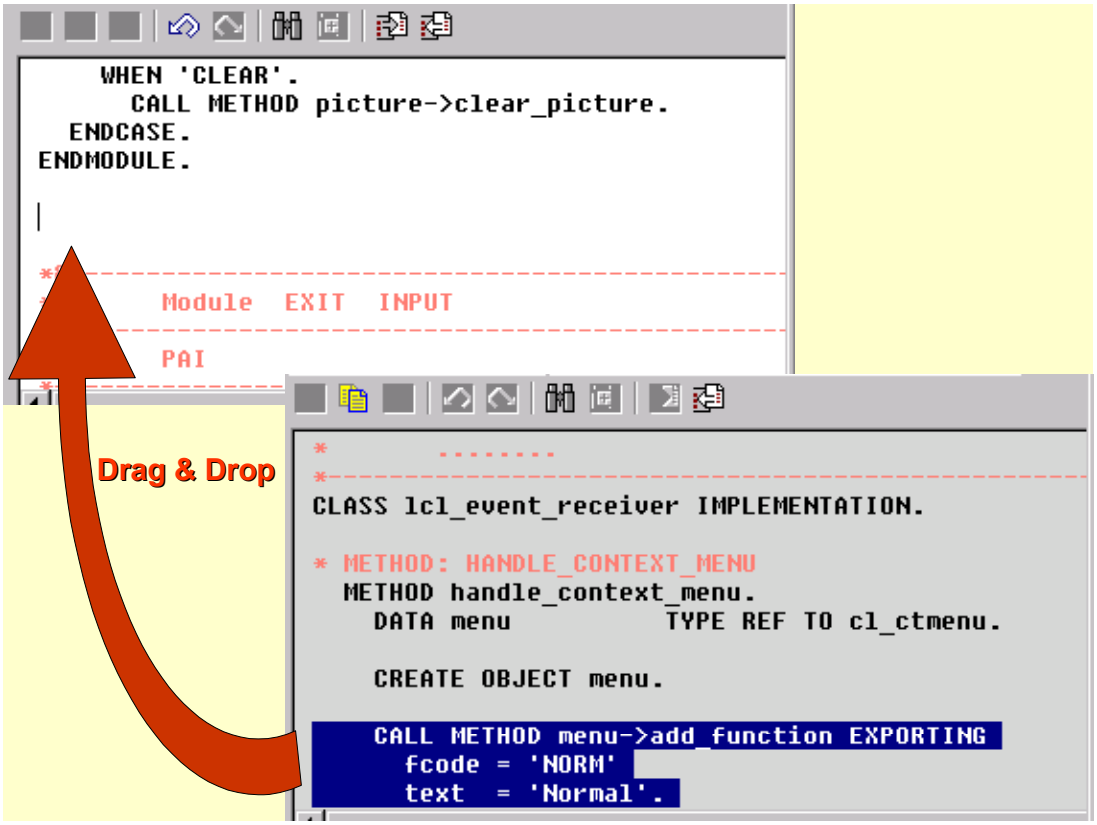
To select	Press
The entire source code	Ctrl + A
The next line up	Shift + ↑
The next line down	Shift + ↓
One character to the left of the cursor	Shift + ←
One character to the right of the cursor	Shift + →
One page upwards from the cursor position	Shift + PgUp
One page downwards from the cursor position	Shift + PgDn
To the beginning of the current line	Shift + Home
To the end of the current line	Shift + End
To the beginning of the current paragraph	Shift + Ctrl+ ↑
To the end of the current paragraph	Shift + Ctrl + ↓
To the beginning of the current word	Shift + Ctrl + ←
To the end of the current word	Shift + Ctrl + →
To the beginning of the visible area	Shift + Ctrl + PgUp
To the end of the visible area	Shift + Ctrl + PgDn
To the beginning of the entire source code	Shift + Ctrl + Home
To the end of the entire source code	Shift + Ctrl + End

Editing Text

Function	Procedure
Cut	Select the relevant text. Choose *Cut* from the context menu (right-hand mouse button), or choose *Ctrl* + *X*. The selected text is placed in the buffer.
Copy	Select the relevant text. Choose *Copy* from the context menu (right-hand mouse button), or choose *Ctrl* + *C*. The selected text is placed in the buffer.
Paste	Position the cursor where you want to insert the text. Choose *Paste* from the context menu (right-hand mouse button), or choose *Ctrl* + *V*. The selected text is placed in the buffer.
Moving text using drag and drop (within the same editor)	Select the relevant text. Press and hold the left-hand mouse button and drag the selected text to where you want to place it. Release the mouse button. The selected text is moved to the new position.
Copying text using drag and drop (within the same editor)	Select the relevant text. Press and hold Ctrl + the left-hand mouse button and drag the selected text to where you want to place it. Release the mouse button. The selected text is copied to the new position.
Commenting out source code	Select the relevant code. From the context menu, choose *Comment*, or press *Ctrl* + *<.*
Removing comments from source code	Select the relevant code. From the context menu, choose *Uncomment* or press *Ctrl* + *>.*

Inserting Code From Other Programs Using Drag and Drop

To insert code from one ABAP program in another, open both programs (each in its own session). Then, select the code that you want to copy and use drag and drop to place it in the other editor. The source code will either be moved or copied (if you press and hold the *Ctrl* key while dragging it, or if you opened the source program in display mode).


WHEN 'CLEAR'.
CALL METHOD picture->clear_picture.
ENDCASE.
ENDMODULE.
Module EXIT INPUT
PAI
Drag & Drop
CLASS lcl_event_receiver IMPLEMENTATION.
* METHOD: HANDLE_CONTEXT_MENU
METHOD handle_context_menu.
DATA menu TYPE REF TO cl_ctmenu.
CREATE OBJECT menu.
CALL METHOD menu->add_function EXPORTING
fcode = 'NORM'
text = 'Normal'.

Editing Source Code (Backend Editor

By default, the editor works on the current line, which is determined by the cursor position and, in the backend editor, is highlighted.
You can, however, combine groups of lines into blocks.
This allows you to apply functions like *cut*, *copy*, and *concatenate* to a block of lines instead of just to individual lines.

You can also store blocks of lines in buffers and reuse them in other sessions or systems. For further information, refer to Using Buffers [Page 117].

Selecting Source Code

To select	You must
One line	Position the cursor in the required line and choose *Mark* from the Editor toolbar.
A block of lines	Position the cursor in the first line of the block. Choose *Mark*. Position the cursor in the last line of the block. Choose *Mark* again.

To remove the marking from a block, choose *Edit* → *Deselect*.

Editing Source Code

Function	Procedure
Delete a block of lines	Select the block and choose *Cut.*
Copy a block of lines	Select the block and choose *Copy.* The selected block is copied to the standard buffer.
Insert a block of lines	Position the cursor where you want to insert the source code and choose *Insert.* The system inserts the block currently stored in the standard buffer
Duplicate a block of lines	Select the relevant block and choose *Duplicate line/block.*
Shift a block horizontally	Select the relevant block and choose *Shift line/block.*
Delete a line	Position the cursor on the relevant line and choose *Delete line.*
Split a line	Position the cursor in the required line and press **ENTER**.
Insert a new line	Choose *Insert line*.

Join two lines	Position the cursor within the required line and choose  *Concatenate.*
Comment out a block	Select the required block and choose *Utilities* → *Block/buffer* → *Insert comment *.*
Remove comment marks from a block	Select the required block and choose *Utilities* → *Block/buffer* → *Delete comment *.*
Print a block	Select the required block and choose *Utilities* → *Block/buffer* → *Block* → *Print*.

See also:

Using Buffers [Page 117]

Using Buffers

Use

Both modes of the ABAP Editor contain a standard buffer and three other temporary storage areas. You can also use the clipboard of your local presentation server to copy source code between programs in different SAP Systems.

Features

The following table provides an overview of the various buffers:

Function	Use with	Meaning
Copy to buffer	*Insert buffer*	Copy and insert within a single editor session in the same system (standard buffer). When you leave the editor, the system deletes the contents of the buffer.
Copy to X buffer, Copy to Y buffer, Copy to Z buffer	*Insert X buffer, Insert Y buffer, Insert Z buffer*	Copy and insert between sessions in the same system. The editor saves the contents of these buffers, so they are available from all of the sessions in the current system.
Copy to clipboard	*Insert clipboard*	Copy and insert between different SAP systems. For example, you can use this to copy material from a production system into a development system. This option uses the clipboard on your local presentation server.

From the frontend editor, you can access the buffers using the **context menu** (*block/buffer*).

Copying Source Code to a Buffer

To copy program code into a buffer, select the relevant lines in the ABAP Editor and choose *Utilities* → *Block/buffer* → *Copy to...* with the required buffer.

Inserting Source Code from a Buffer

To copy the source code from a buffer, position the cursor at the position where you want to insert it and choose *Utilities* → *Block/buffer* → *Insert*... with the required buffer. The system then inserts the source code from the buffer before the current line

Copying Source Code to the Clipboard

You can edit material while it is in the buffer by choosing *Utilities* → *Block/buffer* → *Edit buffer* with the required buffer.

Find and Replace (Frontend Editor)

Use

You can find and replace single words or any strings in the current source code. If you are searching locally, it makes no difference whether you are searching for a whole word or a part of a work. There is also no distinction between upper and lower case.
The local search in the frontend editor starts at the cursor position. Once it reaches the end of the source code, it can start again from the beginning.

You can also start a global search for any string in the source code of the corresponding main program and replace it. You can restrict the scope of a global search by setting various search options.

Procedure

Finding Any Text

1. Choose *Find/replace*, either from the context menu or  in the toolbar.
2. Enter the required string and choose the relevant option.
3. Choose *Find Next*.
   If the string exists in the text, the system positions the cursor on it.

Finding and Replacing Any Text

1. Choose *Find/replace* either from the context menu or from the application toolbar.
2. Enter the string for which you want to search and the replacement string, and set the relevant options.
3. Choose *Replace all*, or *Find next* followed by *Replace.*
   If the search string exists in the source code, it is replaced by the replacement string. If you choose *Replace all*, all instances of the search string are replaced in a single step.

Using Find and Replace to Delete Words and Strings

1. Choose *Find/replace* either from the context menu or from the application toolbar.
2. Enter the search string and set the relevant options.
3. Choose *Replace all*.
   All instances of the search string within the source code are deleted.

If you want to start a global search that extends beyond the current source code, choose Edit → Find/replace or the corresponding icon in the standard toolbar. For further details, refer to Editing Source Code in the Backend Editor [Page 111].

Search and Replace (Backend Editor)

Use

You can search for any character string, either in the current source code, or in the source code of the corresponding main program (global search). You can use search options to restrict the scope of the search. As well as searching for a particular character string, you can also search generically.

Procedure

Simple Local Search

1. Choose *Search* or  .

   The *Find and Replace* dialog box appears.

Enter your search string.

2. Choose  .

   The system searches from the current cursor position. If it finds the search string in the current source code, it positions the cursor before it.

3. To find the next occurrence of the search string, choose  .

Restricted or Generic Search

1. Choose Edit → *Find/replace*.

   The *Find and Replace* dialog box appears.

2. Enter your search string.

   If you want to run a generic search, use the `*` (asterisk) character to represent any string. For example, if you enter `st*r`, the system will stop at both `star` and `steer`.
   If you want to set a placeholder for a single character, use the + (plus sign). For example, if you entered `st+r`, the system would find `star`, but not `steer`.

   Use the pound sign (`#`) as an escape character. For example, to find the string `1+2`, enter `1#+2` as your search string.

3. If required, restrict the scope of the search using the following options:

As a string	The system stops at the search string if it is a whole word or a part of a word
As a word	The system stops at the search string only if it is a whole word
Upper-/lowercase	The system stops at the search string only if it is written with the same combination of upper- and lowercase letters

4. If you want to search the whole main program, select *In main program*.

5. Choose  .

If the system finds the search string in the source code, it displays a hit list of the locations where it was found.

6. Double-click the entry you want to see.

   The system places the cursor at the beginning of the line containing the string.

Replacing a String

1. Choose *Edit* → *Search/replace*.

   The *Find and Replace* dialog box appears.

2. Enter the search string.
3. Select the *Replace with* option and enter the replace string.
4. Enter any other options to restrict the search.
5. If you want to search the whole main program, select *In main program*.
6. Choose  .

   If the system finds the search string in the source code, it displays a hit list of the locations where it was found.

7. Choose the first entry from the list, followed by *Replace*.

   If you are sure that you want to replace the search string wherever it occurred, you can select the first location and then choose *From cursor position w/o confirmation*.

Inserting Statement Patterns

Use

The Pattern function allows you to insert various statement templates in your program. This is particularly useful with complex ABAP statements, since it saves excessive typing and ensures that the syntax of the statement you are inserting is always correct.

Features

You can insert the following statement patterns:

Statement	Explanation
CALL FUNCTION	Inserts a function call.
ABAP Objects pattern	You can insert the following basic ABAP Objects statements: CALL METHOD CREATE OBJECT RAISE EVENT
MESSAGE	A MESSAGE statement for a specified message. Enter a message ID, message type and a number. The choose **ENTER** to continue.
*SELECT * FROM*	Inserts a SELECT FROM <table> statement. Enter a table name in the field provided and choose ENTER. The system queries you for the table fields.
PERFORM	A PERFORM statement for a specified form.
AUTHORITY-CHECK	An AUTHORITY-CHECK statement for a specified authorization object. Choose **ENTER** to continue.
WRITE	A WRITE statement for a specified structure or table.
CASE	Inserts a CASE statement for a specified status.
Internal Table	Inserts an internal table. You can copy the fields or the structure of an existing table.
CALL DIALOG	A CALL DIALOG statement for a specified dialog module.
Other pattern	Inserts a predefined or user-defined ABAP statement.

Procedure

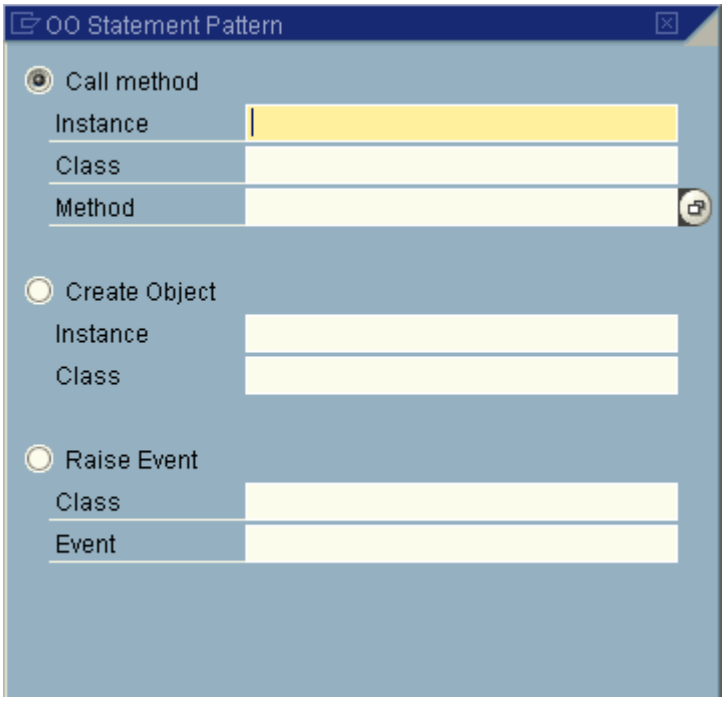
To insert a statement pattern in the ABAP Editor:

1. Ensure that you are in change mode, and place the cursor at the position where you want to insert the statement.
2. Choose *Pattern*.

The *Insert Statement Pattern* dialog box appears.

3. Choose the required pattern and , if necessary, the statement.
4. Choose ✔.
5. Fill out the pattern with the required information.

   Example: Statement pattern for ABAP Objects:



6. Choose ✔ to confirm your entries.

   The system inserts the statement at the cursor position in the program code.

   In the above example, the following lines would be inserted:

```
CALL METHOD picture->set_display_mode
    EXPORTING
      display_mode =
*    EXCEPTIONS
*      ERROR        = 1
*      others       = 2
          .
  IF sy-subrc <> 0.
*   MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
*              WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
  ENDIF.
```

Inserting Patterns Using Drag and Drop

Use

You can insert source code segments in the ABAP Editor easily using drag and drop. To do this, you choose the relevant entry from the tree display in the object list or from your worklist [Page 45] and drag it to the required position in the ABAP Editor.

You can insert patterns for

- Instantiating global classes
- Calling methods of global classes
- Calling function modules
- Calling subroutines using PERFORM.

Prerequisites

You must be using the **frontend editor** mode of the ABAP Editor.

Procedure

To insert the pattern for a method call into your source code, you would:

1. Open the program in which you want to open the pattern.
2. In the navigation area of the Object Navigator, open the class containing the relevant method.
3. Select the method in the tree display.
4. Drag the entry into the editor and drop it where you want to insert the pattern.

Result

The pattern for the method call is inserted at the required position, complete with all of the parameters. Optional parameters appear as comments.
You can now complete the pattern and assign values to the parameters.

Expanding Includes

Use

You can expand includes that you use in your programs. The system then displays the entire contents of the include within the program, allowing you to modify it without having to switch to another window.

You can only use this function in the backend editor in the ABAP Editor.

Procedure

1. Open the program containing the include statement. Make sure you are in change mode.
2. Position the cursor on the line containing the include statement.
3. Choose *Edit* → *Other functions* → *Expand include*.

   The system displays the entire contents of the include within your main program.

   The include area is delimited by comment lines that indicate the start and the end of the include.
4. Save your changes to the include.

   There are two ways to do this:

- Choose  . This saves the main program. The expanded include is then saved as part of the main program, and the include program is removed from the object list of the main program.
- Position the cursor at the beginning of the include and choose *Edit* → *Other functions* → *Save include*. In this case, the changes to the include are saved separately. If you then choose  , you only save the changes to the main program.

Result

When you expand an include, the expansion applies until you explicitly cancel it.

To cancel the expansion, position the cursor at the beginning of the include and choose *Edit* → *Other functions* → *Compress include*. If you have changed the include, the system asks if you want to save your changes.

Using ABAP Help

The help function in the ABAP Editor allows you to display information about ABAP syntax, semantics, and the individual ABAP keywords.

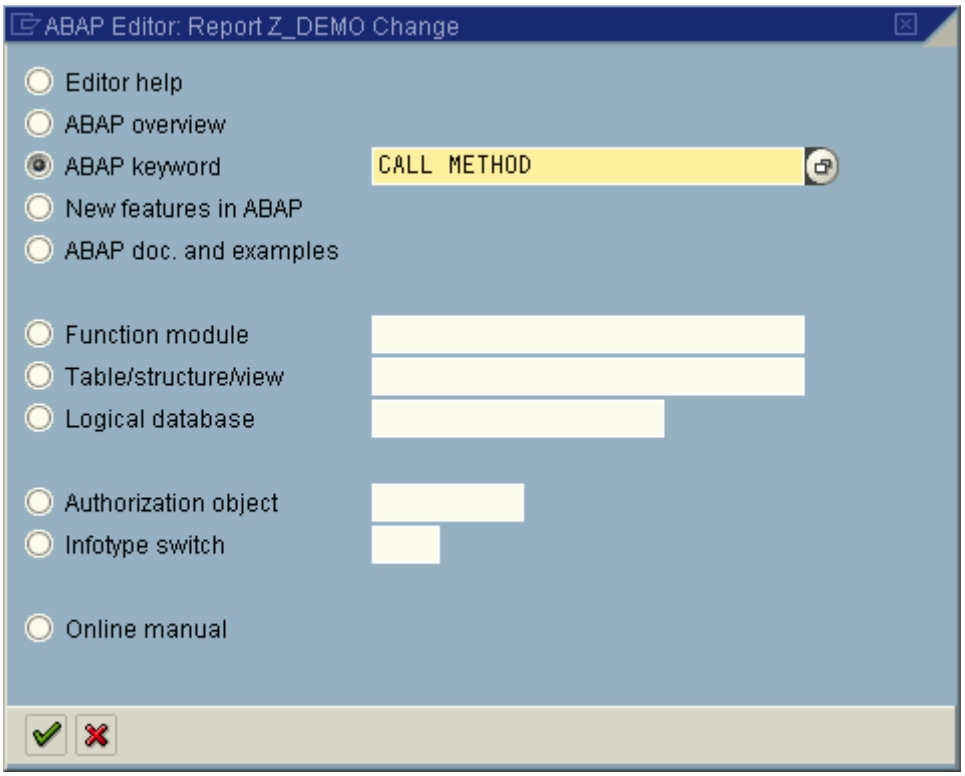
Starting the Help

You can start the help as follows:

- By pressing F1 with the cursor positioned on a keyword in the ABAP Editor
- `I` button in the ABAP Editor
- By choosing *Utilities→ Keyword doc.* from the initial screen of the ABAP Editor
- By starting transaction ABAPHELP

By selecting the bottom node in transaction ABAPDOCU

You can display help for the following areas:



You can also create an **offline version** of the ABAP keyword documentation on your frontend.
The entire keyword documentation (including navigation structure) will be downloaded in HTML format into a directory on your PC. This takes between 10 and 20 minutes. You can then open the file BASIC.HTM. The offline documentation only

supports links to other topics that were also included in the download, but it can be used independently of the R/3 System.

Improving the Layout

Use

ABAP source code is stored in the database exactly as you enter it in the ABAP Editor. The display is **not** automatically standardized (for example, conversion to uppercase characters). However, you can use the **Pretty Printer** to standardize the layout of your program to one of the following display variants:

- Entire program in uppercase letters
- Entire program in lowercase letters
- ABAP keywords in uppercase

The Pretty Printer also groups together statements that logically belong together by indenting statement blocks.

See also:

Features of the Pretty Printer [Page 131]

Note that converting the ABAP keywords to uppercase can be particularly runtime-intensive, especially with long programs.

Do not use the Pretty Printer until you are satisfied that there are no more syntax errors in the source code.

Procedure

Standardizing the layout

Function	Procedure
Uppercase display	Choose *Utilities* → *Settings*. Under Pretty Printer, set the *Uppercase* option, and confirm the settings. Choose *Pretty Printer.*
Lowercase display	Choose *Utilities* → *Settings*. Under Pretty Printer, set the *Lowercase* option, and confirm the settings. Choose *Pretty Printer.*
Keywords in uppercase	Choose *Utilities* → *Settings*. Under Pretty Printer, set the *Keyword large* option, and confirm the settings. Choose *Pretty Printer.*

Indenting blocks

Function	In textedit control mode	In table control mode
Indenting a block	Select the relevant lines and press **Tab**. The system indents the selected lines.	Position the cursor at the point in a line up to which you want to indent. Choose (Indent line/block). Repeat for each line you want to indent.
Removing the indentation for a block	Select the relevant lines and press Shift + Tab.	Procedure: Same as for indentation

Result

When you save, your ABAP source code is stored in the database with all of the layout changes make by the Pretty Printer. When you reload the program, it is displayed exactly as it was stored in the database, regardless of the editor.

Features of the Pretty Printer

You can use the Pretty Printer function to standardize the layout of your program. This function arranges associated key words in groups and indents individual statements clearly. The Pretty Printer ensures that your program layout meets the guidelines described in the ABAP User's Guide.

Features

The Pretty Printer performs the following functions:

- Standardizes the source code display according to the options you set (upper-/lowercase).
- Places event, control, and INCLUDE key words on separate lines.
- Moves event key words, FORM statements, and MODULE statements to the beginning of the line. In this case, the Pretty Printer function uses the first program line as a reference.
- Inserts a blank line before event key words that are not preceded by a blank or a comment line.
- Indents all command lines and control structures associated with an event by 2 spaces.
- Locates commands that extend beyond one line and writes subsequent commands on a new line.
- Inserts appropriate comment blocks before FORM and MODULE statements that do not have comments. The Pretty Printer function uses the routine name and the USING parameter to fill in the comment blocks.
- Left-justifies comments that appear in command lines. Left-justification is only performed on comment lines of 32 characters or less that begin with " (double-quotes). Pretty Printer starts the comment at column 40.

The Pretty Printer does not break down loops and control processing blocks that are contained on a single line. Also, the Pretty printer does not separate statements associated with a WHEN condition if the WHEN statement is contained on one line. To call the pretty printer, select *Program → Pretty Printer*.

Saving and Activating Programs

Saving a Program

When you choose Save, the system saves an inactive version of the current program and adds it to the user's list of inactive objects.

The system saves your program in the database. There is no syntax check when you save.

To save the contents of the ABAP Editor, choose  from the standard toolbar.

Restoring a Program After a System Crash

If the system crashes while you are working on a program, the system tries to save the program in temporary storage. The next time you try to edit the program, the system asks you which version you want to use – the version from the database or the temporary version.

If necessary, you can use the *Compare* function to compare the two versions before deciding which version to restore.

Activating a Program

When you activate a program, the system generates an active version from the inactive version. The activation process checks for syntax errors in your program, then generates the active version. Finally, it generates a load version and deletes the corresponding entry from your list of inactive objects.

To activate a program, choose or *Program* → *Activate*.

When you activate the program, the activation does not apply to all of the program components. Screens and GUI statuses are independent transport objects, and have to be activated separately.

Generating a Program

When you generate a program, the system creates a new load version from the active version. Unlike activation, this operation only generates a new load version.

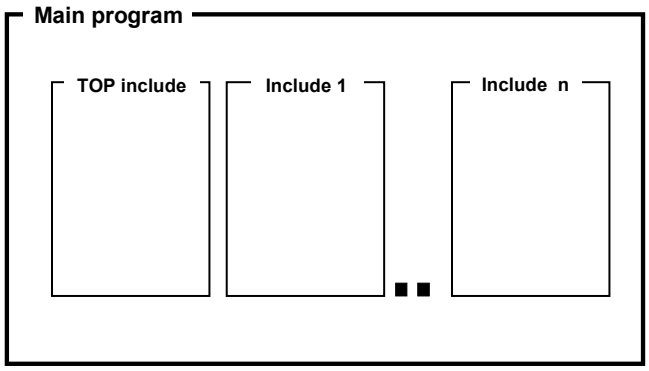
To generate the program, choose *Program* → *Generate*.

See also:

Status Display for Development Objects [Page 494]

Checking Programs

There is a range of checks that you can use in the ABAP Editor. Different checks include different parts of the program. You can imagine an ABAP program as a framework containing a set of components, for example, the top include and PBO and PAI modules.



If you want to check a program or only a component of the program, you can restrict the scope of the check.

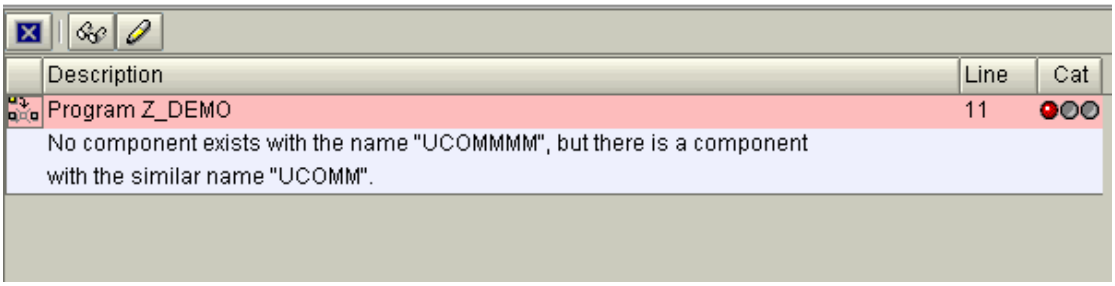
Features

Function	Description
Syntax	The system checks the syntax of the selected program. To start the syntax check ▪ Choose from the application toolbar, if the program is currently displayed in the ABAP Editor. ▪ Select the relevant program in the object list and choose *Check* → *Syntax* from the context menu. This function checks the syntax of the selected program and the top include (if the program has one). It does not check any other includes.
Main program	The system checks the syntax of the whole program and all of its includes. To start this check, select the main program in the object list and choose *Program* → *Check* → *Main program*.
Extended program check	The system checks the entire program and allows you to run additional checks with a greater scope than the normal syntax check. See also: Extended Program Check [Page 135].

Checking Programs

From Checks to Corrections

In the *Syntax* and *Main program* checks, the system displays error messages in a separate pane as soon as they occur:



If the system finds an error during the check for which it can suggest a solution, a special icon appears next to the entry in the list. It also places the cursor on the first error in the list. Choose this icon to correct the error automatically.

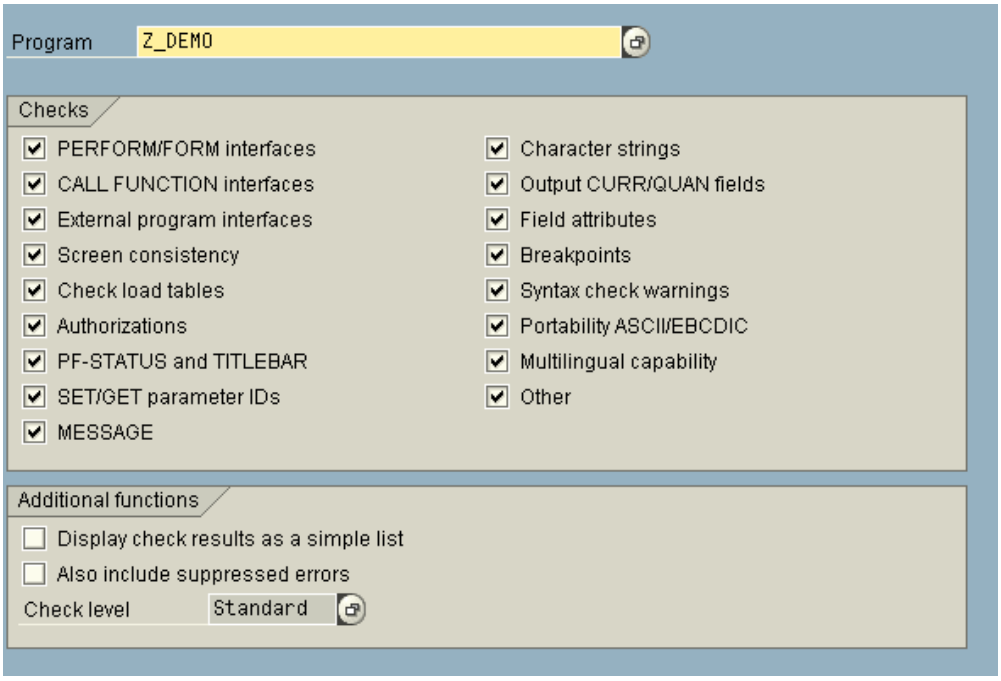
See also:

Extended Program Check [Page 135]

Extended Program Check

Process

1. Choose *Program* → *Check* → *Extended program check*. The system displays a list of possible check options:



   By default, all of the options are selected. For detailed information about a particular option, place the cursor on it and choose `F1`.

   The extended program check takes considerably longer than other checks. However, it uses a buffer, so after the first check, you will find that it speeds up.

2. When you start the extended program check, the system displays an overview of the errors, warnings, and messages that have been generated. For each option that you included in the check, there is an overview of the number of errors, warnings, and messages for that category.

Extended Program Check

Check for program Z_DEMO	Error	Warnings	Messages
Fatal errors	1	0	0
PERFORM/FORM interfaces	0	0	0
CALL FUNCTION interfaces	0	0	0
External program interfaces	0	0	0
Screen consistency	0	0	0
Authorizations	0	0	0
PF-STATUS and TITLEBAR	0	0	0
SET/GET parameter IDs	0	0	0
MESSAGE	0	0	0
Character strings	0	0	0
Output CURR/QUAN fields	0	0	0
Field attributes	0	0	0
Breakpoints	0	0	0
Syntax check warnings	0	0	0
Portability ASCII/EBCDIC	0	0	0
Check load sizes	0	0	0
Multilingual capability	0	0	0
Other	0	0	0
Hidden errors and warnings	0	0	0

3. If you choose an entry, the system displays the corresponding detail screen.
4. From this detail display, you can jump to the appropriate point in the program and correct it.

Maintaining Text Elements

Text element maintenance is a tool in the ABAP Workbench that makes it easier to maintain program texts in different languages. Any text that a program displays on the screen can be maintained as a text element.

Contents

If you want to use text elements in conjunction with the Modification Assistant, refer to the Modifying Text Elements [Ext.] documentation.

Maintaining Text Elements: Overview

Use

Use this tool to create, maintain, and translate text elements for your program.

Text elements include:

- List and column headers that appear in ABAP lists
- Selection texts on selection screens
- Text symbols that you use with the WRITE statement.
  See also the text symbol [Ext.] documentation in the ABAP User's Guide.

Text elements are stored separately from the program in language-specific **text pools**. Your program automatically uses the text elements in the user's logon language.

You can create and maintain text elements without having to change the source code of your program. You can also create standard text elements that you can copy and use in other programs. If you work exclusively with text symbols, and do not hard-code any texts in WRITE statements, your programs will be fully **multilingual**. All you then need to do is to translate the text elements from their original language into the required foreign languages.

Translators can use the ABAP Workbench to translate the text pool of the original language into other languages.

Features

- List and column header creation and maintenance
- Selection text and text symbol creation and maintenance
- Selection text and text symbol comparison
- Text element copy
- Text element translation

Starting the Text Element Maintenance Tool

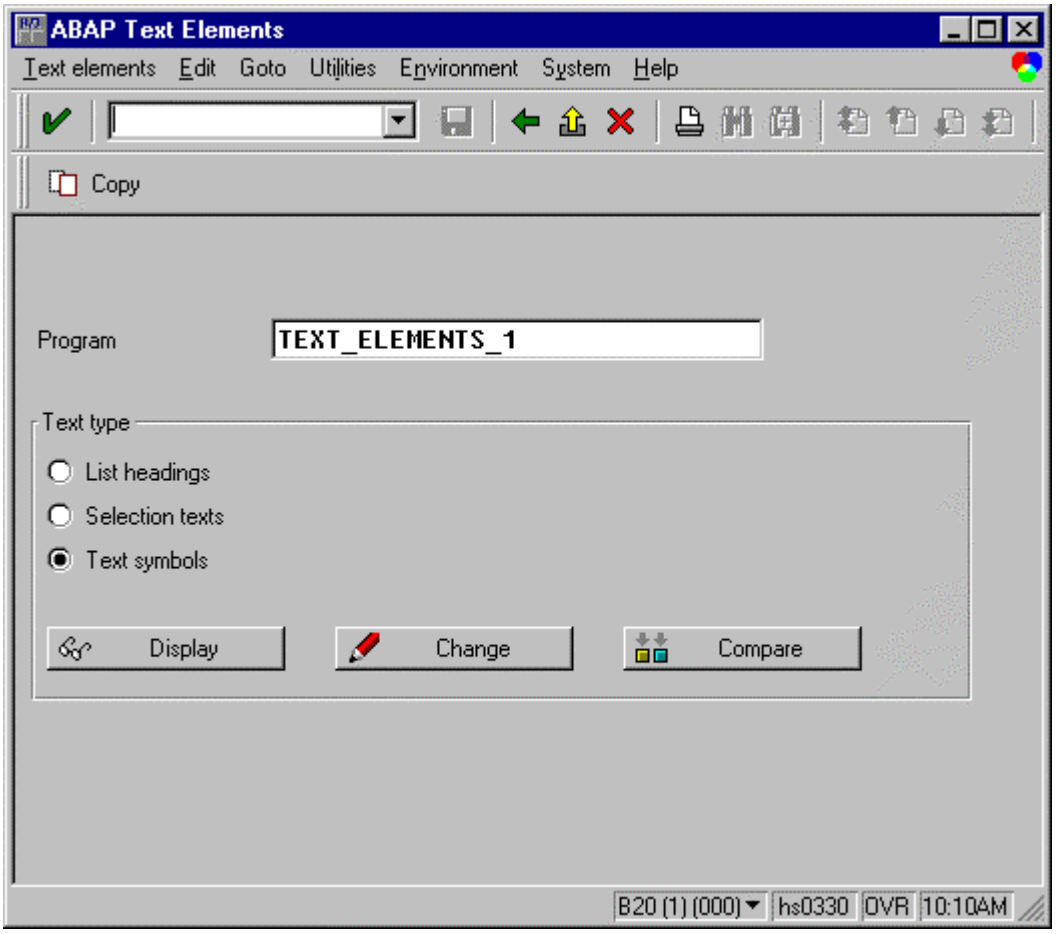
See the initial screen [Page 142] documentation.

Initial Screen

There are several ways of starting the text element maintenance function:

- From the R/3 initial screen, choose *Tools* → ABAP *Workbench* → *Development* → *Programming environ.* → *Text elements.*
- From the Repository Browser, choose the Program objects for a program and select Text elements.
- From the ABAP Editor, choose *Goto* → Text elements.
- From the initial screen of the ABAP Editor, enter the program name. Then, in the *sub-objects* group box, select *Text elements*, and chose *Display* or *Change*.

All of the above lead to the ABAP Text elements screen:



Creating and Maintaining Text Elements

If you are working on a program in the ABAP Workbench and are logged onto the system in a language other than that in which the program was created, the following happens:

- **In display mode**, the system warns you, using a message in the status bar, that the original language and logon language are different. Any text elements that exist in the logon language are displayed in that language, otherwise, they appear in the original language. The language key is displayed in the last column. You can use this method to find text elements that still require translation.
- **In change mode**, the system asks whether you want to maintain the text elements in the original language or change the original language of the program. If you change the language and not all of the text elements yet exist in that language, the system fills the missing entries with the texts from the old original language.

The following sections explain how to create and maintain different kinds of text elements:

Creating List and Column Headers [Page 144]

Maintaining Selection Texts [Page 146]

Maintaining Text Symbols [Page 148]

You cannot create text elements in INCLUDE programs [Ext.] (type I).

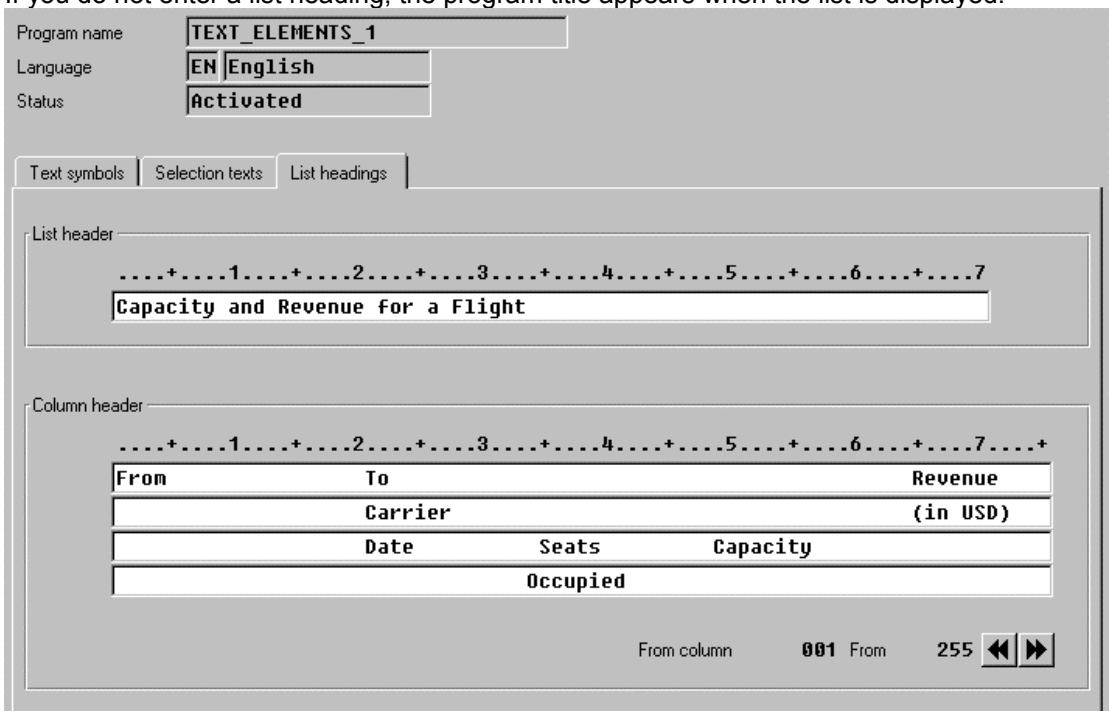
Creating List and Column Headings

When you create a list in a program, you can also create your own list and column headings

Procedure

From the initial screen of the ABAP Editor:

1. Enter the program name.
2. Select *Text elements* and choose *Display* or *Change.*
3. Select *List headings* and choose *Change*.
4. The list heading can be up to 70 characters long. Each of the four columns of the Column heading field can be up to 255 characters long.
   If you do not enter a list heading, the program title appears when the list is displayed:



5. Save your entries.

In the list and column headings, you can enter up to 10 placeholders &0 to &9, each followed by up to 18 periods. The system replaces the placeholders in the TOP-OF-PAGE event with the contents of the system fields SY-TVAR0 to SY-TVAR9. The displayed length of the system fields is the length of the placeholder plus the following periods. For example, a placeholder "&3……" would display the contents of SY-TVAR3 with length 8 characters.

Result

If you created the headers as displayed above, the display would look like this:

Capacity and Revenue for a Flight

From	To			Revenue
	Carrier			(in USD)
	Date	Seats Occupied	Capacity	
NEW YORK				
	SAN FRANCISCO			
	AA 0017			
	28.08.1998	34	660	15.996,32
	30.09.1998	8	660	3.909,19
	19.11.1998	0	660	0,00
	22.11.1998	95	660	45.836,66

Maintaining Selection Texts

You can replace the standard texts that appear next to input fields on selection screens with text elements. You can either use a short text defined in the ABAP Dictionary or create your own texts.

Prerequisites

You must have defined one or more selection screens for your program.

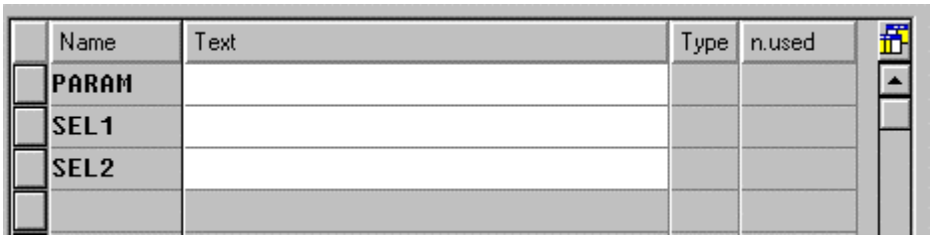
Coding example:

```
PROGRAM TEXT_ELEMENTS_3.

TABLES SBOOK.

PARAMETERS: PARAM(10).

SELECT-OPTIONS: SEL1 FOR SBOOK-CARRID,
                SEL2 FOR SBOOK-CONNID.
```

Procedure

From the ABAP Editor in change mode:

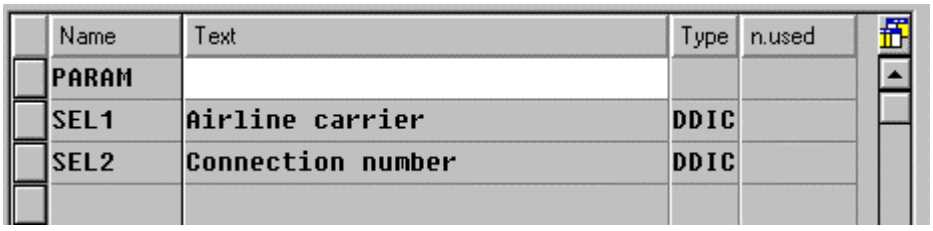
1. Choose *Goto* → Text *elements* → *Selection texts*.
   A table appears in which you can create or edit your selection texts:



Name	Text	Type	n.used
PARAM			
SEL1			
SEL2			

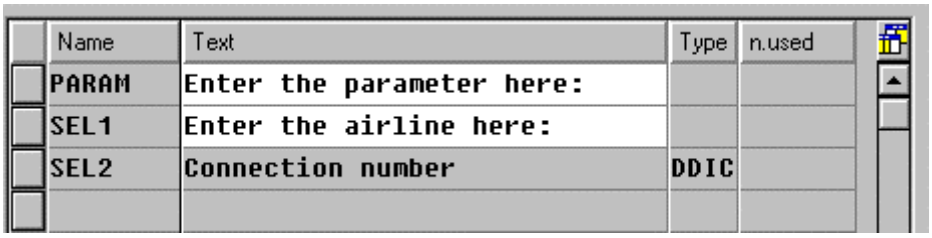
   The names of your parameters and selection options appear automatically in the *Name* column (as illustrated above). Each may have a selection text of up to 30 characters.

2. To use texts from the ABAP Dictionary for your selection texts, position the cursor on the relevant line and choose *Utilities* → *Copy DD text*.
   To use Dictionary texts for all fields for which they exist, choose *Utilities* → *Copy all Dictionary texts*.
   The system automatically fills the selection texts with the short texts from the ABAP Dictionary. The *Type* is automatically set to DDIC. Text elements that contain ABAP Dictionary texts appear in display mode.



Name	Text	Type	n.used
PARAM			
SEL1	Airline carrier	DDIC	
SEL2	Connection number	DDIC	

3. Enter texts for the remaining parameters for which you did not use the ABAP Dictionary text (or for which none existed).
4. To change an ABAP Dictionary text, position the cursor on the corresponding line and choose *Utilities* → *No DD text*.
5. Change the selection text.

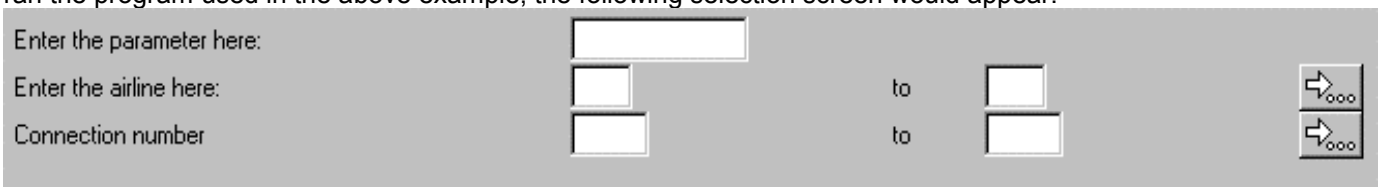


Name	Text	Type	n.used
PARAM	Enter the parameter here:		
SEL1	Enter the airline here:		
SEL2	Connection number	DDIC	

6. Save your entries.

Result

The selection texts for the program are inserted in the text pool for the relevant language. If you ran the program used in the above example, the following selection screen would appear:



If you change or delete parameters or selection options in your program after you have saved the selection texts, a flag appears in the *n.used* column next to the relevant text the next time you start the selection text maintenance function. This enables you to find and delete obsolete selection texts. If you try to delete a selection text that is still in use in the program, the system displays a warning. When you save the selection texts, the system again informs you that there are unused texts.

Maintaining Text Symbols

Text symbols are special text constants that you enter and maintain independently of the program code. In the final versions of your programs, you should use text symbols instead of hard-coded texts. This makes the programs **language-independent** and easier to maintain.

For further information about literals and text symbols, refer to literals [Ext.] and text symbols [Ext.] in the ABAP User's Guide.

Prerequisites

You must assign a three-character ID to each text symbol. You define this in the WRITE statement as follows:
WRITE... TEXT-<idt>...

When you run the program, the system searches in the text pool for a text symbol with the ID <idt> and displays it. If it does not find text symbol <idt>, it ignores that part of the WRITE statement.
The ID may not begin '%_', and may not contain spaces.

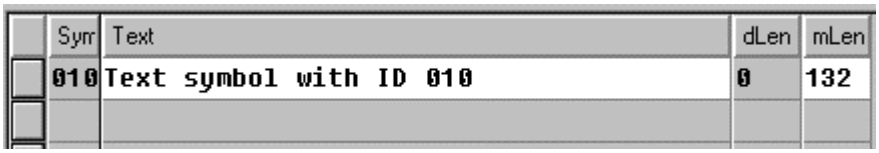
The rest of this section is based on the following coding example:

```
PROGRAM TEXT_ELEMENTS_2.

WRITE:    TEXT-010,
        / TEXT-AAA,
        / TEXT-020,
        / 'Default Text 030'(030),
        / 'Default Text 040'(040).
```

Procedure

From the ABAP Editor:

1. Enter your WRITE statement with the three-character ID and default texts.
2. Double-click one of the entries in the WRITE statement.
3. If the text symbol does not yet exist, a dialog box appears. Confirm that you want to create the text symbol.
   The text symbol maintenance screen appears.
4. Enter the text.
   It may be up to 132 characters long.



	Sym	Text	dLen	mLen
	010	Text symbol with ID 010	0	132

5. Save the text symbol.
   The maximum length *mLen* is automatically set to the defined length *dlen* (actual length of the text).

Result

The text symbol is included in the text pool in the relevant language. You can now carry on and create other text symbols, or change existing ones.

Other Functions

Creating Further Text Symbols

1. Choose *Edit* → Append new lines.
2. In the *Sym* column, enter a three-character ID for the new text symbol, and enter the text in the Text column.
3. Save the text symbol.
   The maximum length mLen is automatically set to the defined length dlen (actual length of the text).

Sym	Text	dLen	mLen
010	Text symbol with ID 010	23	23
030	Text symbol with ID 030	23	23
AAA	Text symbol with underscores_____	33	33

Spaces in text symbols are no longer represented by underscores. This means that you can now output underscores as characters in a text symbol.

Using Text Symbols with Text Literals

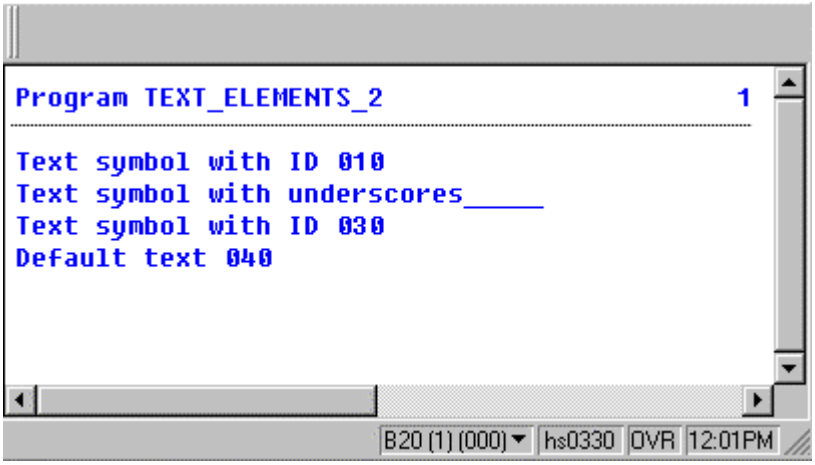
You can link a text symbol to a text literal as follows:

```
WRITE... '<textliteral>'(<idt>)...
```

If the text symbol <idt> exists, the system uses it. If it does not exist, the system uses the text literal <textliteral>.

For an example, see above.

If we did not create text symbols with the IDs "020" and "040" in the above example, the system would display the following:



Here, the system ignores the WRITE statement for the missing text symbol 020, and uses the default text (defined as a literal in the WRITE statement) for text symbol 040, since there is no text symbol defined for it in the text pool.

Deleting Text Symbols

To delete a text symbol, select the appropriate line and choose *Delete*.

Analyzing Text Elements

The text element maintenance function allows you to compare text elements against the program source code. Choose the *Analyze* function, either from the initial screen or from the maintenance screen.

You can only use the *Analyze* function with selection texts and text symbols.

See also:

Analyzing Selection Texts [Page 152]

Analyzing Text Symbols [Page 153]

Analyzing Selection Texts

The Analysis function allows you to find missing or obsolete selection texts.

It is more practical than the normal selection text maintenance function for this task, but does not let you include texts from the ABAP Dictionary.

Prerequisites

You must have defined one or more selection screens for the relevant program.

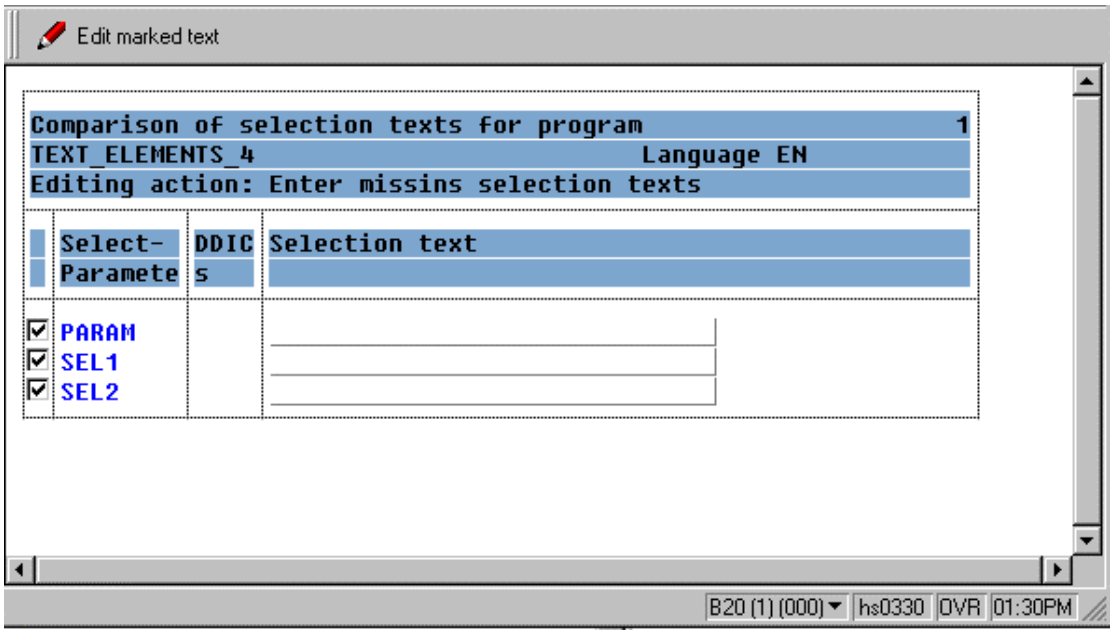
Suppose we have the following program for which no selection texts have yet been defined:

```
PROGRAM TEXT_ELEMENTS_4.

TABLES SBOOK.

PARAMETERS: PARAM(10).

SELECT-OPTIONS: SEL1 FOR SBOOK-CARRID,
                SEL2 FOR SBOOK-CONNID.
```

Procedure

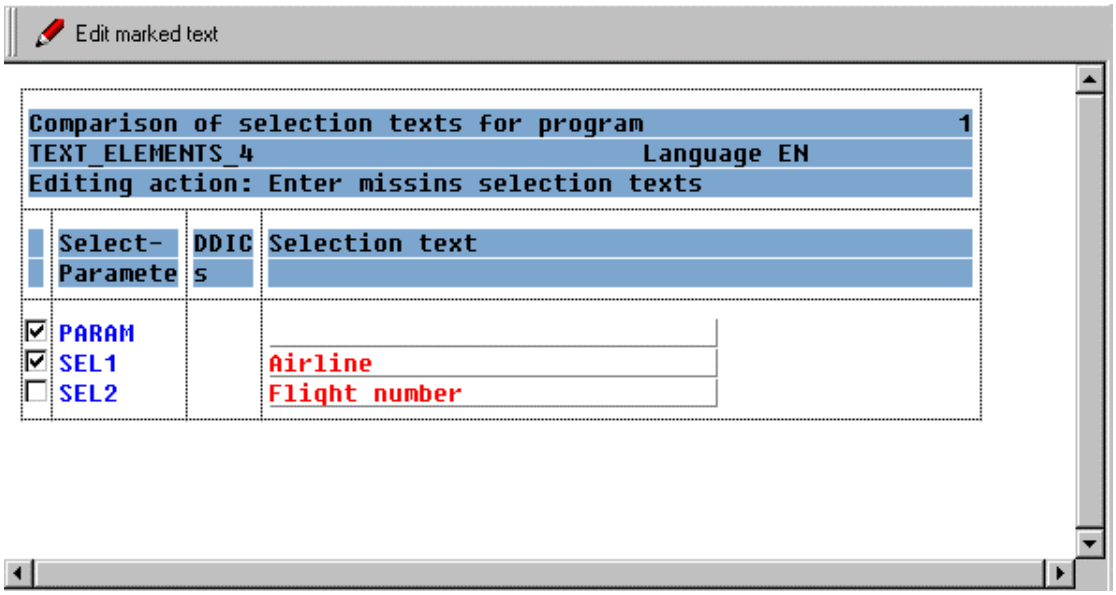
From the ABAP Editor:

1. Go to the text element maintenance initial screen [Page 142].
2. Select *Selection texts* and choose *Analyze*.
   If you have not yet defined any selection texts, the following screen appears:



This shows you the parameters and selection options for which you have not yet entered any selection texts.

3. If you do not want to use texts from the ABAP Dictionary, enter your texts.
4. Select the lines that you want to analyze.



5. Choose *Edit marked text*.
   The changed lines **and** the selected lines are included in the analysis.
6. If you now choose Save, the selection text for SEL1 is included in the text pool.

Result

In this case, both SEL1 and SEL2 were changed, but only SEL1 was selected, so only SEL1 is included in the analysis.

You can edit the selection texts for all of the input fields on a selection screen using the *Analyze* function.

If you want to use ABAP Dictionary texts as selection texts, you must use the text element maintenance function as described under Maintaining selection texts [Page 146].

Analyzing Text Symbols

When you enter text symbols in the source code of your program, they are not automatically entered in the text pool. To keep the list up to date and avoid discrepancies, use the Analyze function.

This allows you to

- Delete obsolete text symbols from the text pool
- Adopt new text symbols in the text pool

  This is important for text symbols that are used in the program, but not yet included in the text pool
- Compare text symbols with the program code

Prerequisites

Suppose we have the following program for which no text symbols have yet been maintained:

```
PROGRAM TEXT_ELEMENTS_4.

WRITE:    TEXT-010,
         /'Default Text'(020),
         /TEXT-030.
```

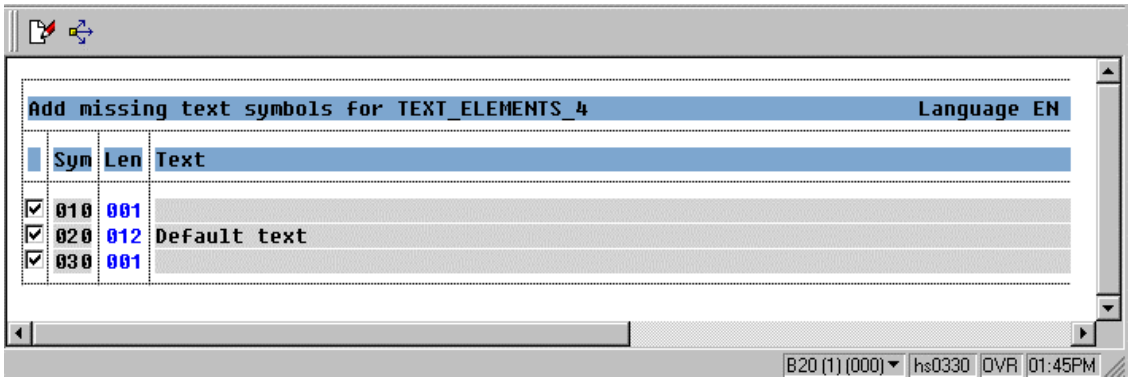
Procedure

From the ABAP Editor:

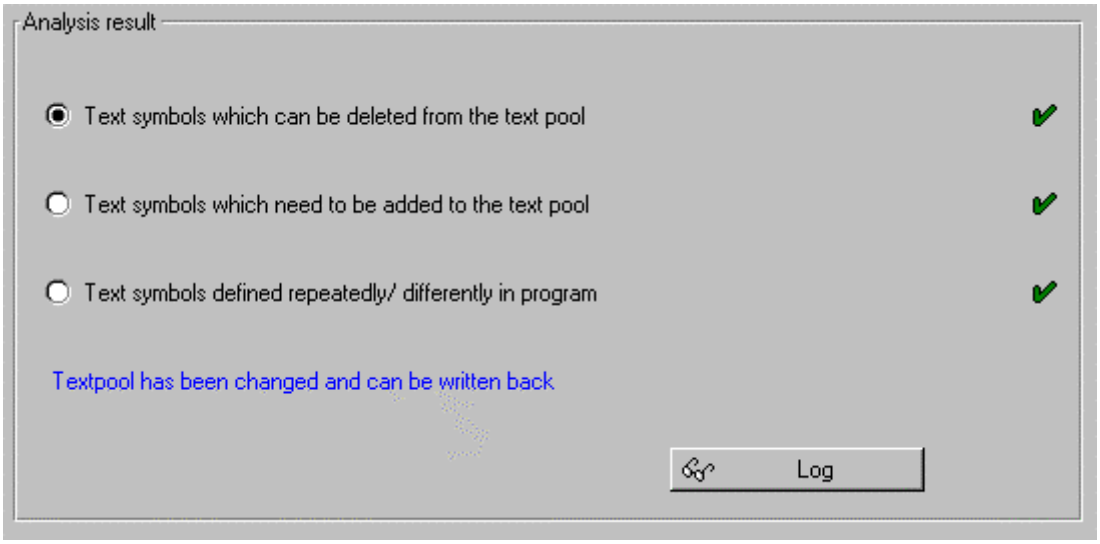
1. Go to the text element maintenance initial screen [Page 142].
2. Select Text symbols and choose Analyze.
   If you have not yet maintained any text symbols for the program, the following screen appears:

Since the list of text symbols is empty, the second option is selected.

3. Choose *Edit*.
   The following screen appears:



4. Select the text symbols that you want to include in the text pool.
5. Choose *Insert text symbol*.
   The selected text symbols are marked for transfer into the text pool. The following screen appears:



6. Use the *Log* function to display the changes again.
7. You can adopt the changes by choosing *Save*, or cancel them by choosing *Undo*.

Result

If we choose Save in our example, the system inserts the text symbols "010", "020", and "030" in the text pool. No texts are assigned to symbols "010" or "030", but the text literal defined in the program is assigned to symbol "020".

Other Functions

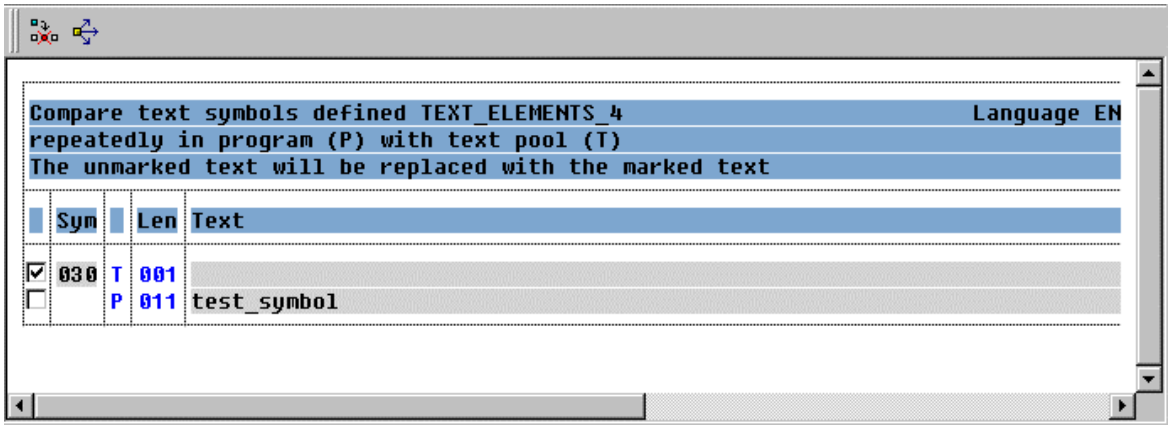
You can change the above program slightly to use other analysis functions.

```
PROGRAM TEXT_ELEMENTS_4.
WRITE:    TEXT-010,
         /'Default Text'(020),
         /'Test_Symbol'(030).
```

Comparing Text Symbols with the Source Code

You can compare texts that are defined differently in the program and in the text pool as follows:

1. Select Text symbols defined *repeatedly/differently in program*.
2. Choose *Edit*.
   In our example, the following screen would appear:



You can now replace the empty text from text symbol 030 in the text pool with the program text "test_symbol".
The third column indicates whether the text is defined in the text pool (T) or in the program (P).

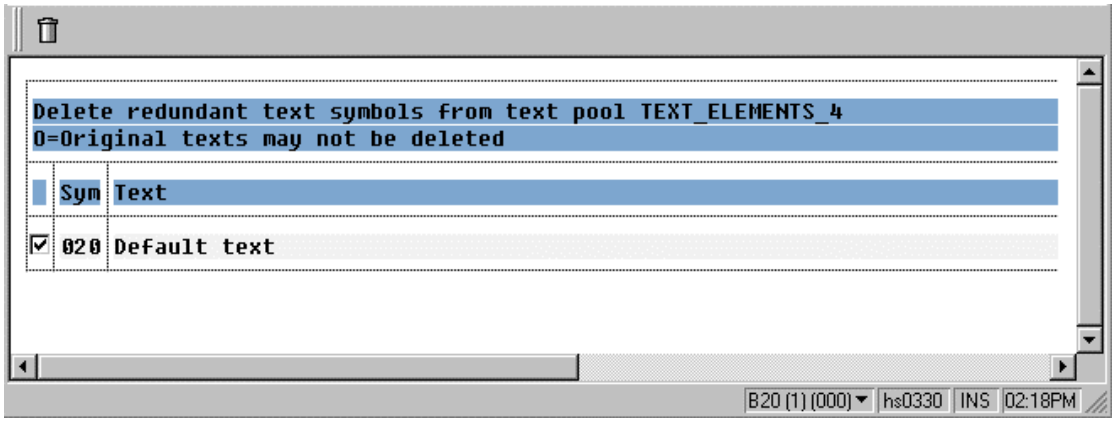
3. Choose *Replace*.
4. *Save* your entries.

Deleting Text Symbols from the Text Pool

Before *deleting* a text symbol, check its where-used list.

To delete text symbols from the text pool in the Analysis results:

1. Select the first option: *Text symbols which can be deleted from the text pool.*
2. Choose *Edit*.



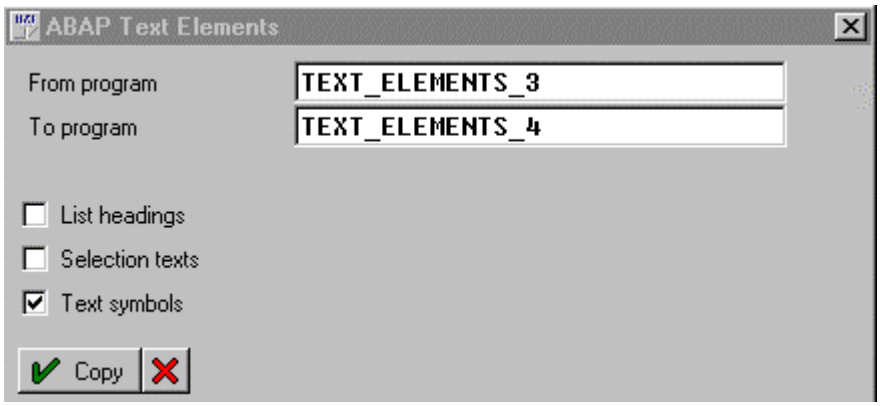
3. If text symbol "020" is no longer needed in the program, you can delete it.
4. Choose *Delete*.

Copying Text Elements

You can copy text elements from one ABAP program to another. The copy function allows you to copy sets of standard text elements.

Procedure

1. Go to the text element maintenance initial screen [Page 142].
2. Choose *Copy*.
   The following dialog box appears: :



3. Enter the name of the target program.
4. Select the text elements that you want to copy.
5. Choose *Copy.*

Result

The selected text elements are copied from the source program to the target program.

Translating Text Elements

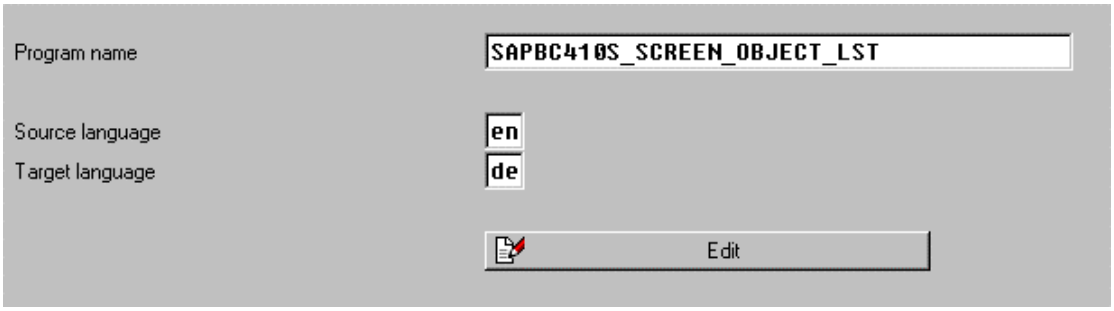
Text elements help you to write language-independent programs. They are stored in the text pool of their language, and can be translated using the normal translation process.

Prerequisites

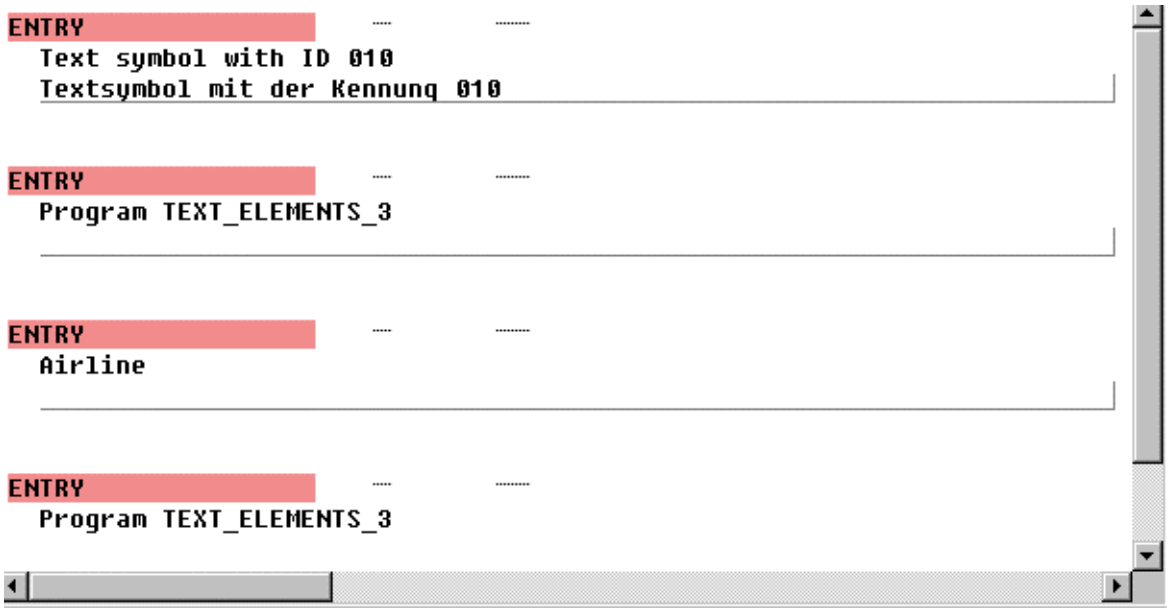
You have maintained text elements in the original language.

Procedure

1. Choose *Utilities* → *Translation* → *Short and long texts*.
   The initial screen of the translation transaction (**SE63**) appears.
2. Choose *Translation* → Short *texts* → *Program texts*.
   The following screen appears:



3. Enter the program name, the source language, and the target language.
4. Choose *Edit*.
   The text elements are displayed in the language in which you created them.



You can also access the translation tool from the text element maintenance screen by choosing *Goto* → *Translation*.

5. Translate the texts.
6. *Save* your translation.

Result

You have created text pool for a different languages.

Once you have created text pools for different languages, you can change the language in which you run the program by changing one of the following:

- **The logon language.** The default language for your program is the logon language of the user.
- **The SET LANGUAGE statement.** This ABAP statement allows you to set the output language explicitly and independently of the logon language.

  Syntax

  SET LANGUAGE < lg >.

  The language <lg> can be a literal or a variable.

Once you have set a language (using either method), the system only looks in the text pool of that language. If it cannot find the relevant text symbols in that pool, it displays the default text specified in the program source code (if one exists), otherwise, it skips the corresponding WRITE statement.

Variants

Variants allow you to save sets of input values for programs that you often start with the same selections. You can use them for any programs except subroutine pools (type S).

Contents

Variants: Overview

Use

Whenever you start a program in which selection screens are defined, the system displays a set of input fields for database-specific and program-specific selections. To select a certain set of data, you enter an appropriate range of values.
For further information about selection screens, see Working with selection screens [Ext.] in the ABAP User's Guide.

If you often run the same program with the same set of selections (for example, to create a monthly statistical report), you can save the values in a selection set called a **variant**.

You can create any number of variants for any program in which selection screens are defined. Variants are assigned exclusively to the program for which they were created.

You can also use variants to change the appearance of the selection screen by hiding selection criteria. This is particularly useful when you are working with large selection screens on which not all of the fields are relevant.

Reports, module pools, and function groups may have several selection screens. It is therefore possible to create a variant for more than one selection screen.

Variants are an interface between the user and the selection screen. They can be used both in dialog and in background mode, although their uses are slightly different.

Variants in Dialog Mode

In dialog mode, variants make things easier for the user, since they save him or her from continually having to enter identical values. They can also make the selection screen easier to read, because you can use them to hide input fields. Running an executable program with a variant containing an optimal set of values also reduces the capacity for user error. The optimized database selections speed up the runtime of the program.

Variants in Background Mode

Variants are the only method for passing values to a report program in a background job. Therefore, when you run a program in the background, you must use a variant (or SUBMIT... VIA JOB). To avoid you having to create a new variant each time you run the report, ABAP contains a mechanism allowing you to pass variable values to variants. See variable values in variants [Page 1 6].

To ensure that an executable program is always started using a variant, you can specify in the program attributes that the program may only be started in this way.

Features

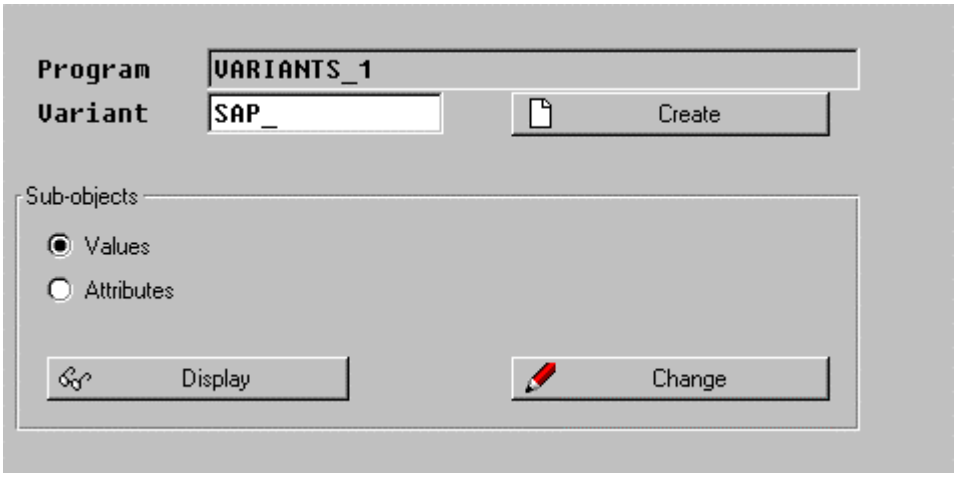
- Creation of variants
- Display, change, copy, print, and delete variants
- Use and definition of variables in variants
  - Variable date calculation

Variants: Overview

- User-specific fixed values
- Fixed values in table TVARV

Initial Screen

You access the variant maintenance tool from the initial screen of the ABAP Editor. Enter the name of the program, select *Variants* in the *Sub-objects* group box, and then choose *Display* or *Change*.



Functions

The above screen allows you to:

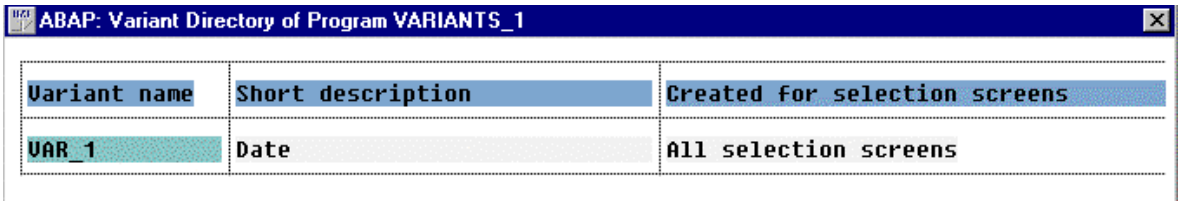
- Create variants
- Display the variant directory
- Display and change values and attributes
- Copy, delete, and rename variants

Displaying an Overview of Variants

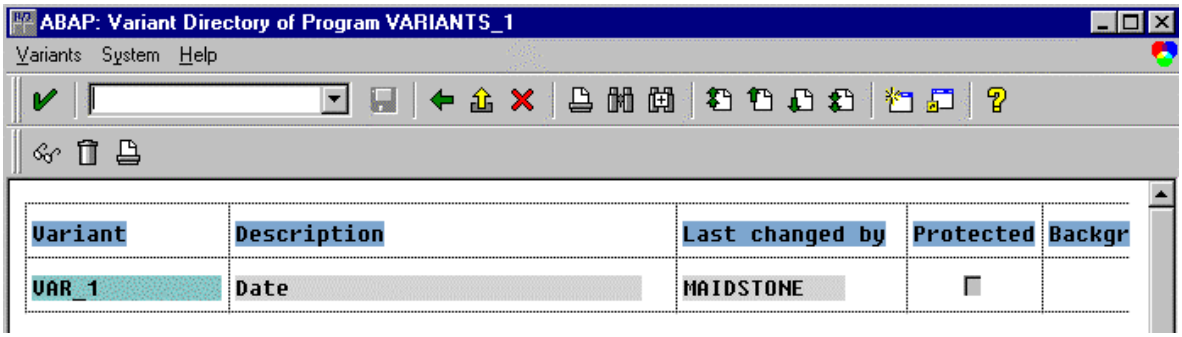
Before creating a new variant for a program, you should check whether you can use or adapt an existing variant instead.

There are two ways to display variants:

- Position the cursor on the *Variant* field on the initial screen and press F4. The following dialog box lists all of the available variants:



- Choose *Variants* → *Directory* on the initial screen:



Creating and Maintaining Variants

Contents

Creating Variants

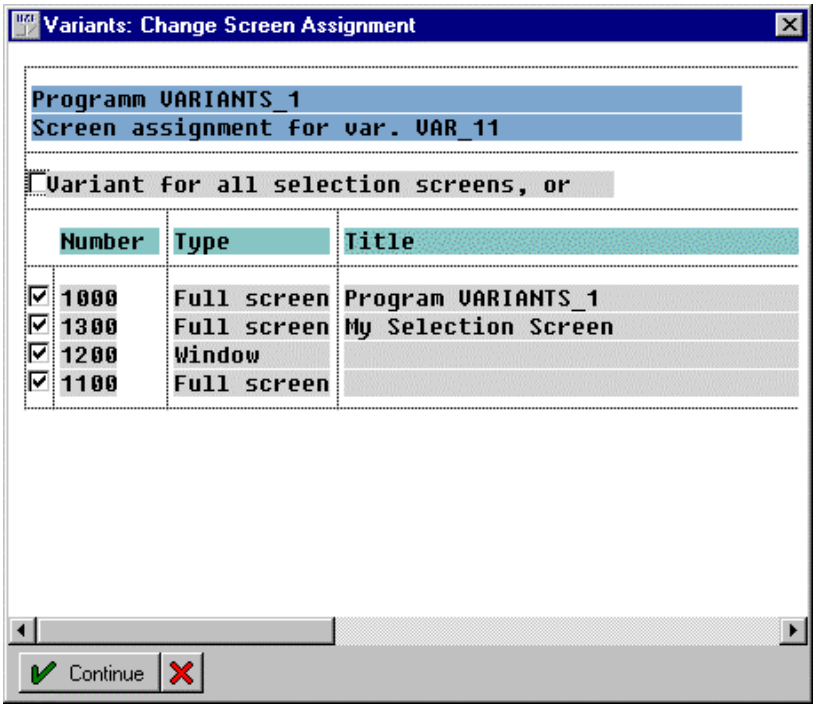
Prerequisites

You must have defined one or more selection screens for the relevant program. The program may have any type except type S.

Procedure

1. On the initial screen of the ABAP Editor, enter the name of the program for which you want to create a variant, select *Variants* in the *Sub-objects* group box, and choose *Change*.
2. On the variant maintenance initial screen [Page 165], enter the name of the variant you want to create.
   Note the naming convention for variants (see below).
3. Choose *Create*.
   If the program has more than one selection screen, a dialog box appears in which you can assign the variant to one or more screens. The dialog box does not appear if the program only has one selection screen. In this case, the selection screen of the program appears straight away.
4. If there is more than one selection screen, select the screens for which you want to create the variant.

   Example:



If you choose Variant for all selection screens, the variant also applies to any selection screens that you create **after** creating the variant.

Otherwise, the variant only supplies values to the selection screens that you select in the list.

5. Choose *Continue*.
   The (first) selection screen of the program appears.
   If your program has more than one selection screen, use the scroll buttons in the left-hand corner of the application toolbar to navigate between them. If you keep scrolling forwards, the *Continue* button appears on the last selection screen.
6. Enter the required selections, including multiple and dynamic selections.
7. Choose *Continue*.

Result

When you have finished, an overview screen appears (ABAP: Save Attributes of Variant), on which you can enter the attributes of your variant [Page 1 0] and save it.

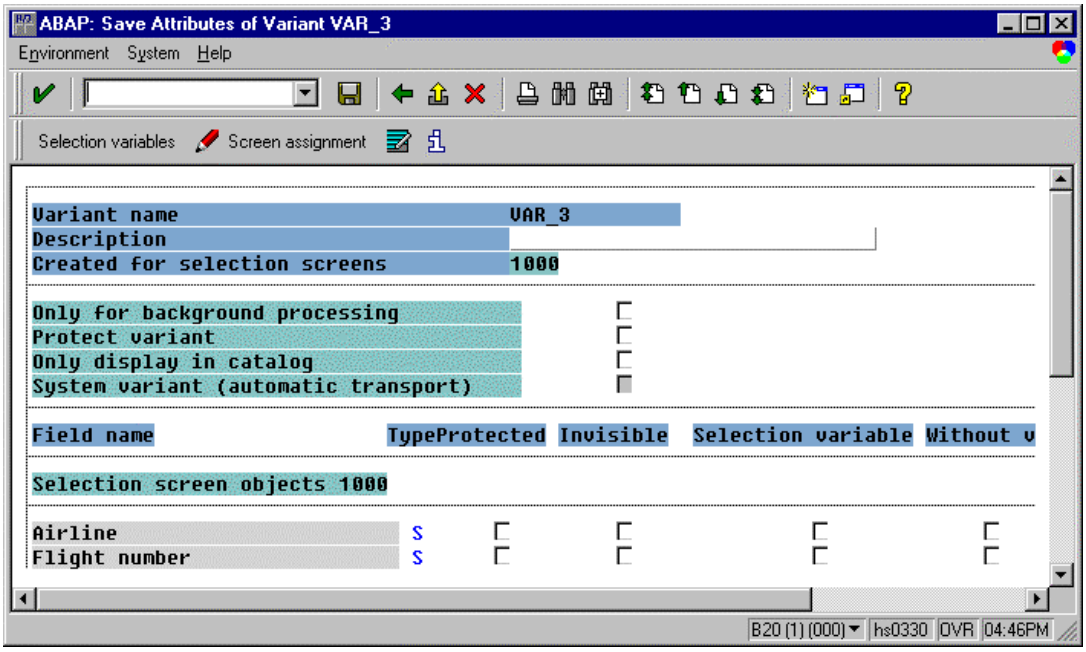
Note that when you create a new variant, you must enter both values **and** attributes.

Names of variants: Names can consist of up to 14 alphanumeric characters. The "% " character is not allowed. If you want the variant to be transported automatically with its program, you must create a **system variant**. The name of a system variant starts "CUS&" for customers, and "SAP&" for SAP system variants. You can only use the "&" character within this prefix in the name of a system variant. It may not occur in any other context. System variants are administered by the Workbench Organizer. Although you can create and access variants from any client, they are always stored in client "000".

Variant Attributes

To maintain the attributes of a variant, follow the same procedure as described in creating a variant [Page 168].

Example:



You can enter the following attributes on the ABAP: Save Attributes of Variant … screen:

- ***Description***

  Enter a short, meaningful description of the variant. This can be up to 30 characters long.

- ***Only for background processing***

  Select this field if you want the variant to be available for background processing but not in dialog mode.

- ***Protect variant***

  Select this field if you want to prevent your variant being changed by other users.

- ***Only display in catalog***

  Select this field if you only want the variant name to be displayed in the variant catalog (and not when the user calls the F4 value help).

- ***System variant***

  This field cannot accept input. It is set automatically when a system variant (beginning with CUS& or SAP&) is created.

  You can also assign the following further attributes to the selections in a variant:

- ***Type***

  The system indicates here whether a field is a parameter (P) or a selection option (S).

- ***Protected***

  Select this column for each selection that you want to write-protect on the selection screen. These fields are visible on the selection screen when the user starts a program with the variant, but do not accept user input.

- ***Invisible***

  If you select this column, the system hides the corresponding field on the selection screen. This allows you to change the appearance of the selection screen.

- ***Selection variable***

  If you select this column, you can set the value of the corresponding selection dynamically at runtime. The different ways of doing this are explained in the section Variable Values in Variants [Page 1 6].

- ***Without values***

  If you select this field, the contents of the corresponding field are not saved with the variant.

  This is useful if you do not want to overwrite the contents of this field on the selection screen.

  For example, suppose you create a report 'SAPTEST', with the parameter 'TEST', for which you create the variant 'TESTVARIANT'. In the variant, you set the 'Without values' flag for the parameter. Then, you run time program and enter the value 'ABCD' in the TEST field. If you now retrieve the 'TESTVARIANT' variant, the TEST field retains the value ABC instead of being overwritten by SPACE.

- ***SPA/GPA***

  This attribute only appears if you created the corresponding selection criterion using 'MEMORY ID xxx'. You can switch the SPA/GPA handling on and off in the variant. This means that fields filled using SPA/GPA appear with their initial values after you have loaded a variant in which those fields have an initial value.

Other Functions

Save

When you have entered all of the parameters, save your settings. When you create a new variant, you must enter both values and attributes. You can only save your variant on the attribute screen. However, if you only want to change values or attributes of an existing variant, you can save on the corresponding screen.

Changing the Screen Assignment

The attribute screen lists all of the selection screens for which the variant is defined. While it is possible for the same selection criterion to appear on more than one screen, the selection criterion itself may be a global field in the program. For this reason, it can only be set once, when the selection criterion occurs for the first time.

Variant Attributes

If you want to change the screen assignment later on, choose *Change screen assignment*.

Defining Selection Variables

See Variable Values in Variants [Page 1 6]

Changing Variants

Procedure

To change a variant:

1. Open the variant as described in Creating a variant [Page 168].
2. On the initial screen [Page 165], choose *Values* or *Attributes*.
3. Choose *Change*.
   Depending on your choice in step 2, the program selection screen or the attributes [Page 1 0] screen appears.
4. When you have finished changing the values or attributes, save your changes on the same screen.

Deleting Variants

Procedure

1. Open the appropriate variant as described in Creating a Variant [Page 168].
2. On the initial screen [Page 165, choose *Variants* → *Delete*.
   The *ABAP: Delete Variants* dialog box appears.
3. Choose whether you want to delete the variant in all clients, or only in the current client.
4. Confirm your choice.

Result

The system displays an appropriate message in the status bar.

You can delete several variants at once from the variant catalog. Choose *Delete variants*, select the relevant variants in the *selection* dialog box, and choose *Delete*.

Printing Variants

Procedure

To print a variant, enter its name on the initial screen of the variant maintenance tool, select either *Attributes* or *Values* in the *sub-objects* group box and choose *Print*. Note that you cannot print the values if you are working in change mode.

The *Print Screen List* screen appears.

If the default print parameters are incorrect, enter the correct values (consult your system administrator if you are not sure). Ensure that the *Output immediately* option is set.

Choose *Print*.

Variable Values in Variants

To avoid having to create a new variant each time you use different values, you can use variables in variants.

There are three ways to do this:

- Variable date calculations (see also Creating variables for date calculation [Page 1])

  You can use this option when you need to use a date in a variant, for example, today's date, or the last day of the previous month.

- User-specific values (see also User-specific selection variables [Page 1 9])

  You can use this option to enter user-specific values in a selection field.

- Values defined in table TVARV (see also Fixed values in table TVARV [Page 183])

  To fill selection fields for a specific task using a variant you can save fixed values in table TVARV. To avoid having to create new variants for each minor change in the selection values, you can assign a value in table TVARV to a selection and then just change this value. This is particularly important if the corresponding value on the selection screen is write-protected.

Using Variables for Date Calculations

Prerequisites

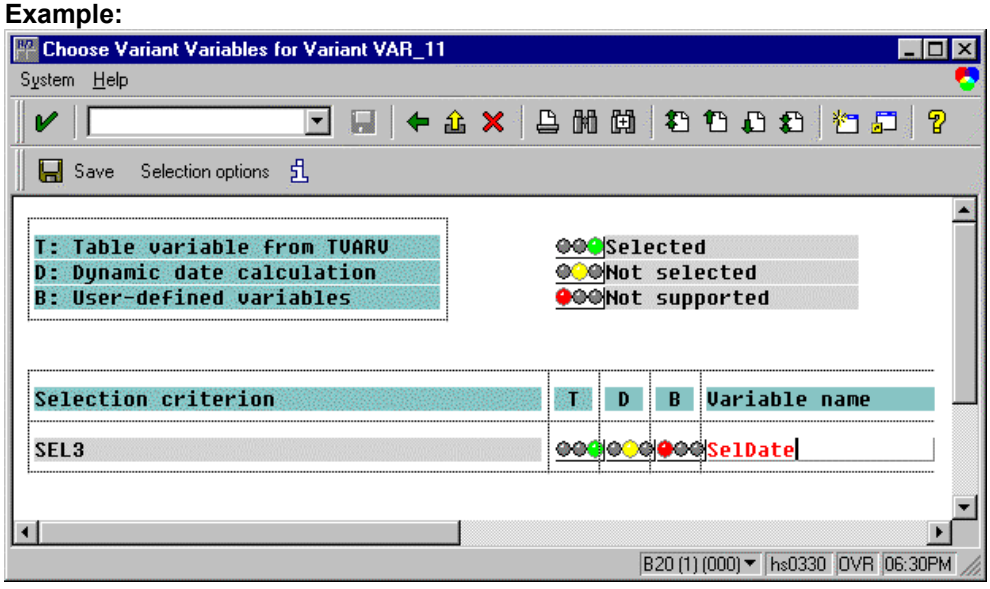
To use a selection variable for date calculation, you must have defined a date field as a parameter or select-option in your program.

```
PARAMETERS DATE LIKE SY-DATUM.
```

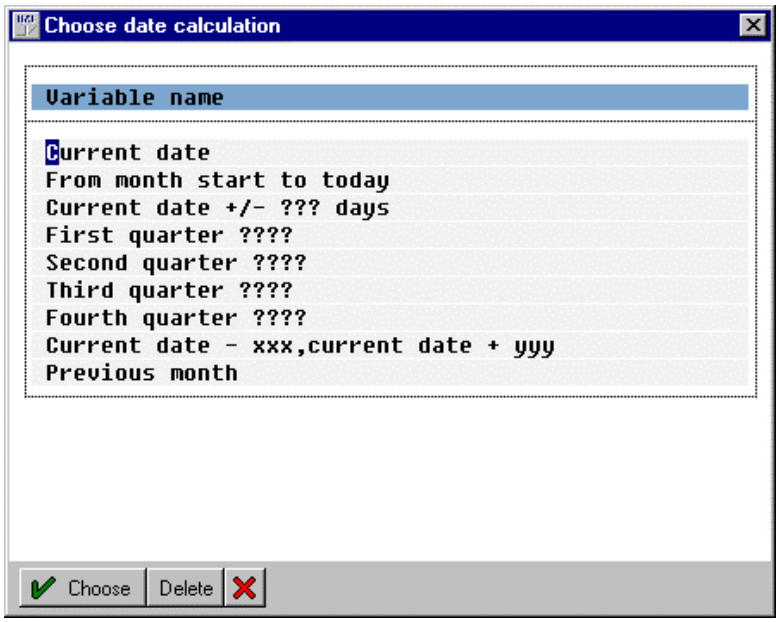
Procedure

To assign a date calculation variable to an existing variant:

1. Enter the name of the variant on the initial screen of the variant maintenance tool.
2. Select the *Attributes* option.
3. Choose *Change*.
   The attributes [Page 1 0] screen appears.
4. Select the *Selection variable* column for the appropriate attribute.
5. Choose *Selection variables*.
   The selection variables screen appears. You can now assign a variable to the date field.
   Example:



6. Position the cursor on the line in which the stoplight in the "D" column is yellow.
7. Single-click the stoplight to turn it to green.
8. Scroll to the right in the display and choose the F4 value help to display proposals for the date calculation:



9. Select an entry and choose *Choose*.

   Add further parameters if necessary. To subtract days, the minus sign must **follow** the number (for example, 10-).

   Note that you cannot change the proposals in the list or add new proposals.

10. Save your entries.
    The attribute screen reappears.
11. Save the attributes.

Result

You have assigned a variable to a variant for date calculation. Note that after choosing the selection variable, you must save the attributes again on the attributes screen.

Use the selection options pushbutton on the selection variable screen to enter further options for the date variable.

User-specific Selection Variables

Use

You can use user-specific selection variables to make input values in a variant user-dependent. The values are saved in tables for all authorized users, and retrieved when the user starts a program using the corresponding variable.

This means that you can create variants in which users do not constantly have to enter static values such as their personnel number or company code whenever they run the program. Instead, they only have to fill out the fields whose values change each time they run the program. As a result, several users can benefit from a single variant.

Prerequisites

To place user-specific values in a field, the master record of the relevant user must contain the corresponding user parameter with a parameter ID <pid>.

In an executable program (report), you must create a parameter or selection option using the addition...MEMORY ID <pid> with the correct parameter ID (see Using default values from SAP memory [Ext.]).

```
DATA: CCODE(6).
...
SELECT-OPTIONS: CC FOR CCODE MEMORY ID BUK.
...
```

Features

Creating User Values

See Creating user-specific variables [Page 180].

Changing User Values

You can change the values of an existing user variable in two ways:

- Using the function module VARI_USER_VARS_*. See Changing values from the program [Page 182])
- From the selection screen (changed by the user him- or herself). See Changing values interactively [Page 181]).

Creating User-specific Variables

Prerequisites

See the prerequisites under User-specific selection variables [Page 1 9].

Procedure

To create a user-specific variable for an existing variant:

1. Enter the name of the variant on the initial screen of the variant maintenance tool.
2. Select *Attributes.*
3. Choose *Change.*
   The attributes [Page 1 0] screen appears.
4. On the attributes screen, select the *selection variable* option for the required fields.
5. Choose *Selection variables*.
   The selection variables screen appears.
6. Position the cursor on the line in which the stoplight in the "B" column is yellow.
7. Single-click the stoplight to turn it to green.
8. Scroll to the right in the display, and choose the F4 value help to choose proposed values from the user master record.
9. Select an entry and choose *Choose*
10. Save your entries.
    The attributes screen reappears.
11. Save the attributes.

Changing Values Interactively

Users can change the values of their user-specific variables on the selection screen.

Prerequisites

The selection screen of the program must be displayed.

Procedure

1. Choose *Goto* → *User variables*. A dialog box appears, containing the user-specific selection criteria and parameters. From here, you can display or change the values.
2. If you choose Change, a further dialog box appears, in which you can decide whether to accept the values proposed in the variables or use the entries on the selection screen.
3. In either case, another dialog box appears, in which you can enter and save the required values.

Note that these changes affect all variants that use the same user-specific variables.

Changing Values from a Program

There is a range of function modules that allow you to work with user variables in your program:

Function module	Function
VARI_USER_VARS_GET	Reads existing variable values
VARI_USER_VARS_SET	Changes existing variable values
VARI_USER_VARS_COPY	Copies variable values
VARI_USER_VARS_DELETE	Deletes variable values
VARI_USER_VARS_RENAME	Renames variable values
VARI_USER_VARS_DIALOG	Dialog for entering variable values

To include these function modules in your program, choose *Edit* → *Insert statement* → *CALL FUNCTION* from the ABAP Editor.

Fixed Values from Table TVARV

Use

Using fixed values from table TVARV is particularly useful in background processing. You do not need to create a new variant or keep changing an existing variant each time a value changes. Instead, you can change the relevant values in table TVARV.

Note that changes to a value in table TVARV are visible in all of the variants that use the variable.

Features

- Creating new table variables from table TVARV.
  See Creating table variables from table TVARV [Page 184].
- Changing an existing entry in table TVARV.
  See Changing entries [Page 186].

Creating Table Variables in Table TVARV

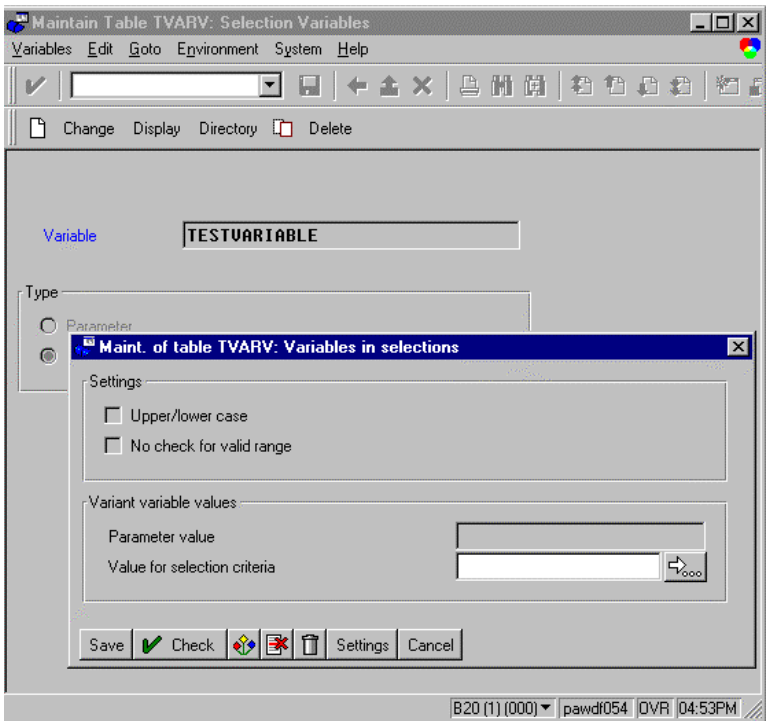
There are two ways of assigning TVARV entries to a selection variant. You can either select existing entries, or **create new entrie**s in the table.

Procedure

1. On the variant maintenance initial screen, enter the required variant.
2. Select the *Attributes* option.
3. Choose *Change.*
   The attributes [Page 1 0] screen appears.
4. On the attributes screen, select the *Selection variable* column.
5. Choose *Selection variables*.
   The selection variables screen appears. You can now assign an entry from table TVARV to the selection variant that you have chosen.
6. Choose the possible entries help next to the Variable name field. A list appears, from which you can select an entry. If you do this, jump to step 11 of the procedure.

   To display the values of a variable in the list, select the variable and chose *Values*. The system displays the corresponding values from table TVARV.
7. To create a new variable, enter a name and choose *Create*.
   The *Maintain Table TVARV* screen appears.
8. Enter the name of the variable and choose *Create*.
   A dialog box appears, in which you can enter the values for the variant.



9. Enter the *parameter value* or the *value for selection criteria*.
10. Choose *Save.*
    You return to the *Maintain Table TVARV* screen.
11. Go back to the selection variable screen.
    The system automatically enters the name of the variable in the right field.
12. Choose *Save.*
    You return to the attributes screen.
13. Save the attributes.

Result

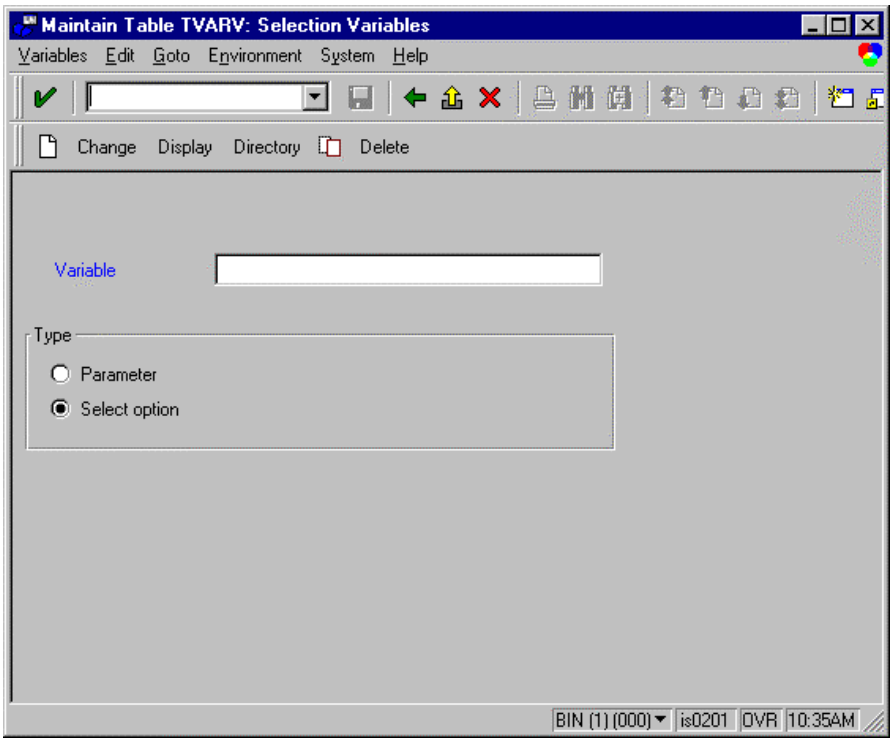
In this procedure, you have created a new variable in the table TVARV, maintained its values, and assigned the variable to a selection variant. The new variable appears when you next call the F4 help.

Changing Entries in Table TVARV

Note that any changes you make to values in TVARV affect all of the variants that use the associated variables.

Starting the Table Maintenance

To change fixed values in table TVARV, start Transaction SM31. Enter “TVARV” as the table name. The table maintenance screen appears:



Functions

Displaying Variable Values

To display the value of a variable:

- If you know the name of the variable, you can enter its name and type directly and choose *Display*.
- To display a list of variables, choose *Directory*.

  A selection screen appears, on which you can enter criteria for the set of variables from which you want to choose. If you do not enter any selection criteria, the system lists all variables in the table.

  Choose *Execute* to display the list of variables.

  To display a variable, double-click its entry in the list.

If you chose a parameter, a dialog box, containing its current value, appears.

If you chose a select-option, its values are displayed on a new screen.

Changing Variable Values

To change the value of a variable:

1. If you know the name and type of the variable, you can call it directly. Enter the name and type and choose *Change*.

   To display a list of variables, choose *Directory*.

   A selection screen appears, on which you can enter criteria for the set of variables from which you want to choose. If you do not enter any selection criteria, the system lists all variables in the table.

   To choose a variable, double-click its entry in the list.

   If you chose a parameter, a dialog box, containing its current value, appears.

   If you chose a select-option, its values are displayed on a new screen.

2. Change the value as required.
3. Save the new value.

   The new value has now been added to table TVARV, and will be placed in the relevant field at runtime.

Adding Variable Values

To add a new value directly to table TVARV, enter its name and type and choose *Create*.

If you set the select option radio button, a screen containing empty input fields appears, on which you can enter lower and upper values, operators, and an inclusive or exclusive flag.

If you set the parameter option, a dialog box appears, in which you can enter the appropriate value.

Save the new variable.

Remember that you must enter the variable name in the variant(s) in which you want to use it.

Copying Variable Values

To copy a variable, enter its name and type and choose *Copy*.

A dialog box appears. Enter the name of the new variable and choose *Copy*.

You can now change the new variable. Remember to enter the variable name in the variants in which you want to use it.

You can also copy variables from the directory display.

Deleting Variables

To delete a variable, enter its name and type and choose *Delete*.

A confirmation prompt appears, in which you can either confirm or cancel the action.

You can also delete variables from the directory display.

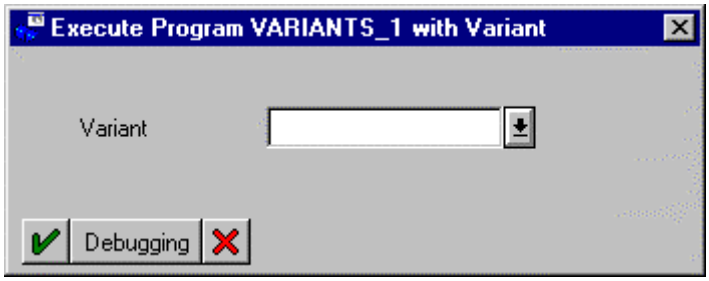
Executing a Program with a Variant

Requirements

The program that you want to execute may be any type except S, and must have one or more selection screens. You must have defined variants for the program.

Procedure

1. On the initial screen of the ABAP Editor, enter the name of the program that you want to run.
2. Choose *Execute with Variant*.
   A dialog box appears, in which you can enter a variant.



3. To display a list of all variants for the program, use the possible values help.
4. Choose a variant.
5. Choose Choose to confirm your choice.
   The selection screen of the program appears. The fields for which values exist in the variant already contain values.
6. Choose *Execute* to run the program.

Maintaining Messages

Messages allow you to communicate with users from your programs. They are mainly used when the user has made an invalid entry on a screen.
To send messages from a program, you must link it to a **message class**. Each message class has an ID, and usually contains a whole set of message. Each message has a single line of text, and may contain placeholders for variables.
All messages are stored in table **T100**. You create and edit them using Transaction **SE91**. Once you have created a message, you can use it in the MESSAGE statement in a program.

For further information about messages, see the messages [Ext.] section of the ABAP Programming Guide.

Starting the Message Maintenance Transaction

- Using forward navigation from the **ABAP Editor**.
- You can display the messages for your program from the ABAP Editor by choosing *Goto → Messages*. *The Maintain Messages* screen appears. By default, the system display the message class linked to the current program.
- You can also enter Transaction **SE91**.

If you choose *Goto → Messages* from the ABAP Editor and your program does not have a defined message class, the system assumes you want to browse an existing class and prompts you for a message class ID.

See also

Creating Message Classes [Page 191]

Adding Messages [Page 192]

Creating a Message Long Text [Page 193]

Assigning an IMG Activity to a Message [Page 194]

Creating Message Classes

Procedure

To create a new message class from the ABAP Editor:

1. Enter a message ID in the introductory statement to the program (like REPORT), or directly in a MESSAGE ID <id> statement. The name may be up to 20 characters long. Example:

   REPORT <Name> MESSAGE-ID <messageclass>.

   Messages are visible systemwide. Your message ID (name of the message class) may therefore not already exist in the system.

2. Double-click the message ID.

   If you specified a message ID that already exists, the system opens the Maintain Message dialog box. If this happens, you can simply enter another ID. If you entered a message ID that does not yet exist, a dialog box appears, in which you are asked whether you want to create a new message class.

3. Choose *Yes*.

   The *Maintain Message Class* screen appears.

4. Enter a short text for the message class.

5. Choose *Save*.

Result

If you choose *Messages*, you can add new messages to your message class. If you double-click the message ID in your program, you can return to the *Maintain Message Class* screen at any time.

See also Adding Messages [Page 192].

Adding Messages

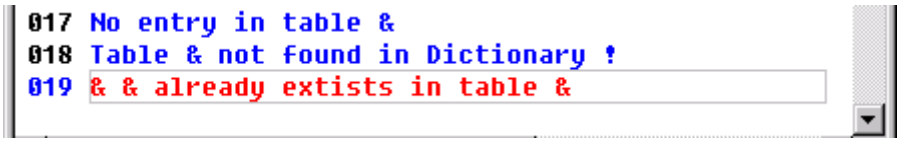
Prerequisites

You must already have specified a valid message class in your ABAP program.

Procedure

To add new messages to a message class from the ABAP Editor:

1. Choose *Goto* → *Messages*.
   A list of all messages in the relevant message class appears.
2. Select the next free message number.
3. Choose *Individual maint.*.
   You can now enter a text in the corresponding line.
4. Enter the message text.



5. If the message text is self-explanatory, set the corresponding flag next to the text field.
   Remember that you cannot maintain long texts for self-explanatory message texts. See also Creating a Message Long Text [Page 193].
6. Choose *Save*.

You can also create new messages using forward navigation. In the MESSAGE statement, enter a new message number and then double-click it. You should preferably enter the next free number in the message class. However, if you do not know it, you can enter any number and subsequently correct it.

Result

You can now use the message that you have defined in the MESSAGE statement in your program.

Creating a Message Long Text

Use

Create a long text whenever the message text itself is not fully self-explanatory.

Prerequisites

You must already have created the message for which you want to create the long text, and **not** flagged it as *self-explanatory*.

Procedure

To enter a long text for a message in the message maintenance transaction (**SE91**):

1. Position the cursor on the relevant message.
2. Choose *Individual maint*.
   The message text appears highlighted.
3. Choose *Long text.*
   The SAPscript editor appears.
4. Enter your long text.
5. Check the text.
6. Save the text.

Result

When you send a message that has a long text, the message is displayed with a yellow question mark symbol.
Depending on the SAPgui settings, the message is displayed:

- In the status bar (at the end of the message area),
- In the message dialog box (on the *Help* pushbutton).

If you then click the message line or the *Help* pushbutton, the long text is displayed.

Assigning IMG Activities to a Message

Use

Use this function whenever you want to branch from a message to one or more relevant activities in the Implementation Guide.

Prerequisites

- You must already have created a short text for the message.
- The IMG activities that you want to specify must exist in the Implementation Guide.

Procedure

To assign an IMG activity from Transaction **SE91**:

1. Position the cursor on the appropriate message in the list.
2. Choose *Individual maint.*
   The corresponding short text is highlighted.
3. Choose *Goto* → *Additional information.*
   The *Maintain Additional Assignment Information* dialog box appears.
4. Choose an activity, using the possible entries help if required.
5. Save the assignment.
   A confirmation message appears in the status bar.

Result

You have now assigned one or more IMG activities to a message. When the message is sent from a program, the user can branch directly to the Implementation Guide and execute the corresponding activity or activities.
To do this, click the status bar when the message is displayed. The *Help* dialog box appears, in which you should then choose *Maintain entries* (pencil icon). The *Choose Customizing Project* dialog box appears, from which you can choose the project in which you want to execute the activities.

The Splitscreen Editor

The splitscreen editor allows you to display the source code of two programs side by side. The two programs do not necessarily have to be in the same system. The editor contains a restricted set of normal ABAP Editor functions, complemented by additional special splitscreen functions.

Contents

If you intend to use the splitscreen editor in conjunction with the Modification Assistant, refer to the Aligning Program Sections [Ext.] documentation.

Overview

Use

The new splitscreen editor replaces the old transaction **SE39**. You can use it in the Modification Assistant and Editor to compare database and clipboard contents.

You can still use the old splitscreen editor. It is available under transaction code TSE39.

Prerequisites

If you intend to compare programs between systems, the remote system must be entered in the RFC destinations table RFCDES.

Features

- Comparison of source code both within a system and between systems.
- Synchronization is now statement-based instead of line-based.
- Each half of the screen is a version of the ABAP Editor with a restricted range of functions.
- Extra splitscreen editor functions, such as comparison, text positioning, and copying blocks to the other side.
- You can configure the editor using the Settings function (editor view and comparison algorithms)
- You can switch the window sizes between ‘Small’ and ‘Large’.

Navigation

You can use forward navigation from the splitscreen editor. Objects to which you navigate are always displayed in the fullscreen.

Starting the Splitscreen Editor

There are various ways of starting the splitscreen editor:

- From any screen by entering Transaction **SE39**.
- From the Modification Assistant
- From the ABAP Editor, if the clipboard is not empty.

Special Features

Using Transaction SE39

You can display or change any source code. You can display, change, compare, and save both code extracts in the splitscreen editor. The extracts may either both be in the same system, or one of them may be in a remote system. You cannot change or save a source code extract that you are displaying remotely. Equally, you cannot navigate in the remote system.

From the Modification Assistant

The splitscreen editor has an **editing window** on the **left**, and a **display window** on the **right**. The old SAP source code appears in the display window. Modified lines of code are flagged in the editor. You cannot switch the right-hand window to change mode.

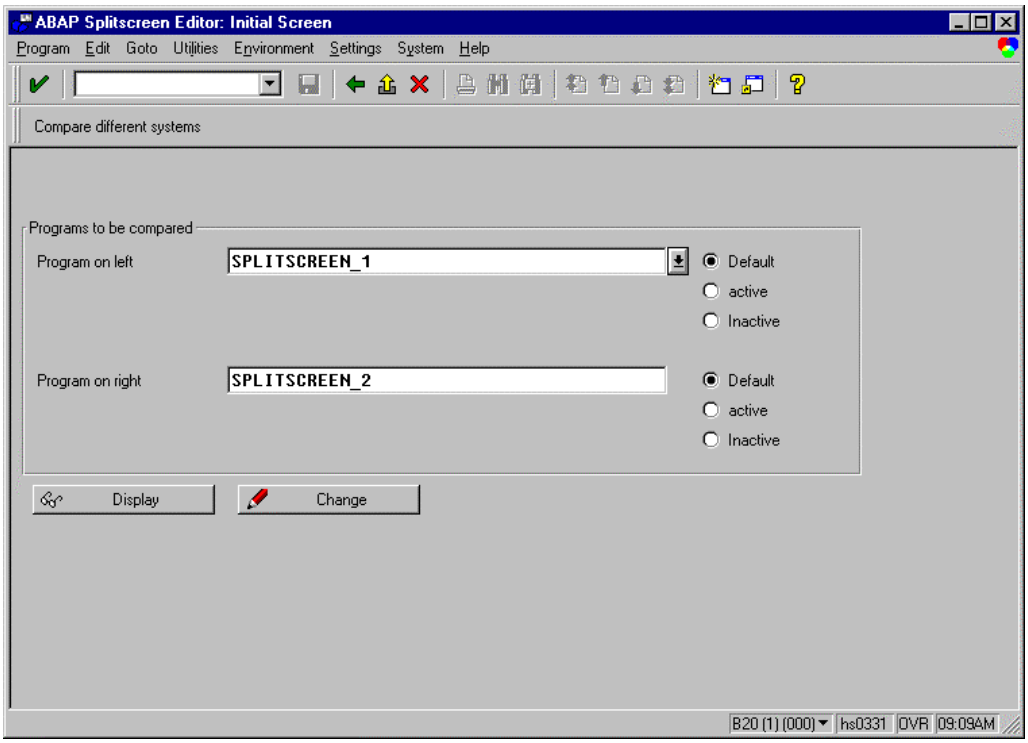
The new SAP source code appears in modification mode in the editing window. You cannot change this code, but you can use the functions *Delete*, *Insert*, and *Replace*. The system compares the two versions and marks the differences on both sides. There is also a function that allows you to select a single block in the display window. You can then copy this to the corresponding position in the editing window. It is also possible to copy source code into the editing window using the *Select*, *Copy to buffer*, and *Insert* buffer functions.

From the ABAP Editor

If there is data on the clipboard during an ABAP Editor session, you can use the splitscreen editor to compare the two versions. The system starts the editor in compare mode. You can change the database version, but not the clipboard version.

Initial Screen

Comparing Programs in the Current R/3 System



Comparing Programs in Different Systems

To compare a program in the current system with one in another system, choose *Compare different systems* from the initial screen of the splitscreen editor. A field appears on the screen in which you can enter the *RFC destination* of the R/3 System containing the required program.

In the *RFC destination* field you can only enter systems for which an entry exists in table RFCDES.

Special Splitscreen Editor Functions

The splitscreen editor suppores the special functions listed below. Some of these are not active in all of the contexts in which you can use the editor. For further information, refer to Starting the splitscreen editor [Page 197].

Compare

Compares the two sets of source code. The comparison is no longer line-based (as in previous versions of the splitscreen editor). Instead it is statement-based. You can suppress comment lines and indentations using the *Settings* function.

The results of the comparison are displayed using highlighting. Where lines have been inserted, the system inserts the same number of blank lines at the corresponding position. In compare mode, the system scrolls both windows when you use the arrow keys. To scroll asynchronously, use the scrollbars and page up / page down keys.

Compare Mode Off

Switching off compare mode removes the blank lines inserted in the source code and the special formatting used to highlight the difference. It also switches off the synchronous scrolling function.

Next Similarity

The system positions both windows at the next point that is identical on both sides.

Presvious Similarity

The system positions both windows at the previous point that is identical on both sides.

Next Difference

The system positions both windows at the next difference.

Previous Difference

The system positions both windows at the previous difference.

The starting point for the above positioning functions is always the current page. You can therefore skip several differences simply by scrolling.

Copy to Buffer (only with Modification Assistant)

Copies the block of code at the current cursor position that was marked by the compare function into the buffer. The comment lines inserted by the Modification Assistant are removed.

Find Marked Block (not yet implemented)

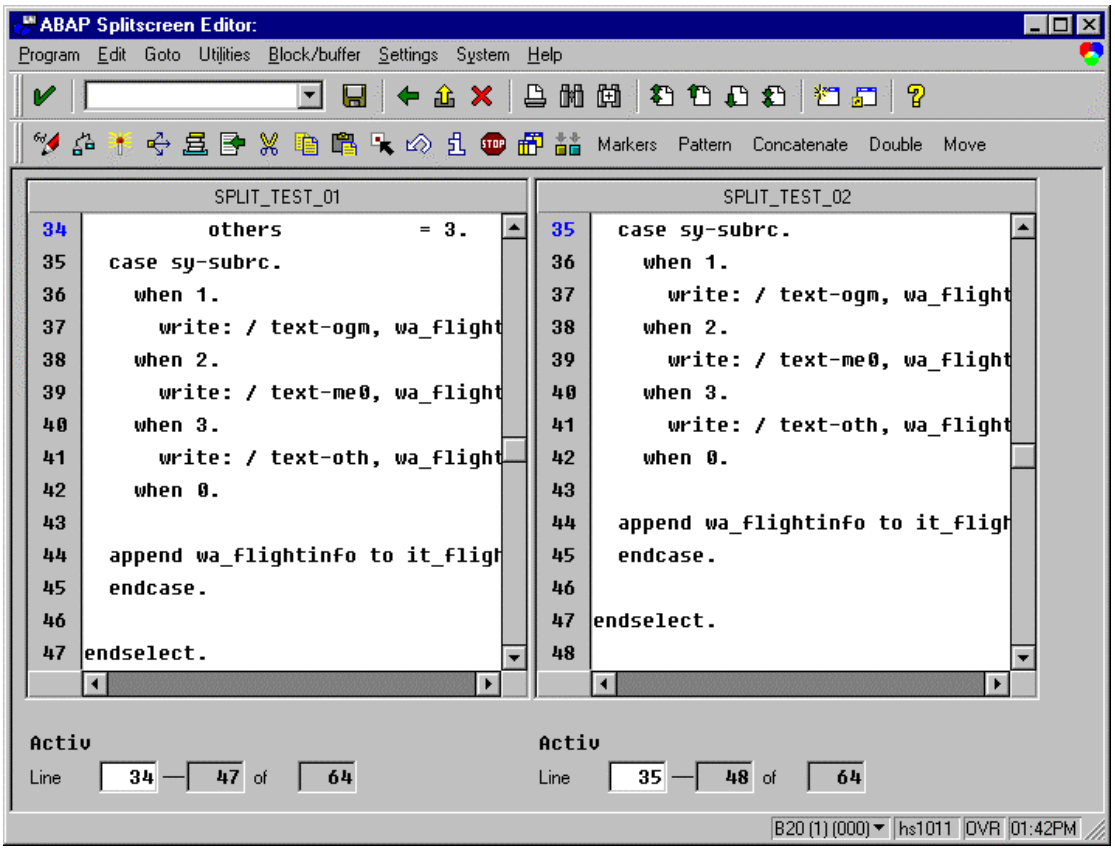
It will be possible to find a block, marked in any way, in the opposite side of the editor.

Set Window Size (narrow <-> wide)

 Use this button (or choose Settings → screen narrow <-> wide) to toggle between the two scren sized. The default size is wide if you start the splitscreen editor using Transaction SE39, and narrow if you start it from the ABAP Editor.

Example:

Special Splitscreen Editor Functions



Editor Functions

Some of the standard ABAP Editor functions behave differently when used in the splitscreen editor.

Display/Change

This function reacts to the cursor position. If you call the splitscreen editor from the Modification Assistant, only the new SAP source code may be switched to change mode.
Code read from a remote system may also not be switched to change mode.

Other Program

If you are reading a program from a remote system or have called the splitscreen editor from the Modification Assistant, you cannot use this function.

Save

The system saves the half of the screen that is currently active.

If you were working in compare mode, the system removes the blank lines inserted by the comparison before saving.

Sources read from a remote system cannot be saved.

Display Active/Inactive Source

You can display both versions of an object (wherever both an active and inactive version exist).

This function is not available when you call the splitscreen editor from the Modification Assistant.

Execute

You cannot execute a source from a remote system. Furthermore, you cannot execute programs when you call the splitscreen editor from the Modification Assistant.

Generate Version

This function is not available when you call the splitscreen editor from the Modification Assistant.

Select, Copy to Buffer, Insert Buffer

These functions allow you to copy blocks of code from one window to the other.

Class Builder

The Class Builder is a tool within the ABAP Workbench that allows you to create, define, and test global ABAP classes and interfaces.

Contents:

Introduction to the Class Builder

Purpose

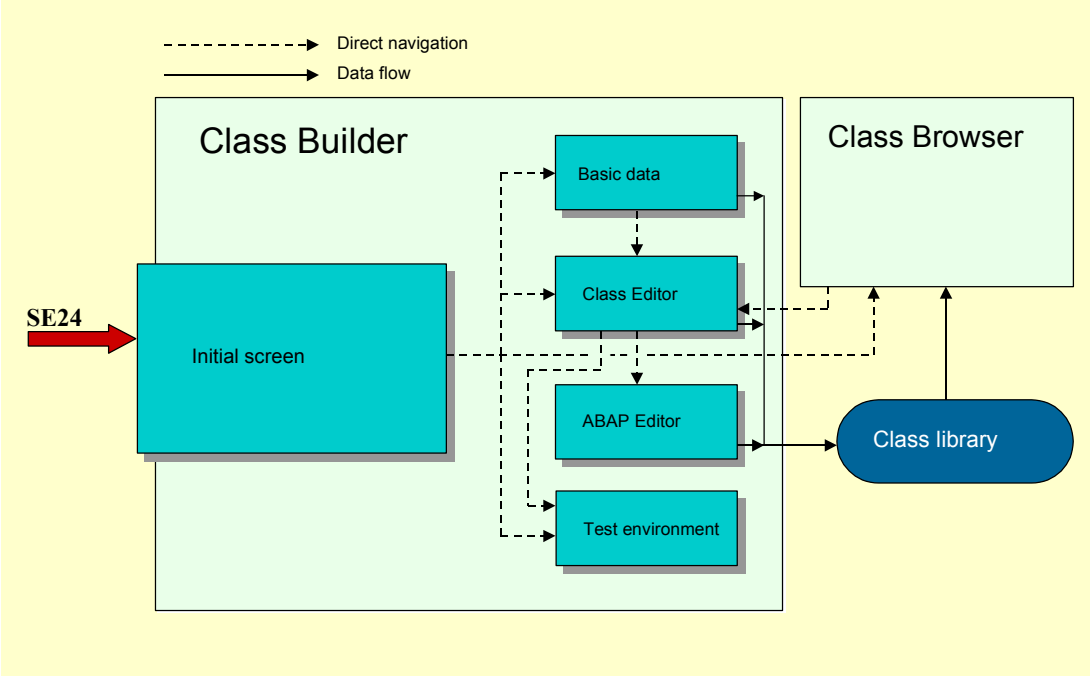
The Class Builder allows you to create and maintain global ABAP classes and interfaces. Both of these object types, like global data types, are defined in the R/3 Repository. Together, they form a central class library and are visible throughout the system. You can display existing classes and interfaces in the class library using the Class Browser [Page 213].

You can define local classes as well as global classes. Local classes are defined in programs, function groups, or in a class pool of auxiliary classes within a global class. They are only visible in the module in which they are defined.

Integration

The Class Builder is a fully-integrated tool in the ABAP Workbench that allows you to create, display, and maintain global object types from the class library. The diagram below illustrates the architecture of the Class Builder and the relationships between its components (including the Class Browser).

To reach the initial screen [Page 216] of the Class Builder, choose *Development* → *Class Builder* from the initial screen of the ABAP Workbench or enter transaction code SE24. From here, you can either display the contents of the R/3 class library or edit a class using the Class Editor. Once you have defined an object type, you can implement its methods. From the initial screen or the Class Editor, you can also access the Class Builder's test environment.



Features

Use the Class Builder to:

- Display an overview (in the Class Browser) of global object types and their relationships.
- Maintain existing global classes or interfaces.
- Create new classes and interfaces.
- Implement inheritance between global classes
- Create compound interfaces
- Create and specify the attributes, methods, and events of global classes and interfaces.
- Define internal types in classes.
- Implement methods.
- Redefine methods
- Maintain local auxiliary classes.
- Test classes or interfaces in a simulated runtime environment.

Constraints

You cannot define object types on the basis of graphical object modeling.

Naming Conventions in ABAP Objects

Global classes and interfaces that you create in the Class Builder are stored in the class library and administered by the R/3 Repository: they therefore have the same namespace as all other Repository objects (database tables, structures, data elements, and so on).
It is therefore necessary to have naming conventions for object types and their components and to use them uniformly within program development.

The following naming convention has been conceived for use within the **SAP namespace**:
If you do not observe the naming conventions for object types (classes and interfaces), conflicts will occur when the system creates persistent classes, since it will be unable to generate the necessary co-classes.

Namespace for Components

A single namespace within a class is shared by:

- All components of the class itself (attributes, methods, events, constructors, interfaces, internal data types in the class, and aliases)
- All public and protected components of the superclasses of the class.

Method implementation has a local namespace. The names of the local variables can obscure those of class components.

Naming Convention

The naming convention has been kept as general as possible to avoid adversely influencing the naming of objects.

General Remarks

When you choose names for development objects, you should:

- Use **English** names
- Use **glossary terms** when possible

  For example, CL_COMPANY_CODE instead of BUKRS
- In compound names, use the underscore character (_) as a separator. Since names are not case-sensitive, this is the only character that you can use to separate names.

  Example: CL_COMPANY_CODE, CL_GENERAL_LEDGER_ACCOUNT
- Names should describe the **action**, not the implementation of the action.

  Example: PRINT_RECTANGLE, not RECTANGLE_TO_SPOOL

Conventions for Object Types

Naming Conventions in ABAP Objects

Class and interface names in the class library belong to the same namespace as data elements, tables, structures, and types. They are maintained centrally in table **TADIR**.

Class in the class library	**CL_<class name>** The class name should be made up of singular nouns. CL_COMPANY_CODE, CL_GENERAL_LEDGER_ACCOUNT
Interfaces in the class library	**IF_<interface name>** The same naming convention applies to interfaces as to classes. IF_STATUS_MANAGEMANT, IF_CHECKER
Local classes in programs (recommendation)	**LCL_<class name>** The class name should be made up of singular nouns. LCL_TREE_MANAGEMENT
Local interfaces in programs (recommendation)	**LIF_<interface name>** The sane naming convention applies to interfaces as to classes. LIF_PRINTER

Recommended naming conventions are not compulsory. However, if you use prefixes for these class and interface names, you should use those listed above.

Conventions for Components

Method name	**<method name>** Method names should begin with a verb: GET_STATUS, CREATE_ORDER, DETERMINE_PRICE
Events	**<event name>** Event names should have the form <noun>_<participle>: BUTTON_PUSHED, COMPANY_CODE_CHANGED, BUSINESS_PARTNER_PRINTED
Local type definitions within a class (recommendation)	**TY_<type name>** TY_INTERNAL_TYPE, TY_TREE_LIST

Data definitions (variables)	**<variable name>** When you name variables within a class (CLASS-DATA or DATA), avoid using verbs at the beginning of the name (to avoid conflicts with method names). LINE_COUNT, MARK_PRINTED, MARK_CHANGED, STATUS
Data definitions (constants) (recommendation)	**CO_<constant name>** CO_MAX_LINE, CO_DEFAULT_STATUS, CO_DEFAULT_WIDTH, CO_MAX_ROWS

Recommended naming conventions are not compulsory. However, if you use prefixes for these class and interface names, you should use those listed above..

Concrete Method Descriptions

Attribute access	**SET_<attribute name>, GET_<attribute name>** Methods that access attributes of any kind should be prefaced with GET_ or SET_. GET_STATUS, SET_USE_COUNT
Event handler methods	**ON_<event name>** Methods that handle events should begin with ON, followed by the name of the event that they handle. ON_BUTTON_PUSHED, ON_BUSINESS_PARTNER_PRINTED
Methods that perform type conversions	**AS_<new type>** AS_STRING, AS_ISOCODE
Methods that return a Boolean value These methods **may not** return any exceptions. Recommendation: Use SPACE and 'X' to represent false and true respectively.	**IS_<adjective>** IS_OPEN, IS_EMPTY, IS_ACTIVE
Check methods	**CHECK_<objective>** CHECK_AUTHORIZATION, CHECK_PROCESS_DATE

Local Conventions Within Methods

For parameters

The parameters are regarded from the point of view of the method that implements them:

IMPORTING parameters	**IM_<parameter name>**
EXPORTING parameters	**EX_<parameter name>**
CHANGING parameters	**CH_<parameter name>**
RESULT	**RE_<result>**

Using prefixes is NOT compulsory. However, if you do use them, use those listed above.

For exceptions

The following table contains a series of possible exception names, that can also be used generically (for example, NOT_FOUND could also be used as DATE_NOT_FOUND)

EXCEPTION	Meaning
ACTION_NOT_SUPPORTED	The requested action or function code is not supported.
CANCELLED	If a method uses a dialog to find out what has to be dome (for example, a list of choices), and the user chooses "Cancel", you can set this exception.
EXISTING	A new object that you want to create already exists in the database.
FAILED	The method could not be executed because of the current environment. This exception is intended for cases where the method cannot be executed because of variable system circumstances.
..._FAILED	Part of the method could not be completed because of the current environment. (OPEN_FAILED, CLOSE_FAILED, SELECTION_FAILED, AUTHORIZATION_FAILED)
FOREIGN_LOCK	Data is locked by another user.
INCONSISTENT	Object data in the database is inconsistent.
..._INCONSISTENT	The component data for … of an object in the database is inconsistent.
INVALID	The object data entered is incorrect (for example, company code does not exist). Compare NOT_QUALIFIED.
..._INVALID	The component data entered for an object is incorrect. Compare NOT_QUALIFIED.

EXCEPTION	Meaning
INTERNAL_ERROR	Last resort. Only use this exception if you cannot be more precise about the nature of the error.
NOT_AUTHORIZED	The user does not have the required authorization.
NOT_CUSTOMIZED	The object requested is not correctly customized.
..._NOT_CUSTOMIZED	The component … of the requested object is not correctly customized.
NOT_FOUND	Unable to find the requested object.
..._NOT_FOUND	Unable to find component … of the requested object.
NOT_QUALIFIED	The combination of input parameters is insufficient to run the method. Compare INVALID.
..._NOT_QUALIFIED	One parameter of the method is not qualified.
NUMBER_ERROR	Error assigning a number.
SYSTEM_ERROR	This exception is set if the Basis system returns an unexpected error message.

Overview of Existing Object Types

Use

Use this function to:

- Display the definitions of classes and interfaces stored in the function library.
- Display further information about object type components such as methods and parameter definitions.

Activities

To display an overview of existing object types in the class library, use the Class Browser [Page 213].

You can also use the Repository Information System. From the initial screen of the ABAP Workbench, choose *Overview* → *Information system*. If you then choose ABAP Objects, you can open all or part of the class library.

Class Browser

Use

Use the Class Browser to display global ABAP classes and interfaces or business object types from the class library.

The Class Browser enables you to:

- Display an overview of existing classes, interfaces, and business object types.
- Display the relationships between object types.
- Switch from the overview to maintain an individual object type.

Integration

The Class Browser is an integrated part of the Class Builder [Page 202]. You can start it either from the Class Builder, or using Transaction **CLABAP**.

Features

Display

There is a range of preconfigured views that you can use to display object types. You can also set a selection of filters to meet particular display requirements.

- All classes
  Displays all classes and interfaces in the R/3 class library. The display is based on the R/3 component hierarchy.
- Business objects
  Displays business object types from the R/3 class library.
- *Other settings*
  You can adapt the display further by setting filters. There are three separate selection criteria:

  1. *Object types*
  You can select object types by type, status, and transport attributes.

  2. *Relationships*
  You can select object types based on the relationships between them.

  3. *Other*
  You can use this filter to set whether the object types should be selected according to the component hierarchy or not.

Maintenance

You can switch from the display to maintain an object type by double-clicking it. The system starts the Class Editor of the Class Builder. You can then switch to change mode and modify the object type.

Restrictions

You cannot create new object types from the Class Browser.

Creating Object Types

Features

When you create classes and interfaces, you only specify their basic data. This definition produces a table entry for the object type in the ABAP Dictionary. Once you have defined the basic data, you may go on to work on the object type components in more detail. Once you have entered the basic data, the system automatically opens the Class Editor [Page 224].

Activities

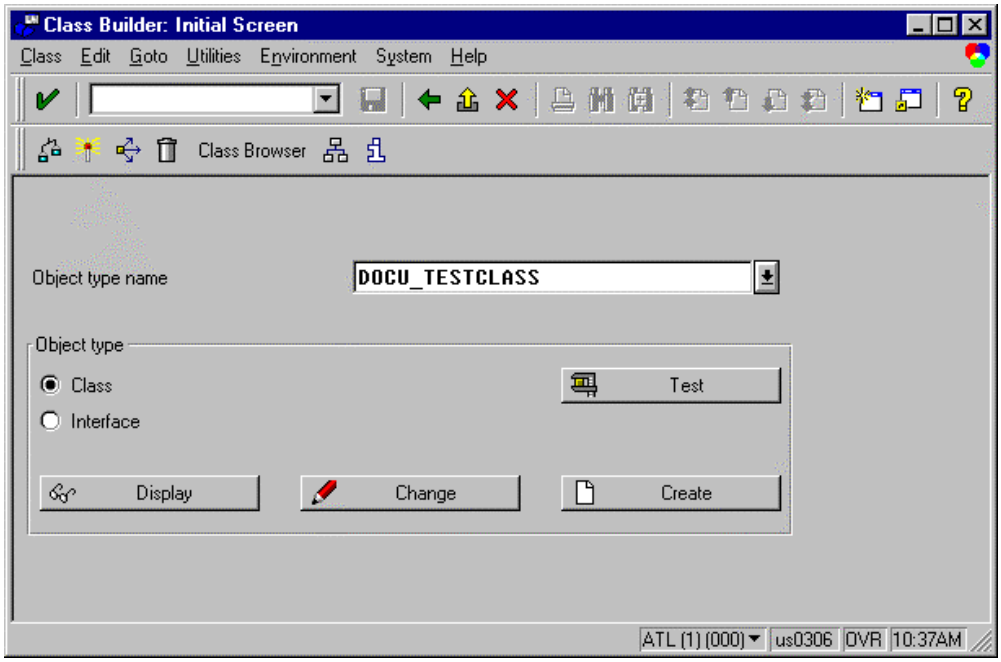
To start the Class Builder, enter Transaction SE24, or choose *Development* → *Class Builder* from the initial screen of the ABAP Workbench.

- To maintain an existing object, enter the name of the object type and choose *Display* or *Change*.
- To create a new class, refer to Creating New Classes [Page 218].
- To create a new interface, refer to Creating New Interfaces [Page 220].

Initial Screen

Procedure

1. Start the ABAP Workbench.
2. Choose *Development* → *Class Builder* (Transaction SE24) to start the Class Builder. The initial screen appears:



3. Enter the name of the object type that you want to display, change, create, or test. The name can be up to 30 characters long.
4. Select the relevant *Object type*:

Class	The definition and implementation of global classes and their components: Attributes, methods, events, and internal data types within the class. You can extend the class definition using interfaces. In this case, the class must implement all of the methods that are declared in the interface.
Interface	The definition of interfaces that describe a point of contact with an object. Interfaces are independent of classes. Like classes, they can contain attributes, methods, and events. However, unlike classes, they do not implement them. You can only use an interface once you have implemented it in a class.

5. Choose the required function: *Display*, *Change*, *Create*, or *Test*.

Other Functions

- You can start the Class Browser [Page 213]. This allows you to display the existing classes and interfaces in the class library.
- The *Check* function allows you to check a class or interface for syntax errors.

Creating New Classes

For general information about ABAP classes, refer to the classes [Ext.] section of the ABAP User's Guide.

Prerequisites

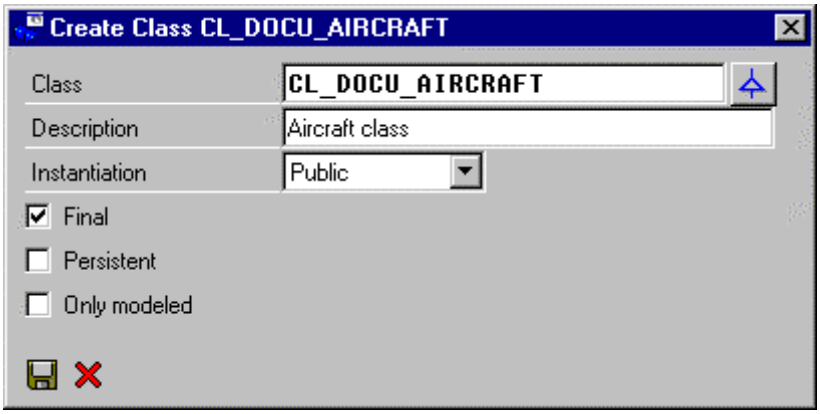
When you name your class, observe the naming conventions for global ABAP classes.

Procedure

To create a new class from the initial screen of the ABAP Workbench:

1. Under *Object type name*, enter the name of the new class, not forgetting to observe the naming conventions.
2. Select object type *Class.*
3. Choose *Create.*

   The *Create Class* dialog box appears with the name of the class:



4. You define the basic data by entering the following information:

   - *Class*

   Name of the new class.

   - *Description*

   A short text describing the new class.

   - *Instantiation*

   The default setting for all classes is *Public*. This means that any user can instantiate the class using CREATE OBJECT. If you specify *Protected*, only the class itself or its subclasses can instantiate the class. If you choose *Private*, only the class can instantiate itself (using one of its own methods).
   If you select *Abstract*, you create a class that cannot be instantiated at all. Abstract

classes serve as templates for subclasses, and can only be accessed using their static attributes or one of their subclasses.
With these options, you can restrict the instantiation conditions further. For example, they enable you to fulfil conditions for administering persistent objects, where you must be able to guarantee that an object is unique.

- *Inheritance*

If you select *Final*, you define a final class. A final class is the end of an inheritance hierarchy, since it may not have any subclasses of its own.
If you make an abstract class final, you will only be able to access its static components.
.
.

- *Persistent*

- *Modeled*

If you select this checkbox, the system will not define the subclass in the class pool. You will not be able to address it at runtime or test it. In future, this option will allow you to design classes from a graphical model without having to implement them.

- *Create inheritance* icon

When you choose this function, the *Inherits from* dialog box appears. Here, you can enter the name of a superclass. The superclass can be any class from the class library that is not defined as final.

5. Choose *Save*.

The *Create Object Directory* Entry dialog box appears.

6. Enter the development class.
7. Choose Save.
   The Class Editor [Page 224] appears. From here, you can define its components and include interfaces in it.

Result

You have now defined a new class and entered its basic data. The system creates a class pool for the new class (as long as you did not define it as *Modeled*).

A class pool contains the definition of a single global class. They are similar to function groups in that you can define local auxiliary classes and local types.

Creating New Interfaces

For general information about ABAP interfaces, refer to the Interfaces [Ext.] section of the ABAP User's Guide.

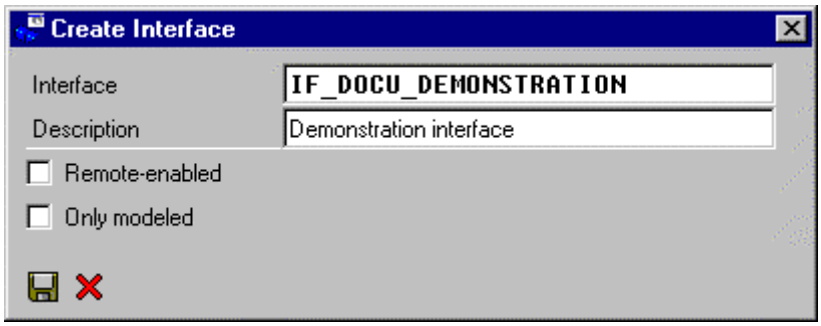
Prerequisites

When you name your interface, remember to observe the naming conventions for global ABAP interfaces.

Procedure

To create a new interface from the initial screen of the ABAP Workbench:

1. Enter the name of the interface, remembering to observe the naming convention
2. Select object type *Interface*.
3. Choose *Create.*
   The *Create Interface* dialog box appears.



4. Enter the following information:

   - Description

   A short descriptive text for the new interface.

   - Modeled

   If you select this option, no interface pool is generated for the interface, and you cannot access it at runtime.

   In future, this option will allow you to design interfaces based on a graphical model.

5. Choose *Save*.
   The *Create Object Directory Entry* dialog box appears.
6. Enter the development class.
7. Choose Save.
   The methods screen of the Class Editor [Page 224] appears. From here, you can define the

components of the interface. Only after you have defined the interface can you include it in class definitions.

Result

You have created a new interface along with its basic data in the ABAP Dictionary. The system generates an interface pool for the interface, as long as you did not create the interface as *Modeled*.

Defining Components

Use

You use this function:

- To define classes or interfaces by assigning components to them.
- To implement the methods of classes.
- To add interfaces to classes and implement their methods in the class.
- To change the existing definition and implementation of classes.
- To define local data types within classes.

Prerequisites

You must already have created a class or interface as described under Maintaining Object Types [Page 213].

Features

Assign Components by Defining:

- Attributes
- Methods
- Events
- Local types in classes

Note that these components (apart from interfaces) are all stored in the **same namespace**. Consequently, they must all have different names.

- Interfaces

Methods

- Define method parameters
- Define method exceptions
- Implement methods

Interfaces

- Assign interfaces to classes
- Implement interface methods in classes

Activities

You define and implement object types in the Class Editor [Page 224].

To open the Class Editor:

- Create an object type. The Class Editor starts automatically.
- Choose *Change* from the initial screen of the Class Builder.
- Double-click an entry in the Class Browser and switch to change mode.

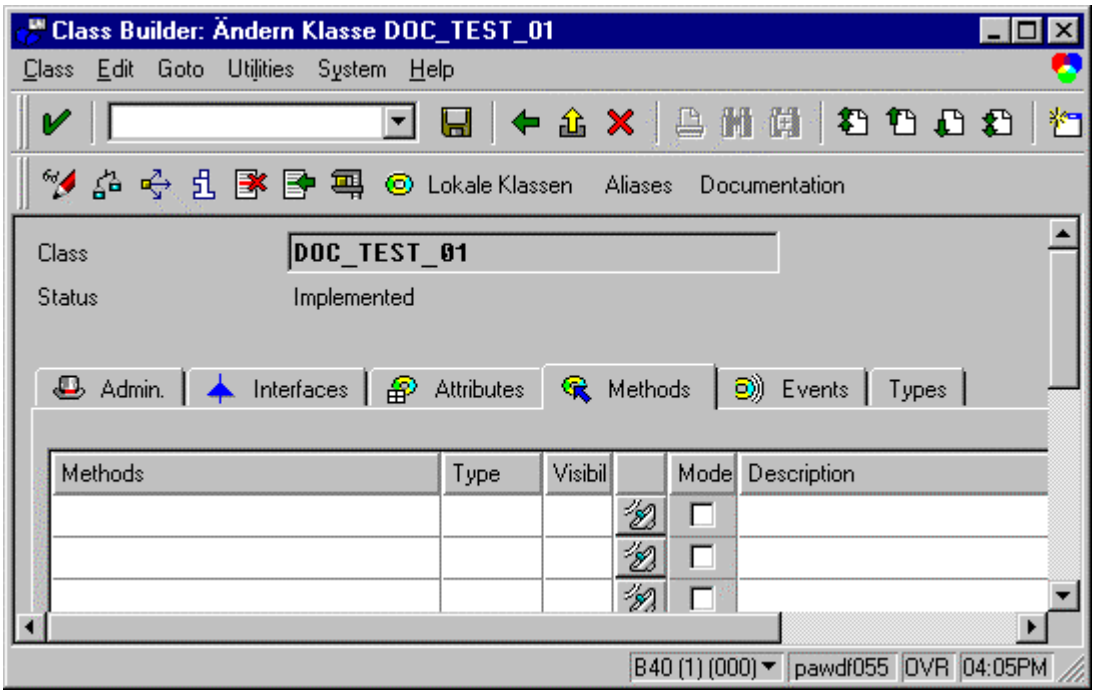
Class Editor

Implementation Considerations

The Class Editor is the part of the Class Builder in which you actually define the attributes, methods, events, and user-defined data types that make up the components of a class.

Integration

The class or interface components that you define in the Class Editor are saved in the class library. You can branch directly from the Class Editor to the ABAP Editor to write a method implementation.



Features

Basic Functions

The basic functions involve maintaining components, that is:

- Creating attributes [Page 226]
- Creating methods [Page 228]
- Creating events [Page 233]
- Implementing methods [Page 232]
- Creating interfaces in classes [Page 239]
- Creating internal types in a class [Page 235]

Other Functions

- The *Local classes* function allows you to maintain local auxiliary classes in the class pool of the global class.
- The *Aliases* function allows you to define short component names.
- The *Documentation* function allows you to document classes or interfaces and their components.
- The *Goto* menu allows you to jump to coding extracts.

Creating Attributes

Attributes contain data. They define the state of an object.

Prerequisites

You must already have created the internal data types in the class to which you want to refer when you create the attributes. For further information, refer to Creating Internal Types in Classes [Page 235].

Procedure

1. Start the Class Editor in change mode.
2. Choose *Attributes*.
3. To create an attribute, enter the following:

   - *Attribute*
   A unique name that identifies the attribute. Remember to observe the naming conventions in ABAP Objects [Page 207].

   - *Type*
   You can specify an attribute as a **constant**, an **instance attribute**, or a **static attribute** (that is shared by all instances of the class).

   - *Visibility*
   Defines the visibility of attributes to users of the class. If an attribute is **public**, it is assigned to the public section of the class and can be addressed by any user. Remember that public attributes form part of the external point of contact to the class, and as such stand in the way of full encapsulation.
   Protected attributes are visible in and can be addressed by all subclasses of the class.
   Private attributes are only visible in and can only be addressed by the defining class. In particular, they are not visible in the subclasses of the class.

   - *Modeled*
   If this option is set, the system does not enter the component in the class pool and the component cannot be addressed at runtime.

   - *Read Only*
   If this option is set, users cannot change this attribute.

   - *Typing method*
   ABAP keyword to specify the type reference. You can use **TYPE, LIKE** or **Type Ref To** (for class references).

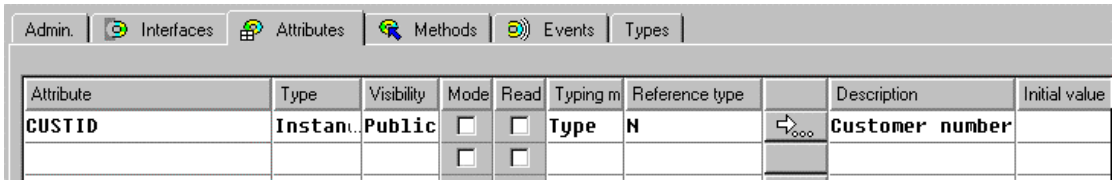
   - *Reference type*
   This may be any elementary ABAP type (including generic types) or an object type (classes and interfaces).

   - *Description*
   Short description of the component

- *Initial value*
If the attribute is a constant, you must specify an initial value.

4. Repeat steps 1 to 3 for each attribute.

   Example:



5. Save your entries.

Result

You have now created a set of attributes. The system generates the corresponding ABAP code in the definition part of the class or interface pool for all of the attributes except those for which you set the *Modeled* option.

Creating Methods

Methods describe how an object behaves. You implement them using functions defined within classes. They are operations that change the attributes of a class or interface. There are two types of methods: **Instance methods**, which always refer to a particular class instance, and **static methods**, which are shared by all class instances. Static methods can only address static attributes.

Prerequisites

You must already have created a class or interface. It is useful if you have already created the attributes of the class, since you can branch directly from a method definition in the Class Builder to its implementation.

The following description assumes that you are familiar with the principles of ABAP Objects.

Procedure

1. Start the Class Editor [Page 224] in change mode.
2. Choose *Methods*.
3. To create a method, enter the following information:

   - *Methods*
   A unique name to identify the method. Remember to observe the Naming Conventions for ABAP Objects [Page 207].

   - *Type*
   Specifies the type as an **instance method** or a **static method** (not instance-specific).

   - *Visibility*
   Defines the visibility of the method for the users of your class. If the method is **public**, it is assigned to the public section of the class, and can be called by any user. If you make the method **protected**, it is visible to and can be used by the class itself and any of its subclasses. If the method is **private**, it is only visible in and available to the class itself. Private methods do not form part of the external point of contact between the class and its users.

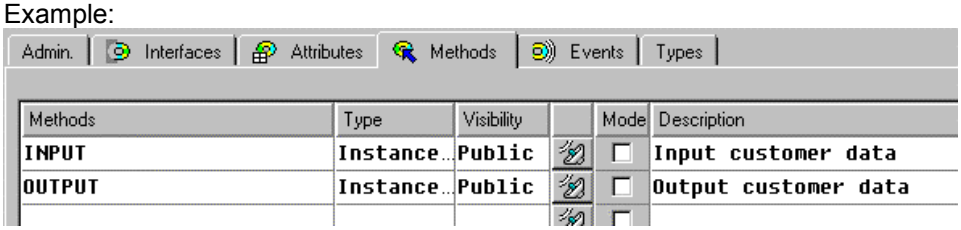
   - *Modeled*
   If you select this option, the system does not enter the method in the class pool. You cannot then address it at runtime.

   - *Description*
   Short description of the method.

4. Repeat steps 1-3 for any further methods.

If you create a constructor or class constructor method, it is assigned the predefined name CONSTRUCTOR or CLASS_CONSTRUCTOR respectively. The Class Builder also predefines certain other attributes.

Example:



5. Save your entries.

Result

You have now created methods for an object type. These are included in the definition part of the class or interface, that is, generated into the corresponding class pool or interface pool.

Before you can implement the methods, you must create your parameters and exceptions. For details of how to do this, refer to Creating parameters and exceptions [Page 230].

Creating Parameters and Exceptions

You define methods in a similar way to function modules. Firstly, you create interface parameters and exceptions. Then, you code (implement) the method. Methods can have input parameters - the **importing** and **changing** parameters, and output parameters - their **exporting**, **changing**, and **returning** parameters.

Prerequisites

- You must already have created the methods, attributes, and events of a class or interface.
- You must have opened the corresponding class or interface in change mode in the class editor and chosen either *Methods* or *Events*.

When you redefine inherited methods, you may not change the interface parameters (signature) nor add new parameters.

Procedure

Creating Parameters

1. Position the cursor on the name of the method or event.
2. Choose *Parameters*.
3. To create parameters of methods or events, enter the following information:

   - *Parameter*
   A unique name for the parameter. Ensure that you observe the naming conventions [Page 207] for method parameters in ABAP Objects

   - *Type*
   As in function modules, a parameter can have the type, **importing**, **exporting**, **changing**, or **returning**.
   Note the following special rules:
   If you use changing parameters, you cannot use returning parameters. If you use returning parameters, you cannot use exporting or changing parameters.
   Constructor methods may only have importing parameters.

   - *Pass Value*
   Unlike function modules, the default way of passing values to a method is by reference. However, you can force the system to pass a parameter by value by selecting this option. This is only possible for importing, exporting, and changing parameters. Returning parameters can only be passed by value. The Class Builder automatically checks against this rule.

   - *Optional*
   If you select this option, the parameter does not have to be specified when the method is called.

   - *Typing method*

ABAP keyword defining the type reference. You can use **Type**, **Like**, and **Type ref to**.

- *Reference type*
This may be any elementary ABAP type (including generic types) or object type (class or interface). For further information, refer to the Data Types [Ext.] section of the ABAP Programming Guide.
You can specify the type of a parameter of a private or protected method using an internal data type defined in the class.

- *Default value*
Default value for the parameter

- *Description*
A short description of the parameter.

Creating Exceptions

1. Position the cursor on the name of the corresponding method.
2. Choose *Exceptions*.
3. To define exceptions for methods, enter the following information:

   - *Exception*
   A name for the exception. Remember to observe the naming conventions [Page 207] for exceptions in ABAP Objects.

   - *Description*
   A short description of the exception.

Result

You have now created the interface parameters and exceptions for a method. You can now implement the method [Page 232].

Implementing Methods

Prerequisites

You must have created the methods and attributes of the class or interface. If you want to implement the methods of interfaces, the interfaces must have been listed in the class definition. You must have created any parameters and exceptions required by the methods.

Procedure

1. Start the Class Editor [Page 224] in change mode.
2. Choose *Methods*.
3. Position the cursor on the name of the relevant method.
4. Double-click, or choose *Source code*.
   The ABAP Editor appears, containing an empty statement block between the **METHOD** and **ENDMETHOD** statements.
5. Write the ABAP code for the method. .

   You can also create **text elements** in the source code using forward navigation.
6. Check the syntax of your ABAP code.
7. Save the code.
8. Choose *Back* to return to the class editor.
9. Document the method by choosing the corresponding function from the *Methods* display.

Creating Events

Objects can indicate that their state has changed by triggering a method. You can define events in both classes and interfaces, which you can then trigger from within a method using the RAISE EVENT statement. Each class (or interface) that is going to handle the corresponding event must implement a relevant handler method, and register it using the SET HANDLER statement. When an event occurs, the system calls all of the handler methods registered for that event.

Like method definitions, events have a parameter interface. The only difference is that events may only have EXPORTING parameters.

Prerequisites

You must already have created the class or interface.

The following description assumes that you are familiar with the principles of ABAP Objects.

Procedure

1. Start the Class Editor [Page 224] in change mode.
2. Choose *Events*.
3. Enter the following information:

   - *Event*
   A unique name to identify the event.

   - *Type*
   Specifies the event as an **instance event** or a **static event**.

   -*Visibility*
   Defines the visibility of the events as **public**, **protected**, or **private**.

   - *Modeled*
   If you select this option, the system does not enter the event in the class pool. You will not be able to access the component at runtime.

   - *Description*
   Short description of the event.

4. Repeat steps 1 - 3 for any further events.

5. Save your entries.

Result

You have now created events for your object type. The events are listed in the declaration part of the class or interface after the EVENTS statement.

You can specify events further by giving them EXPORTING parameters. For the procedure, refer to Creating parameters and exceptions [Page 230] and Implementing methods [Page 232].

For further information about event handling within ABAP Objects, refer to the syntax documentation in the ABAP Editor (for example, for the RAISE EVENT statement).

Creating Internal Types in Classes

Prerequisites

For up to date information about data types, classification, visibility, refer to the data types [Ext.] section of the ABAP User's Guide.

You must not create public data types within global classes.

Procedure

1. Start the Class Editor [Page 224] in change mode.
2. Choose *Types*.
3. To create an internal type within a class, enter the following information:

   - *Type*
   A unique name to identify the type. The recommended naming convention for internal types in classes is to use the prefix **TY_**.

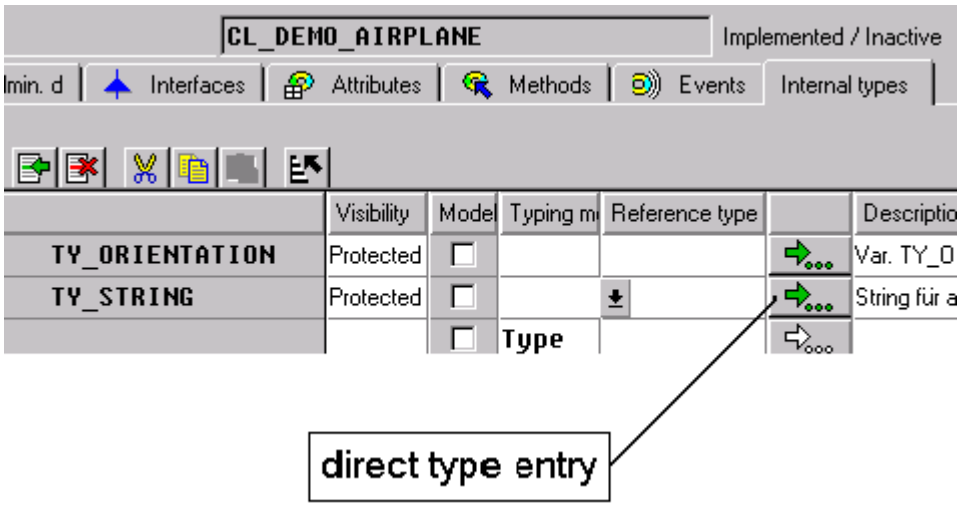
   - *Visibility*
   Defines the visibility. If you make the type **private**, it can only be accessed from within the class itself. If the type is **protected**, it is also visible to the subclasses of the class.

   - *Modeled*
   If you select this option, the system does not enter the type in the class pool. You will not be able to address the component at runtime.

   - *Typing method*
   ABAP keyword indicating the reference type. This can be one of TYPE, LIKE, or TYPE REF TO (for class references).

   - *Reference type*
   You can use any elementary ABAP type (including generic types) or object type (classes and interfaces).

   - *Description*
   Short description of the type.
4. Repeat steps 1-3 for any further types.
5. Save your entries.
6. If you need to qualify an internal data type in a class further (for example, to specify the field length of a character field), choose *Direct type entry*. Note that this only makes sense if you have not selected *Only modeled*.



The contents of the class pool appear for the corresponding visibility section.

7. Modify the data type

```
6   *" types
7   TYPES:
8   | TY_ORIENTATION(5) TYPE C ,
9     TY_STRING(1024) TYPE C . .
10
```

8. Check the syntax.
9. Save your entries.
10. Choose *Back* to return to the *Internal types* display.

Result

You have not created internal data types within your class. You can use these in your class, and, if they are defined as protected, also in its subclasses. You can define **private** and **protected** attributes and interface parameters using the **TYPE** addition.

Defining Relationships Between Object Types

You can define the following relationships between two object classes:

- **Inheritance** between two classes
- Extending the functions of a class by implementing interfaces. This is a relationship between classes and interfaces
- **Compound interfaces**. This is a relationship between interfaces.

Characterization

- Inheritance is a relationship between classes. It allows you to derive a new class from the definition of an existing class. The new class is called a **subclass**, the existing class is its **superclass**. Inheritance is used to create a subclass that is **more specialized** than its superclass. You can add new components to a subclass, and also **redefine** the methods that it inherits. The result of inheritance is a class hierarchy. The Class Builder allows you to create a class hierarchy very simply. You can create a subclass for any class that is not defined as final, and can also define a direct superclass for a class that was itself not derived.

- Interfaces allow you to extend a class definition. When a class implements an interface, all of the interface components appear as components of the class. You can access these components either using a class reference or an interface reference.
  Interfaces allow you to work with several different classes in a uniform way. The actual implementation of the interface components takes place in the classes. Consequently, interfaces provide a way of separating the definition and implementation of components.
- Interfaces can contain attributes, methods, and events, but also other interfaces. Classes that implement a **compound interface** must also implement all of its components. Compound interfaces are a specialization of their **component interfaces**.

Features

- Creating subclasses
- Redefining inherited methods
- Creating superclasses
- Assigning interfaces to classes
- Defining and implementing interface methods within a class definition
- Creating compound interfaces

Activities

Implementing Interfaces in Classes [Page 239]

Creating Subclasses [Page 241]

Nesting Interfaces [Page 245]

Implementing Interfaces in Classes

Interfaces are extensions to class definitions and provide a uniform point of contact for objects. Unlike classes, interfaces cannot be instantiated. Instead, classes implement interfaces by implementing all of their methods. You can then address them using either **class references** or **interface references**.

Each class can implement interfaces differently by implementing different coding for its methods. Interfaces thus form the basis for **polymorphism** in ABAP Objects.

Prerequisites

You must have created both the class and interface in the Class Builder.

Procedure

1. In the Class Builder, open the Class Editor in change mode.
2. Choose *Interfaces*.
3. To add an interface to a class, enter the following information:

   - *Interface*
   The interface name. When you press Enter, the system checks that the interface exists in the class library. You can use the possible entries help to display a list of all interfaces.

   - *Modeled*
   If you have selected this option, the system does not enter the interface in the class pool. You cannot access it or its components at runtime.

Repeat steps 1-3 for any further interfaces.

Example:

4. If the interface is nested, the system displays all of the component interfaces in the hierarchy after it has checked your input.
5. Save your entries.

Result

You have added one or more interfaces to the components of the class. The interfaces are listed in the definition part of the class in the class pool under the INTERFACES statement.

Implementing Interfaces in Classes

You must now implement all of the methods listed in the interface within the definition part of the class declaration. For further details, refer to Implementing methods [Page 232].

The components defined in the interface (attributes, methods, and events) appear in the class in the from <interface name>~<component name>. This ensures that no naming conflicts can occur with class components.

Creating Subclasses

Use

Inheritance allows you to derive classes from other classes. The new class contains a greater range of more specific functions than its superclass. You can do this by adding new components to the subclass or redefining methods inherited from the superclass.

Procedure

To create a new direct subclass from an existing class in the class editor:

1. Choose *Basic data* for the current class.
2. Choose *Subclass.*

   The *Create Class* dialog box appears.
3. Enter the following details for the subclass definition:

   - *Class*

   Name of the subclass you want to derive.

   - *Inherits from*

   Enter any global class that is not defined as final.

   - *Description*

   Enter a descriptive short text for the subclass.

   - *Instantiation*

   The default setting for all classes is **public**. This means that all users can instantiate the class using CREATE OBJECT. If you specify **protected**, only inherited classes or the class itself can instantiate the class. If you specify **private**, the class can only be instantiated by itself.
   These options allow you to restrict the instantiation of the class. For example, it provides the basis for managing persistent objects, where you have to ensure that objects are unique.

   - *Inheritance*

   If you specify *Abstract*, you can define an abstract class. Abstract classes cannot be instantiated, but you can use them as a template for further subclasses. You can only access an abstract class using its static components or subclasses.
   The *Final* option defines a final class. This class forms the end of the inheritance hierarchy, since final classes cannot have subclasses.
   If you define a class as both abstract and final, you can only access its static components.

   - *Only modeled*

   If you select this option, the subclass is not included in the class pool. You cannot access the class at runtime.
4. Choose *Save*.

   The *Create Object Directory Entry* dialog box appears.

5. Enter the development class.
6. Choose *Save*.

Result

You have now created a direct subclass for a class. The subclass inherits all components from the public and protected sections of the superclass, apart from the constructor methods. The interfaces implemented in the superclass are also implemented and visible in the subclass. You can display the inherited components of the subclass by selecting *With inherited*.
You can now specialize your class. For further information, refer to Extending Subclasses [Page 243].

Extending Subclasses

Changes to subclasses are additive, that is, you cannot delete a component from a class if it was inherited from a superclass. However, you can extend a subclass as follows:

- By adding new components
- By redefining inherited methods

  You can only redefine **instance methods**. Attributes, class methods, and other inherited components of a subclass cannot be redefined.
  Furthermore, the methods that you want to redefine may not have been defined as **final** methods in the superclass. Constructor methods may not be redefined, since they are implicitly final.
  A method redefinition may only extend to a new **implementation** of the method. The **signature** (names and types of the parameters) may not be changed. The interface of the redefined method must remain the same as that of the original method in the superclass.

Procedure

Adding New Components

You can define new components in all three visibility sections (public, protected, and private) of a subclass. Since both inherited and new components belong to the same namespace, you must ensure that all components in the class have **unique names**.
See also:

Creating Attributes [Page 226]

Creating Methods [Page 228]

Creating Events [Page 233]

Redefining Methods

To redefine an inherited method in a subclass from the class editor:

1. Display all methods of the subclass (select the *With inherited* checkbox).

   The system displays the inherited methods of all subclasses.

2. Double-click the method you want to redefine.

   A message appears, informing you that the method is already implemented in the superclass.

3. Choose *Continue*.

   The ABAP source code of the original method implementation appears.

4. Switch to change mode and reimplement the method.
5. Check the syntax.
6. Save your source code.
7. Document the newly-defined method by choosing the *Documentation* function from the *Methods* display.

Result

You have now extended the class, and your new components are visible, as well as the public and protected components of the superclass. Redefined methods appear in a different color in the class editor.
If you redefine a method in a subclass, the corresponding original method in the superclass remains unchanged.

You can access all visible components in the subclass in the same way - inherited components can also be addressed using their local name. If, however, you need to address components of the direct superclass, you can use the pseudoreference SUPER.

For further information, refer to the section on visibility of components.

Nesting Interfaces

ABAP Objects supports nested, or compound interfaces. A compound interface contains one or more interfaces as component interfaces. These may also contain interfaces as components, thus allowing multiple-level nesting of interfaces.
Interfaces that do not contain other interfaces are referred to as simple interfaces.

Use

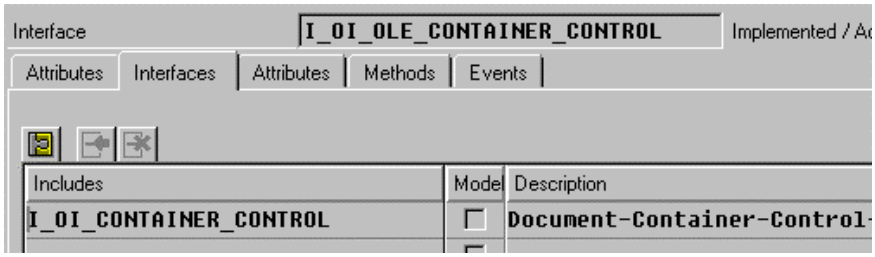
A compound interface is a specialization of its component interfaces. Component interfaces, as well as having their individual uses, can be combined to specify a new interface.

Prerequisites

All of the component interfaces must already exist in the class library.

Procedure

1. In the Class Builder, open the class editor.
2. Choose *Interfaces* for the relevant interface (making sure you are in change mode).
3. Under *Includes*, enter the names of the component interfaces.
   If you select the *Only modeled* option, the system does not create a corresponding entry in the class pool.
4. Press Enter.
   The system checks your entry against the class library, and inserts the short descriptions of the component interfaces.



5. Save your entries.

Result

All of the component interfaces that you entered belong to the same nesting level. When a class uses a compound interface, it must implement all methods of all component interfaces.
The component interfaces are implemented equally, regardless of their nesting level ("flat hierarchy"). You access their components using their original names, that is, the form:

<Interface name>~<component>

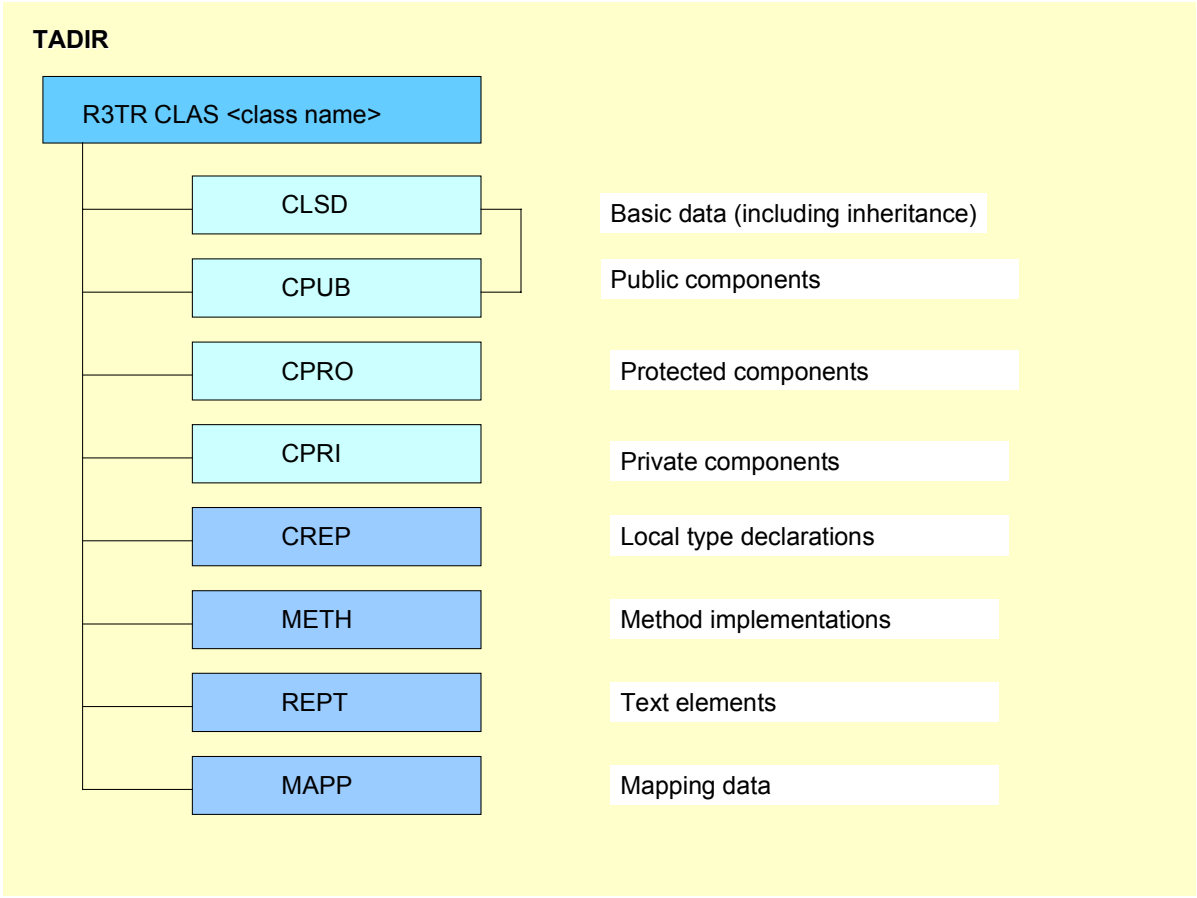
Activating Classes and Interfaces

Significance of Activation

When you create runtime instances, the system always uses active sources. You should remember this when instantiating classes (CREATE OBJECT statement), since this always refers to the activated class. All components of the corresponding global class that you want to access in the calling program must be activated explicitly.

Components of Global Classes

All global classes have an entry in table **TADIR**. The corresponding transport object for a class has the name **R3TR CLAS <class name>** and contains a range of components, each of which is a separate transport unit. Inactive class components appear in your worklist.



- The basic data and public components of a class cannot be activated separately.
- Only the basic data and public, protected, and private sections of a class affect the status display in the class editor. If you activate the entire transport object and then change a method implementation, the status remains *active*.

Components of Global Interfaces

The transport object for an interface has the name **R3TR INTF <interface name>**. It contains a single object with the name **INTF.**

When you activate a class that implements an interface, you must ensure that the interface has already been activated. Otherwise, the public section of the class contains a syntax error.

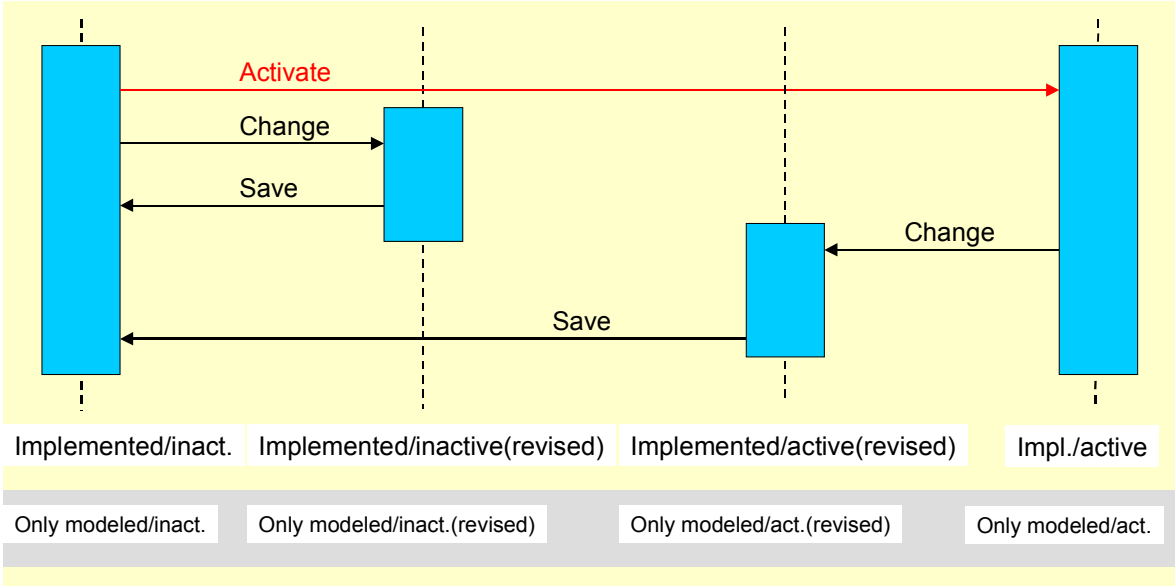
Status Display in the Class Builder

The current status of a class or interface is always displayed in the class editor. It is determined by the:

- Runtime relevance (*Implemented* ↔ *only modeled*)

- Database state (*Revised* ↔ *Saved*)
- Activation (*inactive*↔*active*)

There are eight possible statuses of classes or interfaces that can appear in the class editor:



Testing

Use

Use the test environment to test class components in the Class Builder and business object components in the Business Object Builder. The system dynamically generates a test program that simulates the execution of various ABAP statements.

You can only test the **public** components of objects.

Features

Use the test environment to

- Display access to the attributes of a class
- Simulate changes to the attributes of a class
- Simulate method calls
- Test the event handling using a standard handler
- Test an interface view of an object

Contents

Testing a Class

Requirements

Before you can test a class (object type), you should run the syntax check for it. If there are any syntax errors, you cannot start the test environment.

Procedure

You can use the test environment for classes in the Class Builder and for object types in the Business Object Builder.

In the Class Builder

1. Start the Class Builders [Page 202] (Transaction **SE24**).
2. Enter the name of the class.
3. Choose *Test.*

Alternative: Select the class that you want to test using the Repository Browser and use forward navigation to open the Class Editor [Page 224]. Then choose *Class* → *Test*.

In the Business Object Builder

1. Start the Business Object Builder (Transaction **SWO1**).
2. Enter the name of the object type.
3. Choose *Test.*

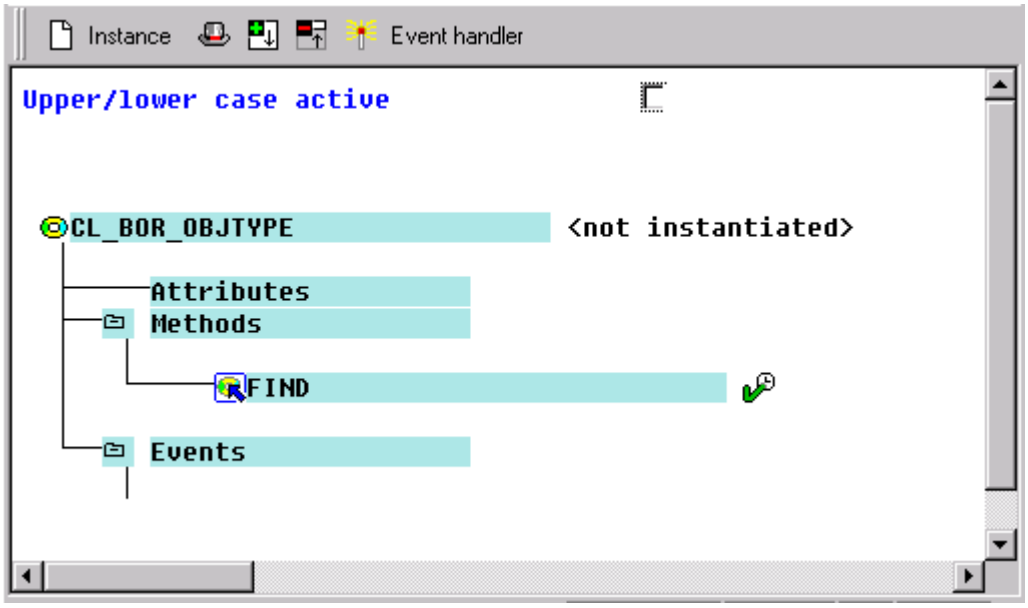
Alternative: Choose the object type that you want to test from the Business Object Repository and double-click to display it. Then choose *Test*.

Result

The system opens the test environment and displays the class in tree form.

Example:

Testing a Class



Creating Instances

You instantiate a class using the ABAP statement CREATE OBJECT. This calls the constructor for the instance. The constructor can contain parameters that you may have to supply with values.

In the test environment, the system automatically instantiates classes that have no static attributes or methods.

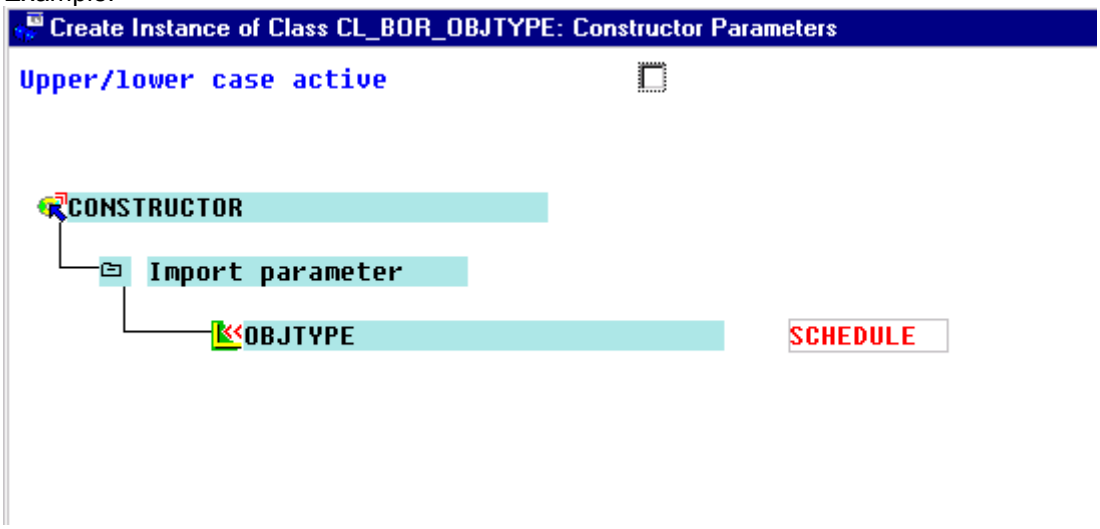
Prerequisites

You must have started the test environment [Page 251].

Procedure

1. Choose Instance.
   If the relevant constructor has parameters, a dialog box appears.
2. Enter valid values for the constructor parameters.

   Example:



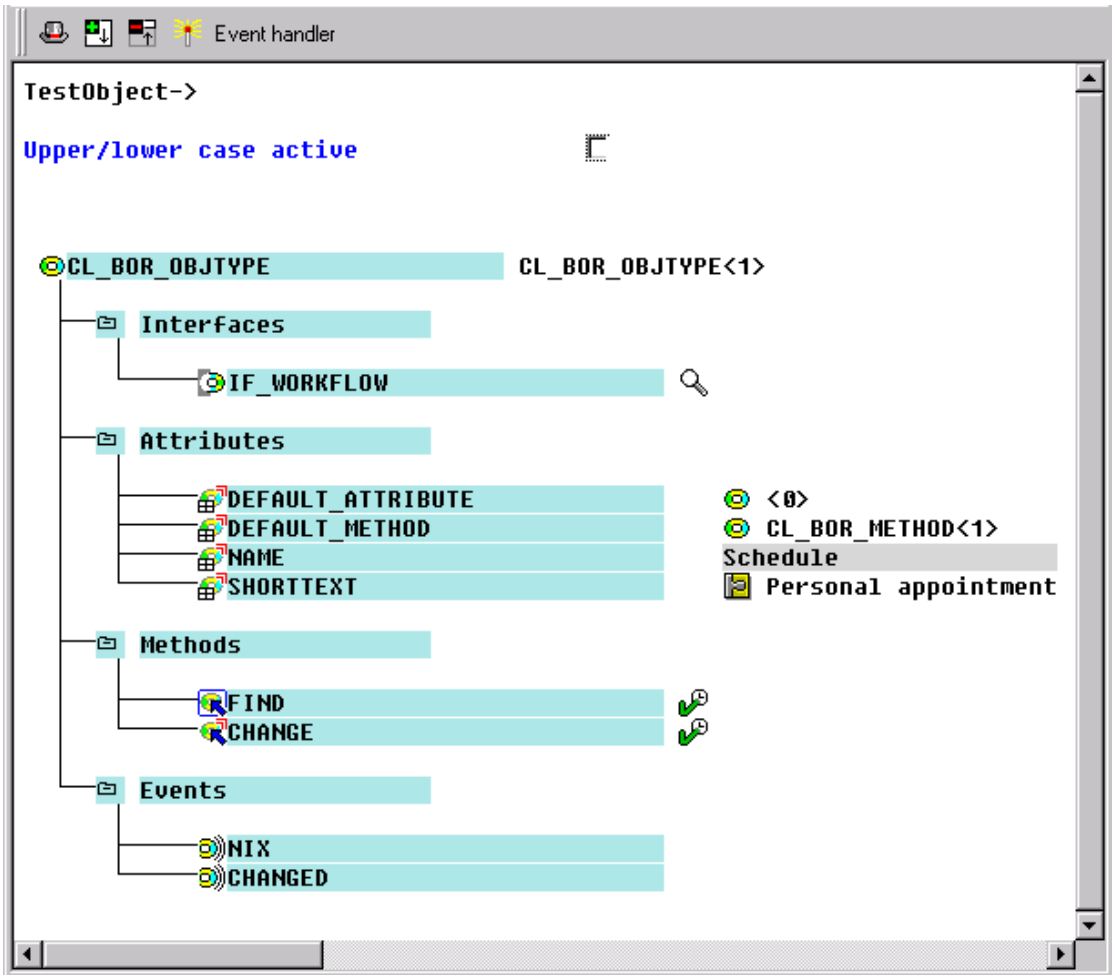
3. To create an instance, choose *Instance*.

Result

You have now created an instance (test object) for the class you want to test. Test object -> appears in the first line. The object ID is displayed next to the class name.

Example:

Creating Instances



Testing Attributes

When you have created a test object, you can address a container for attributes. This can be simulated for classes. You can both display and change the attributes.
The attribute display contains the following types of attribute field:

- Fields for direct input
- Lengthened fields
- Read only fields
- Complex data fields (tables)
- Attributes that are themselves object references.

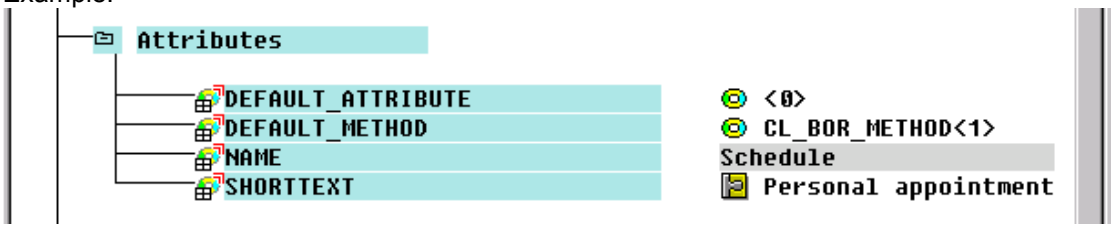
Prerequisites

You must have created an instance of the class you want to test. See Creating an instance [Page 253].

Procedure

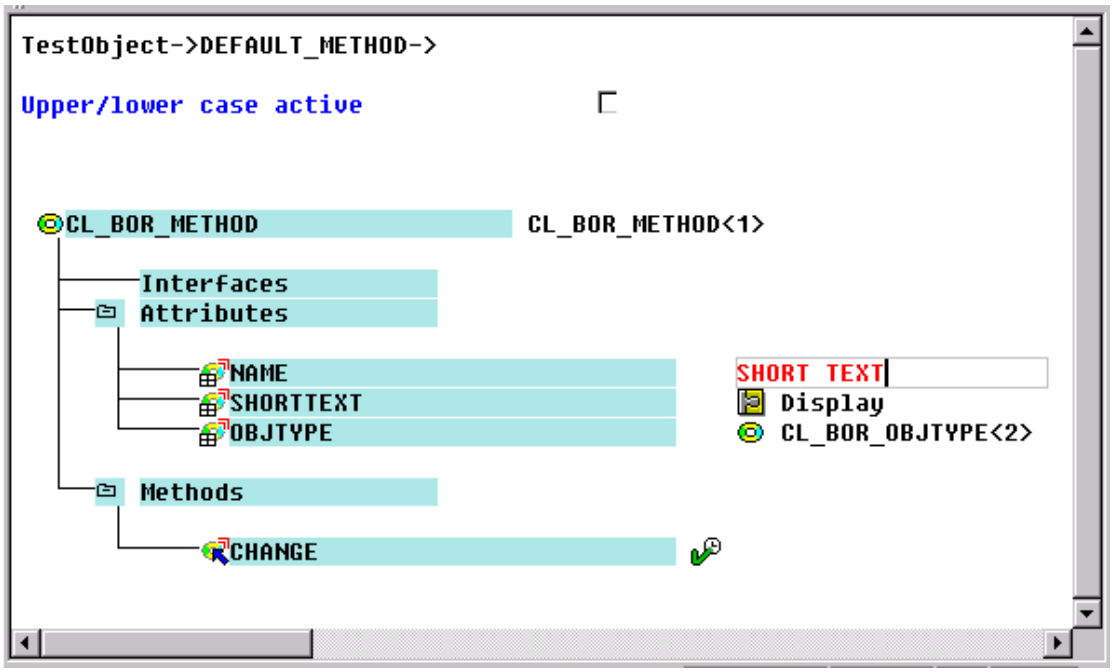
1. In the Attributes section of the object display, select the attribute that you want to test.

   Example:



2. Click the icon that corresponding to the data type of the attribute.
   If the field is an input field, you can change the attribute values. If it is an object reference, the relevant object is displayed.
3. If the field is an object reference, repeat step 2.

   Example:



Result

You have now tested the access to the attributes of a class. The access is displayed in the first line of the object display.

Testing Methods

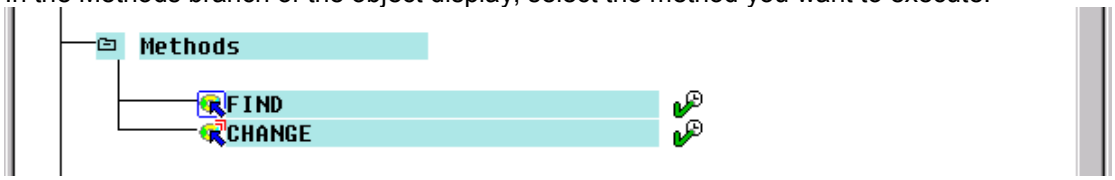
You can test access to methods of a class using a test object. If the method has no export parameters, the system does not display a result screen.

Prerequisites

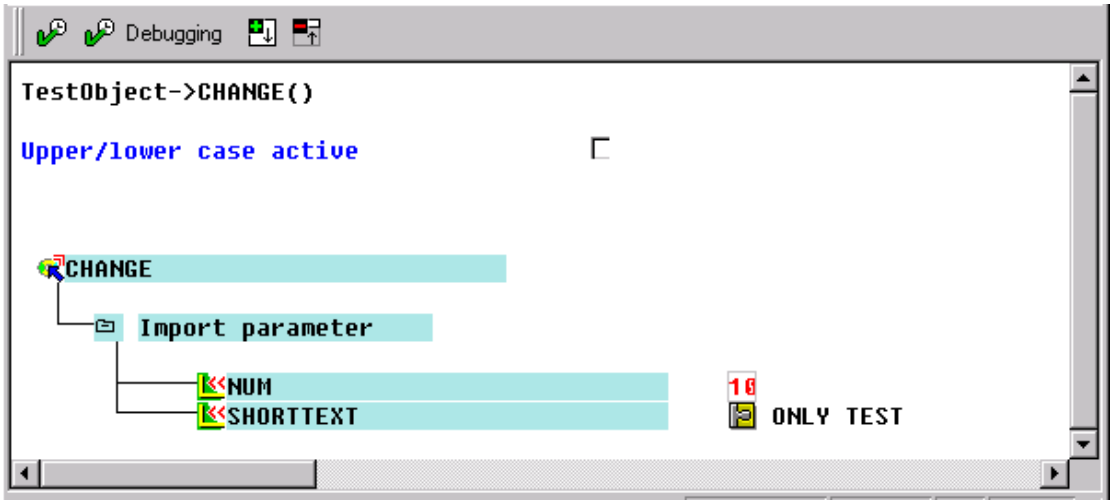
You must have instantiated the class you want to test (see Creating an instance [Page 253]).

Procedure

1. In the Methods branch of the object display, select the method you want to execute.



2. Click the *Execute* icon to execute the method.
   The appearance of the next screen depends on the method parameter definitions:
   a).
   If the method has import parameters, they are displayed, and you can assign new values to them. The system automatically checks their type.
   Then choose *Execute*.
   If your entries do not contain errors, the method is called using the parameters you specified.
   You can also execute the method in debugging mode by choosing *Debugging*.



   b).
   For any other parameters, the method is called directly, and a result screen appears, containing the *runtime*.

You can debug methods that do not have import parameters by choosing *Utilities* → *Debugging* → *Switch on debugging*.

Result

You have now used a test object to test the CALL METHOD statement for a class method.

If the method triggers an exception during the test, the system displays a dialog box containing the exception name and message text.

Testing Event Handling

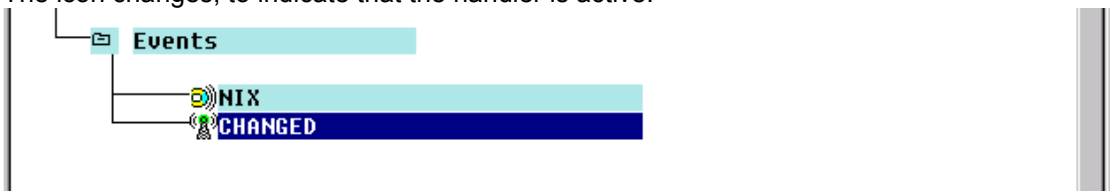
Within the test environment, you can test event handling using a default handler that catches the event when it is triggered. A simulation of the ABAP SET HANDLER statement assigns handlers to triggering events.

Prerequisites

You must have created an instance of the class that you want to test. Refer to Creating an instance [Page 253].

Procedure

1. In the *Events* branch of the object display, select an event.
2. Choose Handler, to enable the handler.
   The icon changes, to indicate that the handler is active.



3. Select a method in the object display.
4. Choose *Execute*. For further information about calling the method, refer to Testing methods [Page 257].

Result

If the method is called successfully, the chosen event is triggered and the result displayed.

To switch off the handler for the chosen event, choose *Utilities* → *Event handling* → *Switch off handler*.

Testing an Interface View of an Object

Interface views of objects allow you to test access to interface attributes and methods of an object, and so to simulate interface references.

Prerequisites

You must already have instantiated the class that you want to test. (see Creating an instance [Page 253]).

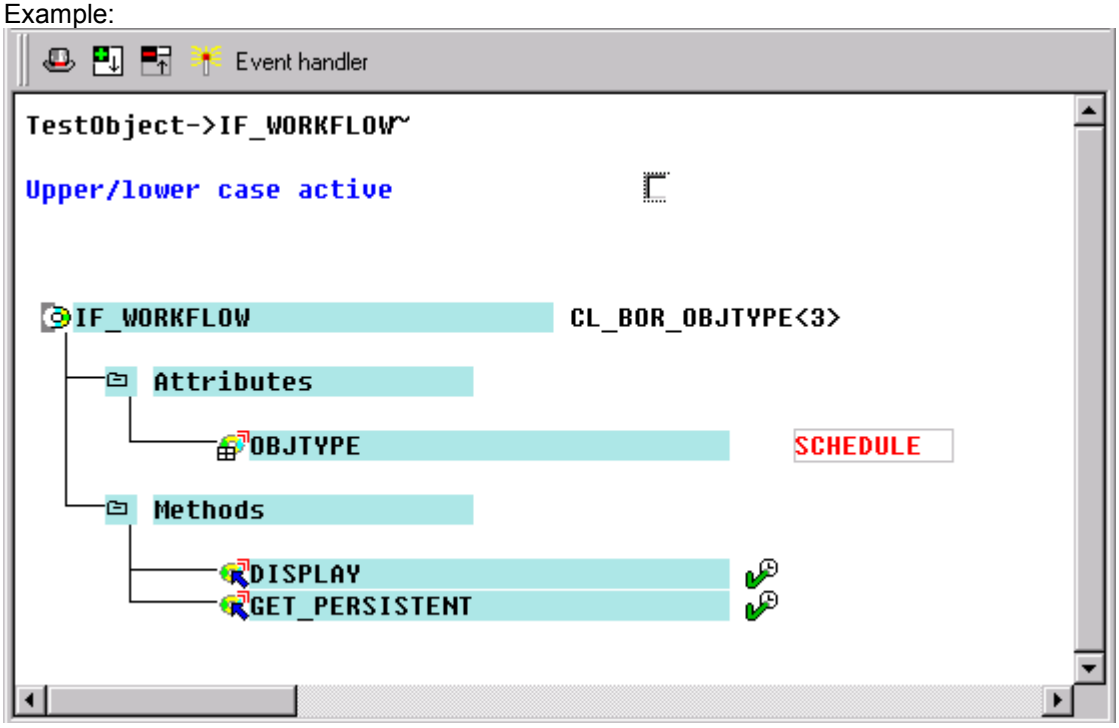
Procedure

1. In the object display, select an interface.

   Example:

2. Click the "magnifying glass" icon to generate an interface view of the object.
   The system generates the view of the test object.

   Example:



3. If necessary, change the attribute values (see Testing attributes [Page 255]).
4. If necessary, execute the methods (see Testing methods [Page 257]).

Result

You have not simulated accessing the attributes and methods of an object using interface references.

Screen Painter

The following documentation describes both the graphical and alphanumeric modes of the Screen Painter. This ABAP Workbench tool allows you to create screens for your transactions.

If you are using the Screen Painter in conjunction with the Modification Assistant, refer to the Modifications in the Screen Painter [Ext.] documentation.

Screen Painter Concepts

The Screen Painter is a ABAP Workbench tool that allows you to create screens for your transactions. You use it both to create the screen itself, with fields and other graphical elements, and to write the flow logic behind the screen.

In older documentation, screens are sometimes referred to as dynpros. This is short for “Dynamic Program”, and means the combination of the screen and its accompanying flow logic.

Screen Painter Architecture

You use the Screen Painter to create and maintain all elements of a screen. These are:

Screen Attributes	Describe a screen object in the R/3 System. Screen attributes include the program the screen belongs to and the screen type.
Screen layout	Screen elements are the parts of a screen with which the user interacts. Screen elements include checkboxes and radio buttons.
Elements	Correspond to screen elements. Fields are defined in the ABAP Dictionary or in your program.
Flow logic	Controls the flow of your program.

Two Screen Painter Modes

The Screen Painter has a *layout editor* that you use to design your screen layout. It works in two modes:

- Graphical mode [Page 274] and
- Alphanumeric mode [Page 315].

Both modes offer the same functions but use different interfaces. In graphical mode, you use a drag and drop interface similar to a drawing tool. In alphanumeric mode, you use your keyboard and menus.

Graphical mode is available only on MS Windows 95, MS Windows NT, and Unix/Motif platforms.

To activate the graphical mode, choose *Utilities* → *Settings* in the Screen Painter, then select the *Graphical layout editor* option.

Creating a Screen: Basics

1. Create a screen in an existing program and define its attributes.
2. Design the screen layout and define the attributes of the elements.
3. Write the flow logic.

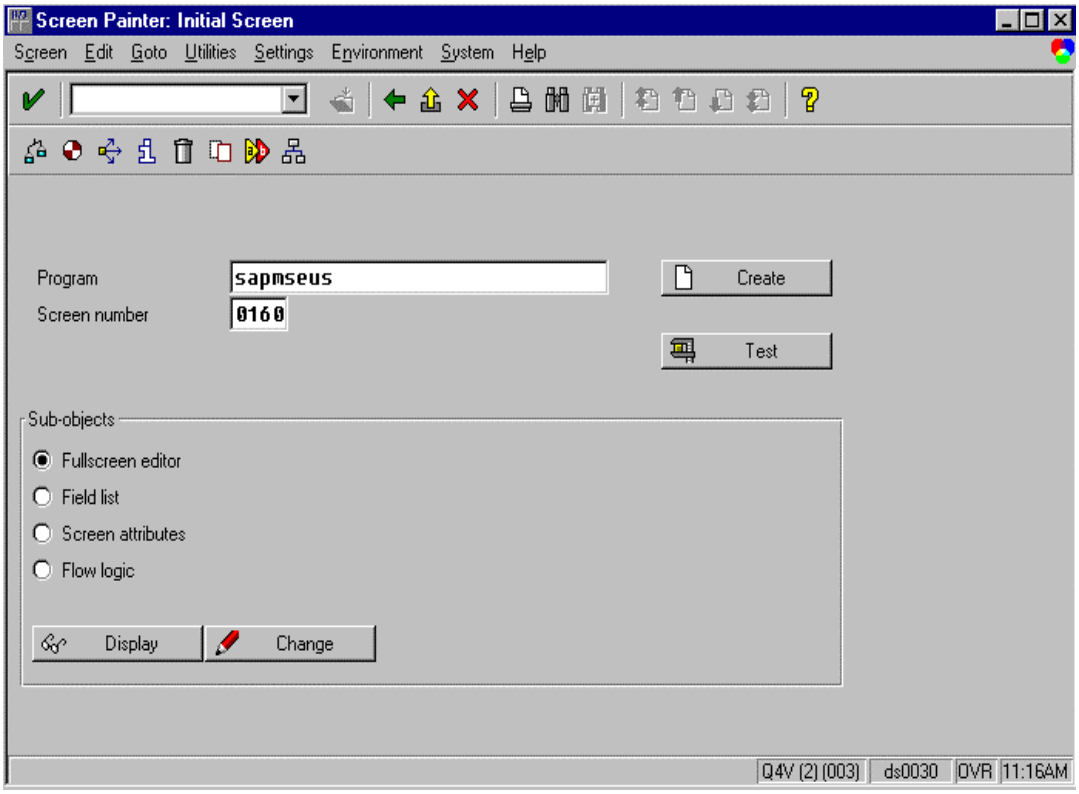
Basic Principles

- Uses predefined elements with links to the ABAP Dictionary or program.
- Supports forwards navigation.
- Supports complex elements: Table Control and Tabstrip Control, and Custom Container.
- Cut/ Copy/ Paste (*Graph. fullscreen)*
- Undo/ Redo (*Graph. fullscreen)*

Screen Painter: Initial Screen

To start the Screen Painter, choose the corresponding pushbutton on the initial screen of the ABAP Workbench or enter Transaction **SE51**. From here, you can:

- Create new screens.
- Test an existing screen.
- Create new components for an existing screen.



The *Object components* text box lists the different screen component views. Each view lets you edit a different aspect of a screen.

If you choose...	You can...
Layout Editor	maintain a screen's layout
Element list	Maintain the ABAP Dictionary or program fields for a screen and assign a program field to the OK_CODE field in the Screen Painter.
Screen attributes	maintain a screen's attributes
Flow logic	edit a screen's flow logic

Once you are within a particular view, you can use the Screen Painter's *Goto* menu to enter the other views.

Creating Screens

You can create a screen from the Screen Painter initial screen or from the object list in the Object Navigator.

Procedure

You create screens from the initial screen [Page 265] of the Screen Painter as follows

1. Start the Screen Painter.
2. Enter a program name.

   The program you specify should be an executable program (type 1), a module pool (type M), or a function group (type F) and must already exist.
3. Enter a screen number.

   A screen number must be unique and up to 4 numbers long. All screen numbers above 9000 are reserved for SAP's customers. The number 1000 is reserved for table screens and report selection screens. Initial screens of transactions are often given a number whose last three digits are 100 (for example, 3100).

   To display a list of a program's screens, use the possible entries button.
4. Choose *Create*.

   The system displays the *Change Screen Attributes* screen.
5. Define the screen attributes.

Screen Attributes

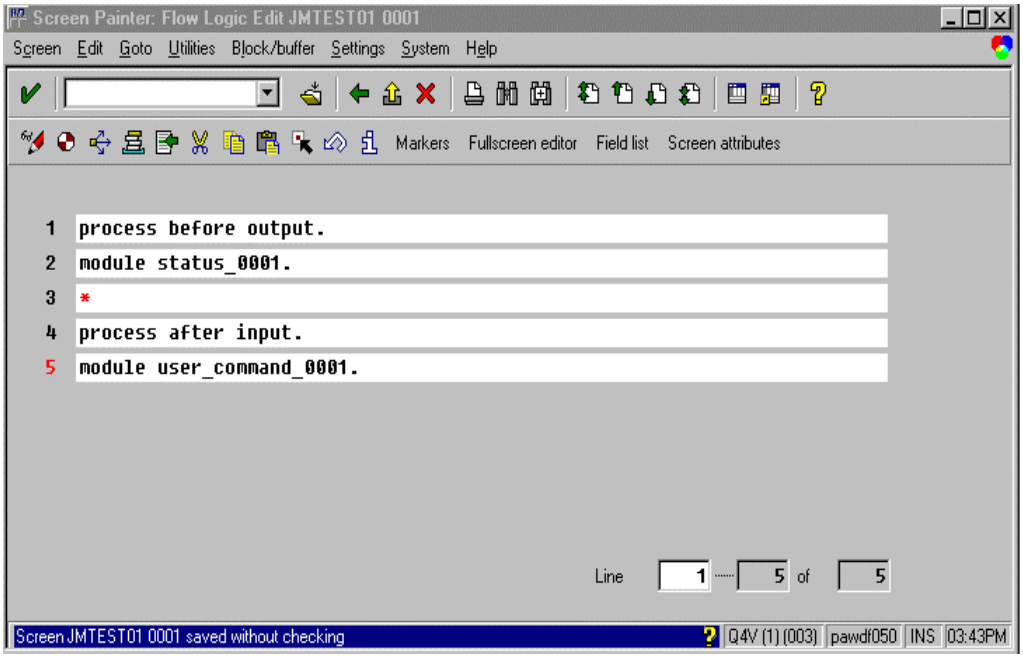
Screen attributes enable the system to assign and process a screen. You can set the following screen attributes:

Attribute	Description and ergonomic guidelines
program	Name of the module pool to which the screen belongs.
Screen number	Identifies a unique name up to 4 numbers long.
Original Language	Identifies a screen's maintenance language. When you create a screen, the system sets this value to the module pool's maintenance language.
Short description	Describes a screen's purpose.
Original language	Maintenance language for the screen. The system sets this attribute to the same original language as the module pool when you create the screen.
Development class	Identifies the development class the screen belongs to.
Last changed/ Last generated	Date and time that the screen was last changed or generated
Screen type	

Normal	If you set this option, the screen is flagged as a normal screen. This is the default setting.
Subscreen	Identifies the screen as a subscreen.
Modal dialog box	Identifies a specialized interface for display of lists in a dialog box. See Using Modal Dialog Boxes [Ext.] for more information.
Selection screen	Automatically-created screen. Selection screens are used to get values from the user at the beginning of a program. This data is used to restrict database selections. The system sets this attribute automatically.
Settings	
Hold data	The system only supports the *Hold data*, *Set data*, and *Delete data* functions (under System → User profile) on the screen if this option is selected. The system automatically redisplays the user's last entries from the screen when the user displays the screen a second or subsequent time.
Switch off runtime compression	If you set this option, the screen is not compressed at runtime. Ergonomic guideline: You should not use this option, since empty lines may appear on the screen if you hide fields dynamically at runtime. When gaps occur, users typically need longer to process the screen.
Hold scroll position	Use this option to specify whether the vertical and horizontal scroll positions should be retained for a screen. If you set the attribute, the scroll position is retained when the user returns to the screen after processing another screen. This also applies if the length or width of the screen changes, if other subscreens are used, or if the cursor is placed outside the visible area. This setting is intended for large screens on which the scroll position has previously been lost as a result of certain actions.
Other attributes	
Next screens	Number of the next screen to be displayed, assuming that the screen sequence is processed statically.
Cursor position	Element on which the cursor is positioned when the screen is first displayed. If you leave this field blank, the system positions the cursor on the first input field.
Screen group	Four-character ID for a group of logically-related screens
Lines/columns occupied	Size of the area occupied by screen elements.
Maintenance lines/columns	Screen size in lines and columns. The size of a screen is measured relative to the position of its top left-hand corner.

The Flow Logic Editor

To display a screen's flow logic from the Repository Browser, double-click a screen name. The system starts the flow logic editor of the Screen Painter:



The flow logic editor is similar to the ABAP Editor [Page 92] and provides functions for editing screen flow logic.

Getting Help on Screen Keywords

The flow logic offers help on flow logic keywords.

Position the cursor on the corresponding keyword and choose `F1`.

Navigation

The Screen Painter supports the same navigation features provided with all Workbench tools. You should be aware of the following features of navigation in the Screen Painter:

Double-click (F2) on...	To reach...
a field name in the element list or in flow logic	the point in the program where the field is defined.
The number of the next screen	the screen attributes of the next screen.
A screen number in the flow logic	the flow logic of that screen.
A module name in the flow logic	the point in the module pool where the module is defined.

The Screen Painter also offers the global find and replace features of the ABAP Workbench. You can find and replace element names (including generic searches) in the flow logic and element lists. Each search generates a hit list, from which you can navigate to the corresponding object.

Flow Logic Keywords

You define the flow logic in the flow logic editor of the Screen Painter, using the following keywords:

Keyword	Description
CALL	Calls a subscreen.
CHAIN	Starts a processing chain.
ENDCHAIN	Ends a processing chain.
ENDLOOP	Ends loop processing.
FIELD	Refers to a field. You can combine this with the MODULE and SELECT keywords.
LOOP	Starts loop processing.
MODIFY	Modifies a table.
MODULE	Identifies a processing module.
ON	Used with FIELD assignments.
PROCESS	Defines a processing event.
SELECT	Checks an entry against a table.
VALUES	Defines allowed input values.

For more information about transaction programming, see the ABAP User's Guide [Ext.]

Example of Flow Logic Example

The following example shows some use of screen flow logic:

```
*----------------------------------------------
*  Sample Code
*----------------------------------------------
*Processing Before Screen Output
PROCESS BEFORE OUTPUT.
  MODULE INIT_FIELDS.
* Self-programmed F1 Help
PROCESS ON HELP-REQUEST.
  FIELD GSSG-BUKRG MODULE V-BUKRG.
* Processing after user input
PROCESS AFTER INPUT.
* Lock customer master record
CHAIN.
```

```
    FIELD GSSG-KTNRG
        MODULE ENQUEUE_CUSTOMER_MASTER.
* Read customer master record
    MODULE READ_CUSTOMER_MASTER.
* Read business area
    MODULE READ_GSSG.
ENDCHAIN.
* Process function code
FIELD OK-CODE MODULE OKCODE ON INPUT.
```

Graphical Layout Editor

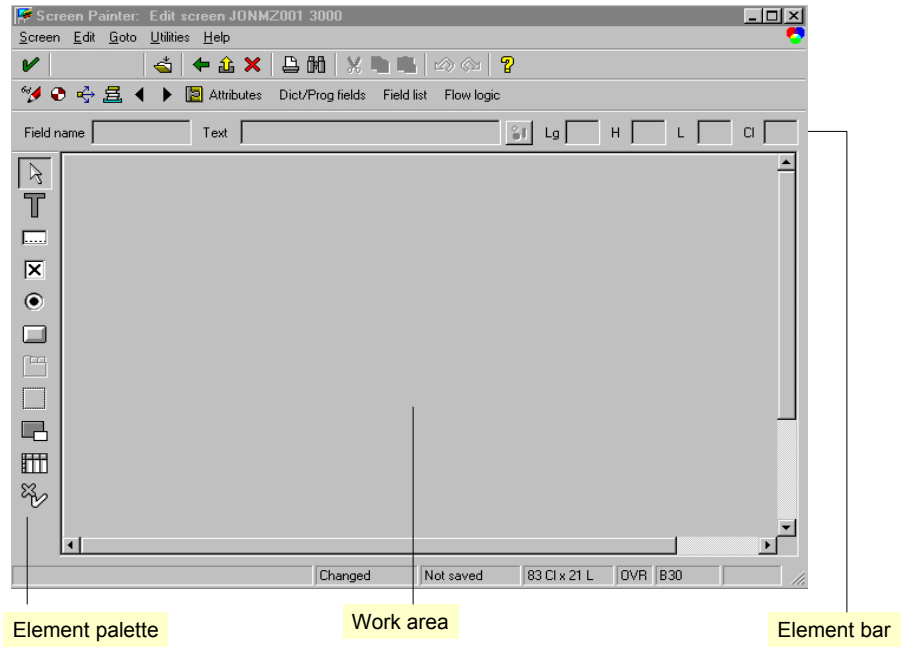
The graphical layout editor provides a user-friendly environment for designing screens. You can start the layout editor either from the initial screen of the Screen Painter or from the Repository Browser.

Starting the Layout Editor

To start the graphical layout editor from the initial screen of the Screen Painter:

1. Enter a program name and a screen number.
2. Choose *Settings* → *Graph. Screen Painter.*
3. Select *Graphical Screen Painter* and choose *Continue*.
4. Choose *Layout Editor* from the list of *Components* on the initial screen.
5. Choose *Change*.

   The system opens your screen in the layout editor.



Components of the Layout Editor

- **Element palette**, for creating screen elements. You can **drag and drop** these onto the screen. For further details about the individual elements, see screen element types [Page 279]
- **Work area**. This is the main part of the layout editor, in which you design the screen itself.
- **Element bar.**

  When you select a screen element, the major attributes associated with the element appear in this line. You can also change these attributes in the corresponding field.

Overview of Screen Layout

Once you have created a screen, you can start designing the screen layout.

You do this as follows:

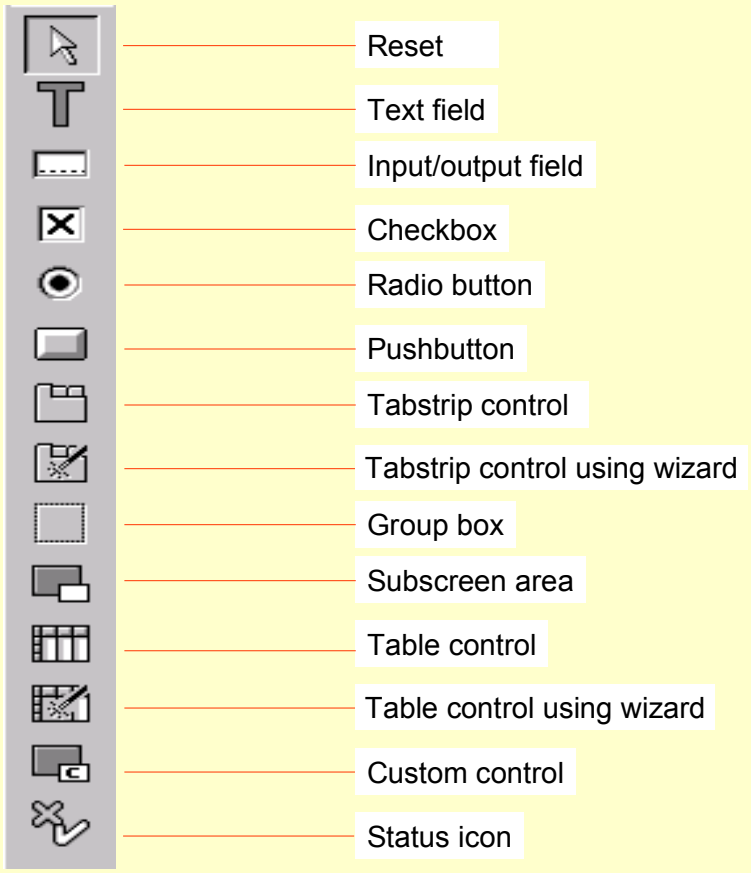
- Select the ABAP Dictionary or program fields that you want to use on the screen. For further details, see Selecting Fields [Page 283].
- Place any further screen elements [Page 279] in the work area.
- Once you have placed a screen element on the screen, you need to specify its attributes. For details of how to do this, see Working with Element Attributes [Page 330].
- Use the *Check* function to ensure that your screen layout conforms with the SAP ergonomic standards.

Screen size is measured in characters from a screen's top left corner. The standard screen size is 21 lines by 83 columns. The maximum screen size is 200 lines by 240 columns. The system maintains the screen title, menu bar, status bar and command field separately. These are not affected if you change the screen size.

Screen Elements

After you add an element to your screen, you can convert it to a another kind of element using the *Edit* menu. You can also use the element palette to select and place screen elements without first identifying fields.

Element Palette



Element Types: Overview

You can use the following screen elements in the Screen Painter (both graphical and alphanumeric):

Text Fields

Text fields provide labels for other elements. Text labels (sometimes called keywords) are display-only elements: neither the user nor the ABAP program can modify them at runtime. Text elements appear in a fixed position on the screen.

Text elements can also include literals, lines, icons, and other static elements. They can include all alphanumeric characters. However, you cannot begin a text with an **_** (underscore) or a **?**

(question mark). If you use a text to label a radio button or checkbox, the text must have the same element name as the element it labels.

If the text consists of several words, join the words together with underscores. The underscores allow the system to recognize the words as a unit. They are replaced by spaces at runtime.

Input/Output Fields

Input/output fields are sometimes called templates. You use them for entering and displaying data. To define the size of an entry element, enter underscore characters in the *Text* field as follows:

You can also use any other characters to format your template. For numeric values, you can define a comma (,) as the separator and a period (.) as the decimal point. As the last character of the template, you can set a **V** as place holder for signs.

Input/Output fields have no text labels. To assign a label to one, place a text field next to it.

Dropdown List Box

This is a special type of input/output field. Dropdown list boxes contain a list of entries from which the user can choose one. You cannot type an entry into the text field of a list box.

For further information about defining list boxes in the Screen Painter, refer to the General Attributes [Page 336] section.

For information about how to program a dropdown list box, refer to the dropdown boxes [Ext.] section of the ABAP Programming manual.

Checkboxes

Use checkbox elements to allow a user to select one or more options in a group. Program control is not immediately passed back to a work process on the application server. Further selections are possible until the user pushes a button or chooses a menu function.

Radio Buttons

Radio buttons are exclusive selection buttons that belong to a logical group. If a user selects one, the other buttons in the group are automatically deselected. You must both add the buttons and define them as a radio-button group in order to make their selection mutually exclusive.

When a user selects a radio button, control is not passed back to a work process on the application server immediately. As with checkboxes, further selections are possible until the user either presses a pushbutton or selects a menu function.

Pushbuttons

You use pushbuttons to trigger a particular function. When a user chooses one, the system sends the associated function code to the underlying ABAP program. At that point, control automatically returns to a work process on the application server that processes the PAI (Process After Input) module.

There is currently no link to the interface defined in the Menu Painter. The system does not check whether the selected function codes correspond to a valid status.

A pushbutton label can be simple text or it can be dynamic text that changes at runtime. You must define fields in your program for dynamic text. For more information about transaction programming, see the ABAP User's Guide [Ext.]

Boxes

Boxes contain sets of elements that belong together - for example, a radio button group. They provide visual emphasis but have no other function.

The top edge of a box normally contains a left-justified header. This header can be either a text field or an output field. If the header is a field element that is empty at runtime, the lines of the box are closed.

Tabstrip Controls

Tabstrip controls are complex graphical elements. For further information, refer to:

- Using Tabstrip Controls [Page 291]
- Creating a Tabstrip Control in Graphical Mode [Page 293].
- Creating a Tabstrip Control in Alphanumeric Mode [Page 324].

Tabstrip Control Wizard

The Tabstrip Control Wizard takes you step by step through the procedure for creating a working tabstrip control.

For further information, refer to Using the Tabstrip Control Wizard [Page 297].

Subscreen Areas

Subscreen elements are rectangular areas of a screen reserved for displaying other screens at runtime. A subscreen area cannot include any other screen elements. You use subscreens to include other screens within your main program.

To use a subscreen, define a second screen that appears in a subscreen area of the first screen.

Table Controls

Table controls are also complex graphical elements. For further information, See also:

- Table Controls [Page 299].
- Creating a Table Control in Graphical Mode [Page 301]. Creating a Table Control in Alphanumeric Mode [Page 321].
- Editing Table Controls [Page 305].

Table Control Wizard

The Table Control Wizard takes you step by step through the procedure for creating a working table control.

For further information, refer to Using the Table Control Wizard [Page 303].

Custom Container

You can use the custom container to embed one or more controls within a screen area. The custom container, like other screen elements, supports resizing and compression. In the Screen Painter, a rectangular area appears as a placeholder for one or more controls. The control itself does not appear on the screen until runtime.

You can define in the element attributes whether you want the control to be resizable.

For further information

- About the resizing attributes, refer to Custom Control Attributes [Page 347].
- About how to create containers, refer to graphical mode [Page 306] or alphanumeric mode [Page 326].
- About controls, refer to Control Enabling Technology [Ext.].

Status Icons

Status icons are output fields that contain an icon. The icon is specified at runtime. You use icons to indicate statuses in your application. Icons are predefined in the system and are each two to four characters long.

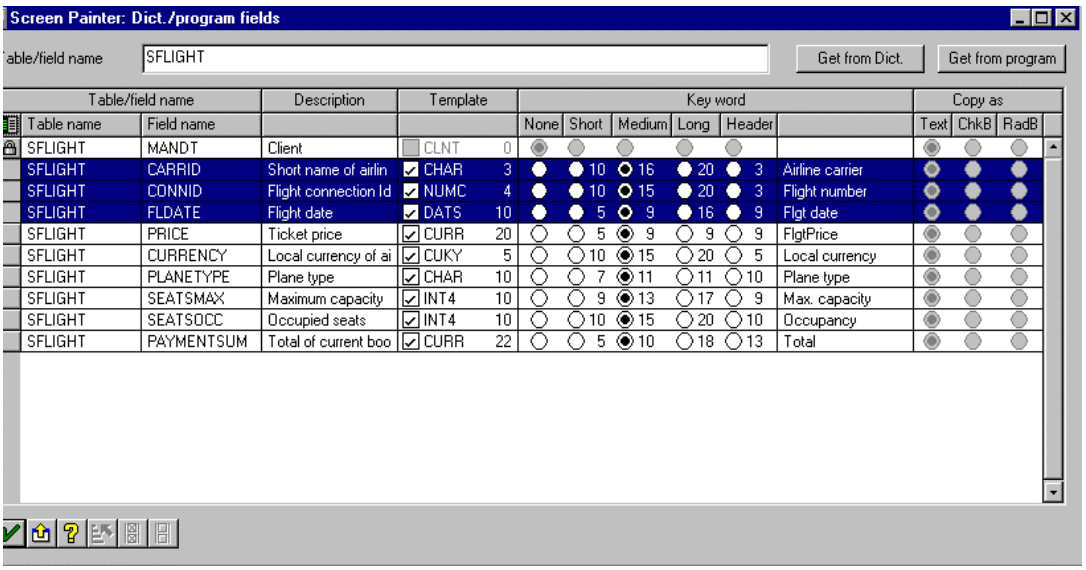
Selecting Fields

You add an element to a screen by using an existing ABAP Dictionary or program field.

Except for text elements, every element on your screen must have a corresponding Dictionary or program field. When you select fields defined in the ABAP Dictionary, your elements come equipped with the information already in the Dictionary, such as labels and formats. Additionally, if you use Dictionary fields, the fields are automatically updated whenever their definition in the ABAP Dictionary changes.

Using the Dict./program fields Dialog Box

You can list fields from the ABAP Dictionary or your program in the *Dict./Program Fields* dialog box. You can open the *Dict/Program fields* dialog from the initial screen of the Screen Painter or using the *Goto* menu on any other screen in the Screen Painter.



Using Dictionary and Program Fields on a Screen

1. Open the *Dict.program fields* dialog box.
2. Specify a table or field name.

   You can enter field names generically, but you must always specify table names in full.
3. Transfer the program or table fields onto the screen.

   To search for a table or field in the ABAP Dictionary, choose *Get from Dict*. To search for a table or field in the ABAP program, choose *Get from program*. The system displays the fields.
4. Select one or more fields.
5. Choose *Continue*.

   The system displays the cursor and an outline of the field block in the work area.

6. Single-click to position the field block on the screen.
7. Single-click outside the field block to deselect the block.

If you copied several fields, they are copied as a group. After copying, you can move each field individually. If you have included blank fields or invalid fields in your selection, the system displays a message and deletes the fields. These fields are not copied.

Canceling a Copy

If you discover before placing a field on the screen that your selection is incorrect, you can cancel the copy. To do this, choose *Reset* in the element palette.

Creating Screen Elements without Fields

You can also add elements without first selecting fields from the *Dict./program fields* dialog. You may prefer this if you want to add all elements to the screen before declaring corresponding fields in your program or defining them in the ABAP Dictionary. SAP does not recommend this method, however, since you lose the advantage of the centralized Dictionary definitions.

Entering Field Names

When you add elements without corresponding fields, you must also provide a field name explicitly (the *Field name* field in the fullscreen editor). Enter a field name (up to 40 characters long). The name can contain only alphabetic and the special characters **#** (pound sign), **/** (forward slash), **-** (dash), **_** (underscore),$ (dollar sign).

Special Case

If you enter a field name with a **-** (dash), the system checks to see if the ABAP Dictionary contains the field. If it does, a dialog box appears, in which you can decide whether your field should refer to the ABAP Dictionary field.

Modifying Screen Elements

There are some operations common to all screen elements: selecting, moving, and deleting. Some elements can also be resized. Selecting, moving, and deleting elements is different if you are using the layout editor in alphanumeric mode [Page 315].

Selecting

To select a single element, single-click it.

You can select several elements by dragging a rectangular "rubber-band" around them as follows:

1. Position the cursor outside the group.
2. Press and hold the left mouse button.
3. Drag the cursor away from its initial position.

   The system displays a "rubber-band" to indicate the size of your selection.
4. Ensure the "rubber-band" encloses the entire group.
5. Release the left mouse button.

You can also select multiple elements by holding down the `CTRL` key while clicking on each element.

Use the arrow keys  to select individual elements one by one.

Deleting

You can delete any screen element by selecting it and choosing *Edit* → *Delete*.

Note

When you delete a table control or a tabstrip control, all of its elements are also deleted.

Deleting a group box deletes only the box itself, not its contents.

Moving

To move an object, select it and use the mouse to drag it to a new position. You cannot drag elements to positions that overlap with other elements.

You can select several elements and move them as a group. Some elements have “handles” on their top edge that you must use to reposition the element. To move a grouped element, select it and then drag it by its handle to a new position.

Elements in a box element behave just like other grouped elements. When you move the box, the elements it contains go with it.

Resizing a Screen Element

1. Select the element.
2. Click on the right or bottom edge of the element.

   The mouse cursor changes shape if the object can be resized.

3. Resize the element by dragging the edge with the mouse.

You can also resize the screen itself by selecting and dragging the screen border. Alternatively, you can change the number of rows and columns in the element attributes.

Undo/Repeat 

You can always undo or repeat your last action, even if you deleted an element, using the undo and repeat functions.

Cut, Copy, and Paste 

You can cut, copy, and paste screen elements using the corresponding functions in the *Edit* menu or in the toolbar.

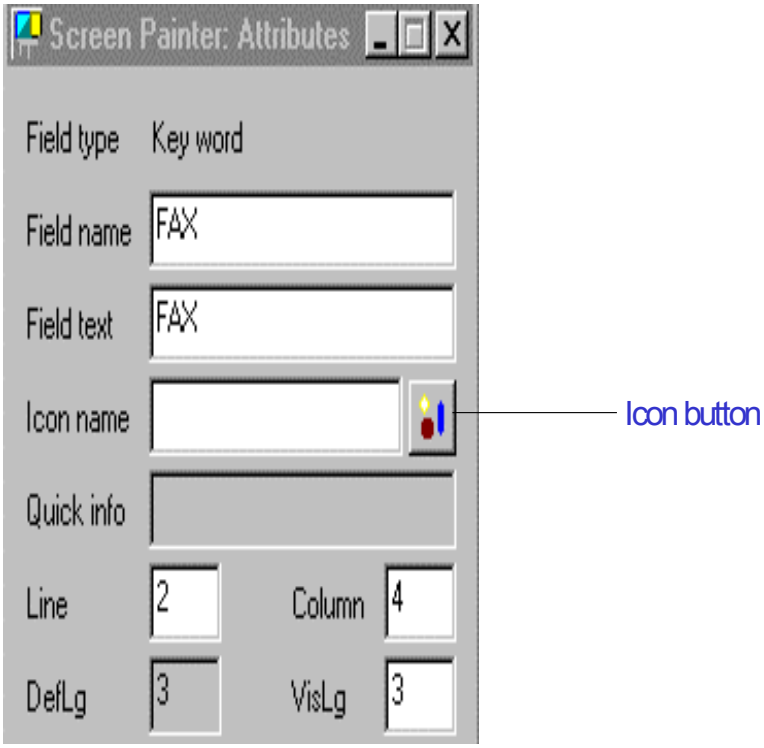
Using Icons

You can use icons in place of or together with text elements, pushbuttons, checkboxes, and radio buttons. Different icon types are available, depending on the element type. If you attempt to use an invalid icon, the system displays an error message.

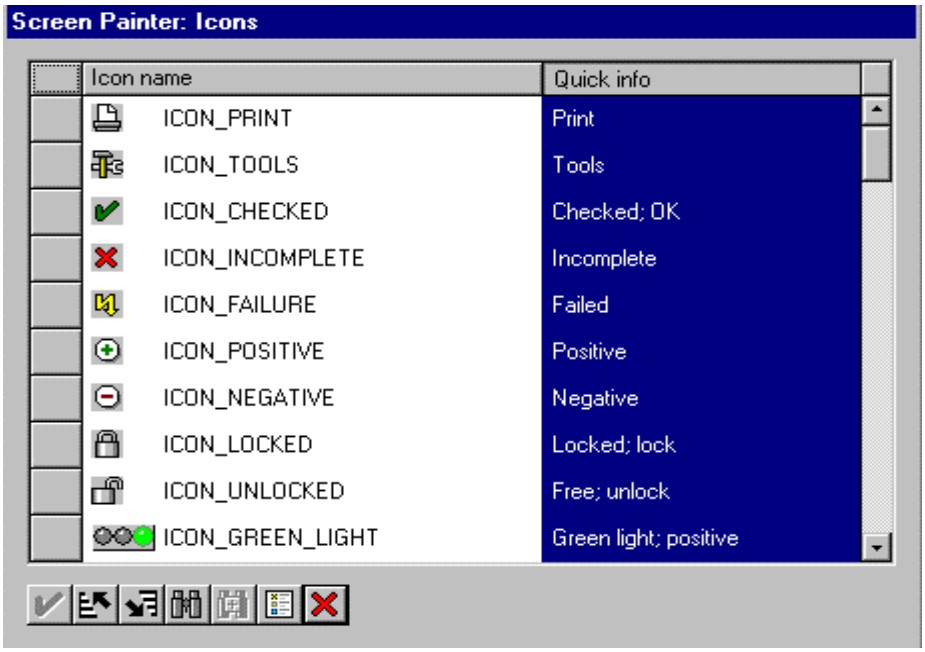
To replace or supplement texts with icons:

1. Select the element to which you want to add an icon.
2. Choose *Attributes*.

   The *Screen Painter: Attributes* dialog appears.



3. In the *Icon name* field, you can enter an icon name directly. However, since you usually do not know icon names by heart, you can click the icon pushbutton to list the available icons:



To find the desired icon, you can on this screen sort a column alphabetically, search for a certain term, or display an overview of all icons.

If you want to use an icon for a template, you must select the *With icon* field. See also General attributes [Page 336].

4. Close the *Attributes* dialog.

Using Radio Buttons

Combining screen elements into a radio button group means you can manipulate them as a unit, instead of one by one. Before you can create a radio-button group, you must create the individual radio buttons separately.

If you want to include checkboxes in a radio button group, you must first convert them into radio buttons.

Defining a Radio Button Group

1. Select the individual radio buttons.
2. Choose *Edit* → *Grouping* → *Radio button group* → *Define*.

   The system defines a radio button group.

Adding Radio Buttons to an Existing Group

1. Select an existing group.
2. Choose *Edit* → *Grouping* → *Radio button group* → *Dissolve*.
3. Add the new buttons to the selection.
4. Select the group and choose *Edit* → *Radio button group* → *Define*.

   The system displays a dotted line around the new group.

Tabstrip Controls

Definition

A tabstrip control is like a card index file that contains various screens belonging to a single application. Unlike normal screen sequences, the user can always see the names of the other screens to which he or she can jump immediately. This makes tabstrip controls an easy way to define several components of an application on a single screen and to navigate between them. The structure of your application thus becomes clearer to the user.

Use

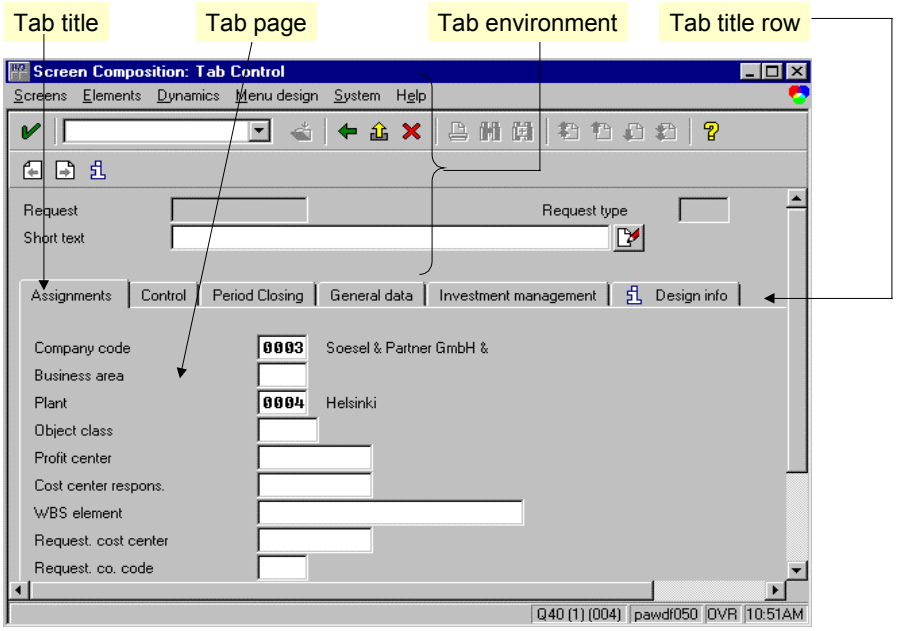
You can use tabstrip controls:

- To give complex applications a uniform structure and make it easier for users to navigate between their components.
- To make the structure of the application easier for users to learn and understand.

You should consider using tabstrip controls whenever a single object or application has different components or logical views that users need to navigate between. Property sheets are typically used to enter the attributes of complex objects.

Structure

The following illustration shows the essential components and environment of a tabstrip control:



Tabstrip Controls

Tab Title

Tab titles contain the titles of the other components to which the user can navigate. Technically speaking, they are **pushbuttons**. You can use icons on a tab title in exactly the same way as you would on a pushbutton. The text on a tab title should be short and meaningful.

Tab Title Row

The first row of the tabstrip control is reserved for the tab titles. If you have defined more tab titles than can be displayed at once, the system automatically displays scroll buttons in the top right-hand corner of the tabstrip control to enable you to scroll between the tab titles.

Tab Page

A tab page contains a collection of fields that logically belong together. This is the equivalent of placing a group box around a group of fields. Tab pages are implemented using **subscreens**.

Tab Environment

The screen environment around the tabstrip must remain constant. When you change between tab pages, the menus, application toolbar, and other fields outside the tabstrip control must not change.

For further information about designing and using tabstrip controls, see Transaction **BIBS**. Under *Elements*, you will find a sample program for a tabstrip control. Under *Rules*, there are tips for designing tabs.

Defining a Tabstrip Control

To define a simple tabstrip control, you must:

1. Define the tabstrip area.
2. Define the tab titles.
3. Define and assign a subscreen area.
4. Program the flow logic.

To find out how to create a tabstrip control in the alphanumeric Screen Painter, refer to Creating Tabstrip Controls in Alphanumeric Mode [Page 324].

Procedure

Defining the Tabstrip Area.

To define a tabstrip area using the graphical layout editor:

1. With the Screen Painter in change mode, choose the *tabstrip control* icon from the element palette.
   The mouse pointer changes its shape.
2. Set the position of the top left-hand corner of the tabstrip with a single click, and keep the mouse button pressed.
3. Drag the object out to the required size and release the mouse button.
4. If necessary, you can still change the position and size of the tabstrip control.
5. Assign a name **<tab_strip_name>** to the new tabstrip control.
6. Enter any further tabstrip attributes [Page 345] as required.

The element name of the tabstrip control is the name that you use to declare the tabstrip control in your ABAP program using the following statement:

```
CONTROLS <tab_strip_name> TYPE TABSTRIP.
```

Defining The Tab Titles

By default new tabstrip controls come with two tab titles. In technical terms, you process tab titles exactly like pushbuttons. If you want your tabstrip control to have more than two tab titles, you must change the *Tab titles* attribute..

1. Double-click a tab title to open the corresponding attribute window (for the pushbutton)
2. Assign the tab title attributes:

Attribute	Meaning
Name	Name of the pushbutton that forms the tab title
Text	Text for the button

Icon name	Icon to be displayed as part of the title. Note: this is **not recommended** for ergonomic reasons. The exception to this are icons for status displays, or icons that are genuinely self-explanatory (phone, fax, alarm…) used instead of text.
FctCode	Function code that triggers the PAI event. When the user clicks the tab, the function code is placed in the system field SY-UCOMM. If you are scrolling at the backend, the function code is also placed in the OK_CODE field.
FctType	A tab title may have the function type <P> or <SPACE>. To scroll locally at the frontend, use type <P>. In this case, the PAI event is not triggered when the user chooses a tab title, and there is no data transfer to the application server. To scroll at the backend, use function type <SPACE> (no special type assignment). In this case, the PAI event is triggered when the user chooses a tab title, and the function code is placed in the OK_CODE field.

3. Repeat steps 1 and 2 for each new tab title.

Just like normal pushbuttons, you can assign dynamic texts to tab titles.

Assigning a Subscreen Area

You must assign a subscreen area to each tab page. If you are using local scrolling at the frontend (function type <P>) you must assign a separate subscreen area to each tab page. If you are scrolling at the backend (function type <SPACE>), you can use one shared subscreen area for all tab pages.

To assign a subscreen area to a tab page:

1. Select a tab title.
2. Choose *Subscreen area* from the element toolbar.
3. Position the subscreen area within the tabstrip control, and drag it to the required size.
4. Enter a name **`<subscreen_area>`** for the subscreen area.
   This name also appears as the *reference field* in the tab title attributes.

You can also assign the subscreen area manually by entering the name of the subscreen name in the *reference field* attribute. Note that this only works if you are using backend scrolling.

Programming the Flow Logic

This explanation of the flow logic is restricted to that necessary to include the appropriate subscreen screens in the right subscreen areas in the tabstrip control. There are two ways of doing this, depending on the scrolling method you are using.

Local Scrolling at the Frontend

If you are using frontend scrolling, you must include subscreen screens in all of your subscreen areas in the tabstrip control.

You can do this in the screen flow logic as follows:

1. Add the following statements to the **PBO** event of your flow logic:

```
PROCESS BFORE OUTPUT.

    CALL SUBSCREEN: <subscreen_area1> INCLUDING [<progname 1>]
<subscreen_scrn 1>,
                    <subscreen_area2> INCLUDING [<progname 2>]
<subscreen_scrn 2>,
                    <subscreen_area3> INCLUDING [<progname 3>]
<subscreen_scrn 3>,
                   ...
...
```

Note that you can take the individual subscreen screens from different ABAP programs.

2. Add the following statements to the **PAI** event of your flow logic:

```
PROCESS AFTER INPUT.
...
    CALL SUBSCREEN: <subscreen_area1>,
                    <subscreen_area2>,
                    <subscreen_area3>,
                    ...
...
```

Scrolling at the Application Server

If you are scrolling at the application server, you only need to include one subscreen at a time in a single subscreen area.

To do this, you must:

1. Add the following statement to the **PBO** event of your flow logic:

```
PROCESS BFORE OUTPUT.
...
CALL SUBSCREEN < subscreen_area> INCLUDING [<progname>]
<subscreen_scrn>.
...
```

2. Add the following statement to the **PAI** event of your flow logic:

```
PROCESS AFTER INPUT.
...
CALL SUBSCREEN < subscreen_area>.
```

Result

You have now created a tabstrip control in the Screen Painter, and determined whether the tab pages should be scrolled at the frontend or on the application server.
You can now turn your attention to programming the reaction to user input in your tabstrip control.

For further information about handling tabstrip controls in your ABAP programs, refer to the Tabstrip Control [Ext.] section of the ABAP Programming Guide.

Using the Tabstrip Control Wizard

The tabstrip control wizard allows you to create and implement tabstrip controls quickly and easily. The generated dialog logic uses backend scrolling.

You can use tabstrip controls that you generate in this way as a basis for further refinement in your application.

Features

- Creates an instance of the tabstrip control.
- Defines the tab pages and associated function codes.
- Assigns a common subscreen area to all tab pages.
- Creates a new subscreen screen (unless you want to use an existing one).
- Generates the flow logic required for backend scrolling.
- Creates the PBO and PAI modules and all necessary data definitions.
- Creates the appropriate includes for the modules and data definitions (if required).

Procedure

To start the tabstrip control wizard:

In the graphical Screen Painter

1. Start the layout editor.
2. Switch to change mode if required.
3. Choose  from the element toolbar.
4. Define the tabstrip control area on the screen.

   The wizard then starts in a separate dialog box.
5. The wizard now takes you through the steps required to create a working tabstrip control. The whole process consists of five steps, in which you define the attributes of the tabstrip control and the ABAP code you want to generate. You can navigate between the dialogs using the *Continue* and *Back* functions. When you choose Finish, the system generates the tabstrip control.

In the alphanumeric Screen Painter

1. Start the layout editor.
2. Switch to change mode if required.
3. Position the cursor where you would like the top left-hand corner of the tabstrip control area to be.
4. Choose *Edit* → *Wizards for creating elements* → *Tabstrip*.
5. Position the cursor where you would like the bottom right-hand corner of the tabstrip control to be.
6. Choose *Mark end of ctrl*.

The wizard then starts in a separate dialog box.

7. See step 5 from the procedure for the graphical Screen Painter.

If an error occurs when the system tries to generate the tabstrip control or you cancel the wizard, all of the objects and source code created by the wizard are deleted, and the starting state of the program is restored.

Table Controls

Definition

A table control is an area on the screen in which you can display data in tabular form. You process it using a loop. Table controls are comparable to step loop tables. While a table control consists of a single definition row, step loop blocks may extend over more than one row. Table controls are more flexible than step loops, and are intended to replace them.

For further information about designing and using table controls, see Transaction **BIBS**. Choose *Elements* → *Table controls* to display sample table controls for various purposes. Choose *Rules* to display hints for designing table controls.

Use

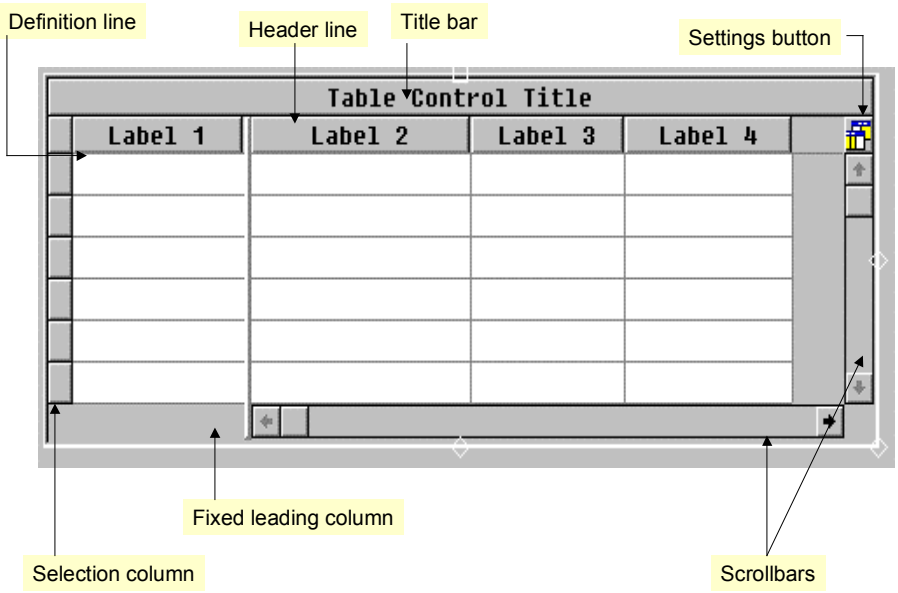
Table controls allow you to enter, display, and modify tabular data easily on the screen.

They provide the following functions:

- On definition:
  - Fixed columns
  - Column headers
- At runtime:
  - Vertical and horizontal scrolling.
  - Modifiable column width.
  - Row and column selection.
  - Movable columns
  - Settings can be saved.

Components of a Table Control

The following diagram shows the main components of a table control:



Note

- Lines in a table control may contain keywords, input/output fields, radio buttons, checkboxes, radio button groups, and pushbuttons.
- A line can be up to 255 columns wide.

Defining a Table Control

1. Define the table control area.
2. Define the table control elements.
3. Add a title (optional).
4. Declare the table control in your module pool.

Procedure

The procedures described in this documentation are different if you are using the fullscreen editor in alphanumeric mode. For further information, see Creating a table control in alphanumeric mode [Page 321].

Defining the Table Control Area

1. With the Screen Painter in change mode, choose the *table control* icon from the element palette.
   The mouse pointer changes its shape.
2. Drag and drop the object onto the screen work area.
3. Resize or reposition the table control if necessary.
4. Assign an element name to the new table control.

Defining Elements

To define elements for your table control, you can use ABAP Dictionary fields, fields from the program, or completely new fields. If you are using ABAP Dictionary or program fields, follow the procedure described in Selecting Fields [Page 283]. If you want to create new fields, you use elements from the element palette as follows:

1. Choose an element from the element palette and place it in the definition line of the table control.
   The system creates a new column.
2. Assign an element name to the element, and any necessary attributes.
3. Create a column header by dragging a text or entry element into the column heading.
4. Repeat steps 1-3 for each additional column.
5. Set the remaining table control attributes [Page 346].

Adding a Title

1. In the table control attributes, select *With title*.
   A dialog box appears, reminding you to create a title element.
2. Enter a text field or an input/output element in the title row.
3. Enter an element name, and the title in the *Text* field.

Declaring the Table Control in the Module Pool

Insert the following `CONTROLS` statement in the global data declaration of your transaction:

```
CONTROLS <tab_ctrl_name> TYPE TABLEVIEW USING SCREEN <screen_no>.
```

For information on processing the table, see Editing Table Controls [Page305].

Using the Table Control Wizard

This wizard allows you to create a working table control quickly and easily. It also lets you generate certain standard table maintenance functions.

You can create table controls in this way and then adapt them to the particular needs of your application.

Features

- Creates an instance of the table control.
- Assigns an ABAP Dictionary or program table to the table control.
- Selects the table fields for the column definition.
- Assigns important table control attributes.
- Generates the relevant statements in the screen flow logic.
- Creates PBO and PAI modules, subroutines (for standard table maintenance functions), and all necessary data definitions.
- Generates standard functions for table maintenance (scrolling, insert/delete lines, select/deselect all).
- Creates includes for the modules, data definitions, and subroutines, if required.

Procedure

To start the table control wizard:

In the graphical Screen Painte

1. Start the layout editor.
2. Switch to change mode if necessary.
3. Choose  from the element toolbar.
4. Define the table control area on the screen.

   The wizard is now started in a separate dialog box.
5. The wizard now takes you through the steps required to generate a working table control. The process consists of seven dialogs in which you define the attributes of the table controls and the ABAP code that needs to be generated. You can navigate between the dialogs using the *Continue* and *Back* buttons. When you choose *Finish*, the table control is generated.

In the alphanumeric Screen Painter

1. Start the layout editor.
2. Switch to change mode if required.
3. Position the cursor where you would like the top left-hand corner of the table control area to be.
4. Choose *Edit* → *Wizards for creating elements* → *Table control*.

Using the Table Control Wizard

5. Position the cursor where you would like the bottom right-hand corner of the table control area to be.
6. Choose *Mark end of ctrl*.

   The system starts the wizard in a separate dialog box.

The procedure then continues as described in step 5 above.

If an error occurs when you generate the table control, or you cancel the wizard, all of the objects and ABAP coding generated by the wizard are deleted.

Editing Table Controls

Changing the Size and Position

You must select the table control in the Screen Painter before you can modify it. When a table control is selected, handles appear on three sides of it. You can use these to move or resize the table control.

Use the square handle on top of the table control to reposition it. Use the diamond-shaped handles to resize it.

Arranging Columns

- To rearrange the columns in a table control, choose a column and drag it to its new position. The system automatically rearranges the column headings.
- To move a column to a position that is not currently visible (scrolled off the screen):
  a) Select the column and its column heading (if it has one), and drag them out of the table control.
  b) Scroll through the table control until the required position is visible.
  c) Move the column and its heading back into the table control at the new position.

  You can also use the cut and paste functions to move columns within the table control. Alternatively, you can enter the new column number directly in the table control attributes.
- To delete a column heading, select it and choose *Edit → Delete*. You can delete the contents of a column in the same manner. When you delete a column, the system also deletes its heading.
- To allow users to select columns in the table control, set the *column sel.* attribute. You can also decide whether to allow single or multiple column selection. The default setting is single selection.

Editing a Row

You can also allow users to select rows in a table control. To do this, you must set the *Line sel.* attribute. Again, you can choose whether to allow single or multiple line selection. In addition, you must mark the *w/ SelColumn* field and enter a field name into the adjacent field. This name allows you to query which line the user selected from your ABAP program.

See also

Table Control Attributes [Page 346]

Creating a Custom Container

A custom container is a control into which you can place other controls. It allows the screen framework to resize and compress controls at runtime.

For further information about control technology, refer to the SAP Control Framework [Ext.] documentation.

.

Prerequisites

You must already have opened the graphical layout editor in change mode.

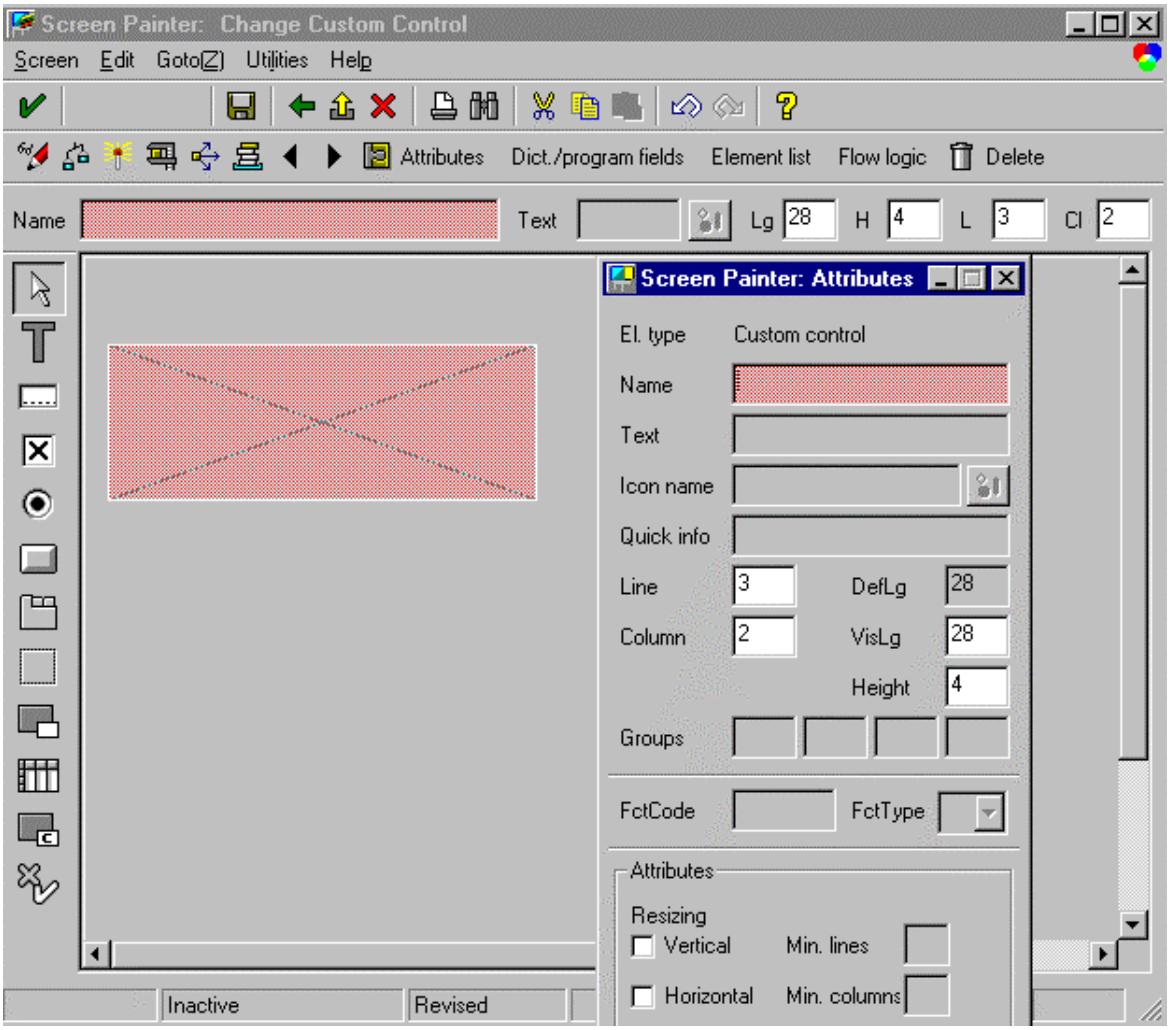
Procedure

1. Choose the *Custom Control* icon in the element palette.
   The shape of the cursor changes.
2. Use the cursor to position the container on the screen and drag it out to the required size.
3. Change the size and position of the container if required.
4. Enter the name of the container.
5. Specify the resizing attributes [Page 347].

Result

You have now created a custom container. The actual reserved area in the Screen Painter is a placeholder for the control that you will embed there. The custom container is referred to as the parent of the control or controls that are to be embedded in it.

Creating a Custom Container



See also:

Creating a Control using the SAP Picture as an Example [Ext.]

Working with Step Loops

Step loops are considerably less flexible than their replacement, table controls [Page 299].

Step Loops

You can group screen elements together into a step loop. A step loop is a repeated series of loop blocks. A loop block consists of one or more loop lines of graphical screen elements. You can define a loop block as fixed or variable.

In a fixed loop, the lower limit of the loop area always remains as originally defined. For a variable loop, the number of repetitions is adjusted dynamically in the screen program to suit the size of the current window. In order to be able to react to the variable loop size, the system always places the current number of loop blocks in the system field SY-LOOPC. If the screen includes several loop blocks, you can define only one of these as variable.

When you execute a screen with several loop blocks, the online processor runs through this "screen table" line by line.

- Do not use the steploop method to format lists. Use a report program instead.
- The step loop procedures in the alphanumeric display are different. For further information, see Creating and Editing Steploops in Alphanumeric Mode [Page 328].

Creating a Steploop

1. Open a screen in the layout editor.
2. In one or more lines, create the elements you want to repeat.
3. Select all the elements on the desired line(s) as a group.
4. Choose *Edit* → *Grouping* → *Steploop* → *Define*.

   Your element lines now make up a single steploop block. The block includes the original elements with their attributes and a predefined number of repetition blocks. Each repetition contains a copy of the first block without attributes. The repetition blocks are consecutively numbered, so that you can establish a reference to a particular line.

Editing a Step Loop

- To edit a loop block as a complete unit, click on the loop's border handles. By choosing *Edit* → *Grouping* → *Steploop*, you can manipulate the block using functions such as *Define*, *Variable* or *Fix*, or *Undefine*.
- To remove an element from a block, click on the element and choose *Delete*. If you delete all the elements, the block is deleted as well.

Working with Step Loops

- To dissolve a loop block, select it and choose *Edit* → *Steploop* → *Undefine*. The individual elements then become normal screen elements again.
- To define a steploop as fixed or variable, select a steploop and choose *Edit* → *Grouping* → *Steploop* → *Fix* or *Variable*. Recall that a variable loop is adjusted dynamically with the screen size and a fixed loop is not.

When you have completed the definition procedure, see the ABAP User's Guide [Ext.] for information on programming steploops.

Converting a Step Loop

You can convert a steploop into a table control. When converting a steploop, you have to decide whether to include headings in the conversion or not.

Converting without Headings

Select a steploop without including its headings and choose *Edit → Convert → Table Control.* The system creates a table control that is exactly the size initially occupied by the original steploop.

The first row of the table control (used for column headings) remains empty. Multiple-line steploops are linked to one table row. If the row is too long to fit into the visible area, you must scroll the table to view the row. The system calculates the table's column widths from the space between the elements in the original steploop.

Converting with Headings

Select a steploop together with its headings as a group and choose *Edit → Convert → Table Control.*

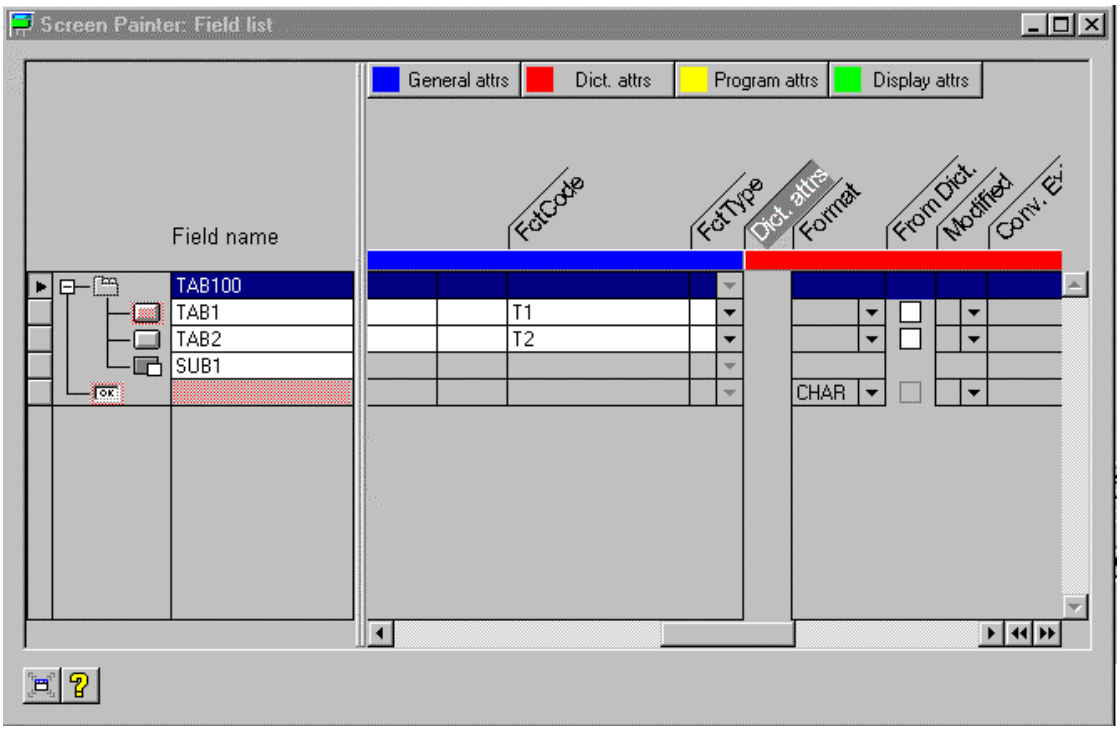
The system creates the table and fills the heading row with the headings from the steploop. The headings are inserted according to the sequence of the steploop columns and not according to their names.

The system adjusts the table control height to include the former headings.

Element List in Graphical Mode

The *Element list* provides you with an overview of all of the attributes of all of the elements on the screen. You can **display** and **change** the attributes in exactly the same way as in the attributes dialog box for a single element.
An element list is always assigned to an editor session. The way in which the attributes are displayed depends on whether you are using the Screen Painter in graphical or alphanumeric mode.

This section describes the element list in the graphical layout editor. To display the list, choose *Element list* from the editor.



If you start the element list from the initial screen of the Screen Painter, the system displays the element list in alphanumeric mode [Page 333].

Special Features

The attributes in the element list are divided into four categories:

- General
- Dictionary
- Program
- Display

The groups are marked with different colors. When you click one of the colored buttons, the corresponding group of attributes is displayed in the dialog box.

You can expand or collapse a group of attributes by clicking the plus or minus sign below the attribute header.

The second column contains a hierarchical list of screen elements. Complex elements such as tabstrip controls and table controls each have a subtree. You can expand or collapse a subtree by clicking its plus or minus sign.

The Alphanumeric Fullscreen Editor

The graphical fullscreen editor provides a user-friendly environment for designing screens on all platforms. You can start the fullscreen editor either from the initial screen of the Screen Painter or from the Repository Browser.

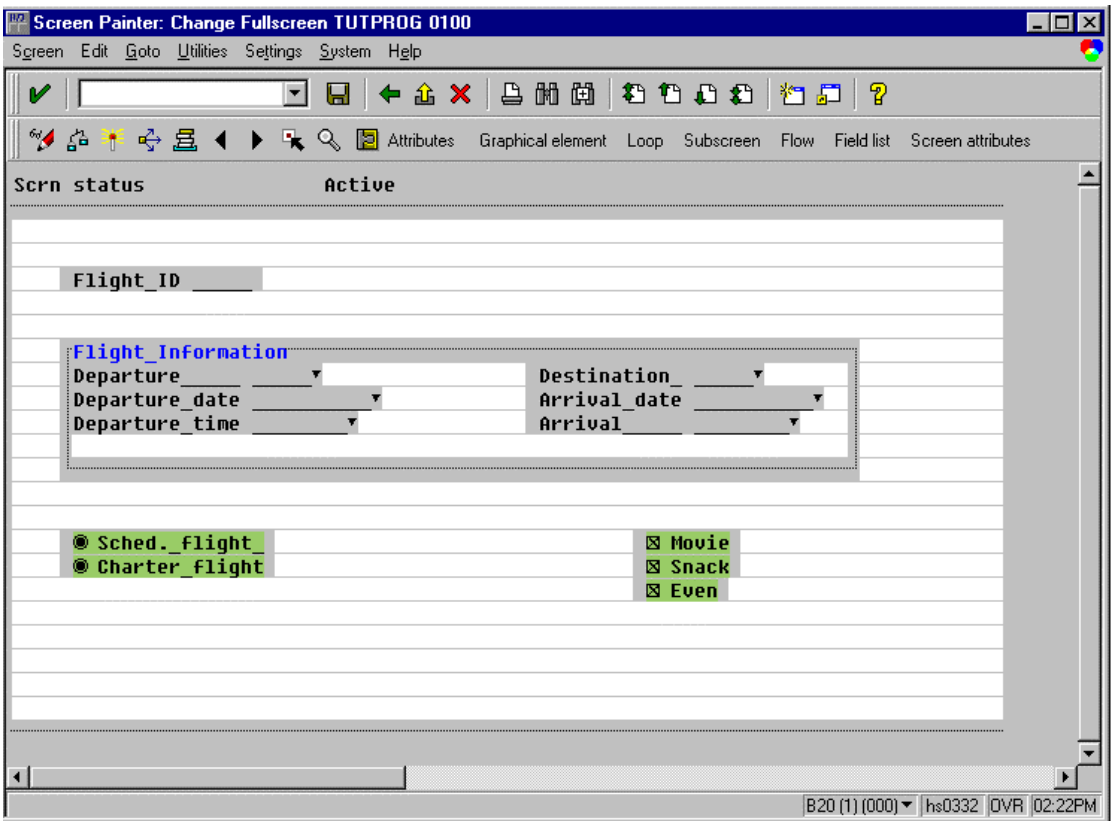
Starting the Fullscreen Editor

To start the alphanumeric fullscreen editor from the initial screen of the Screen Painter:

1. Enter a program name and a screen number.
2. Choose *Settings* → *Graph. fullscreen.*
3. If *Graphical fullscreen* is selected, deselect it and choose *Continue*.
4. Choose *Fullscreen Editor* from the list of *Components* on the initial screen.
5. Choose *Change*.

The system opens your screen in change mode in the fullscreen editor.

Example:



Differences Between the Alphanumeric and Graphical Modes

The alphanumeric fullscreen editor contains the same functions as the graphical fullscreen. The difference between the two lies in the way you create graphical elements and how they are displayed on the screen.

Representation of Graphical Elements

The system uses alphanumeric characters to display graphical elements. The system distinguishes all the graphical elements with color.

To ensure that you do not accidentally overwrite one, they are all write-protected. To change an element, you must select it and then choose an appropriate function.

See also:

Creating Screen Elements [Page 317]

Modifying Screen Elements [Page 330]

Creating Screen Elements

Unlike in the graphical fullscreen editor, you do not choose elements from the element palette. Instead, you use menus.

For a short description of all screen elements, see Screen Element Types [Page 279]

There are two ways of creating a screen element:

- You can link a new element to an existing ABAP Dictionary or program field. For details of how to do this, see Using Dictionary and Program Fields on a Screen [Page 319].
- You can also create new screen elements without reference to an ABAP Dictionary or program field by choosing *Edit → Create element*.

Procedure

The procedure differs according to the type of screen element you want to create.

Creating a Text Field, Input/Output Field, Radio Button, Checkbox, or Pushbutton

1. Place the cursor at the position where you want to insert the element. You can move the cursor using the arrow keys.
2. Choose *Edit → Create element*, followed by the appropriate element.

   The *Screen Element Attributes* dialog box appears.
3. Fill in the appropriate attribute values.

   If you are entering the name of a Dictionary field, the system prompts you for information about how you want to interpret the field. If you are creating check boxes or radio buttons, the system prompts you with two attributes screens. One for the element itself and one for the element's label.
4. Choose *Transfer*.

   The system then inserts the element at the current cursor position.

Creating a Group Box

1. Position the cursor where you want the top left-hand corner of the element to be.
2. Choose *Edit → Create element → Box*.

   The *Screen Element Attributes* dialog box appears.
3. Enter the attributes of the group box.
4. Choose *Transfer*.

   The system displays the change screen for the element.
5. Position the cursor where you want the bottom right-hand corner of the group box to be.
6. Choose End of box to set the group box on the screen.
   The system displays the group box on the screen as you specified.

Creating a Subscreen

1. Position the cursor where you want the top left-hand corner of the subscreen to be.
2. Choose *Edit* → *Create element* → *Subscreen*.

   On the screen, the system displays the area in which the subscreen can occur.
3. Position the cursor where you want the bottom right-hand corner of the subscreen area to be.
4. Choose *Area end*.

   The *Screen Element Attributes* dialog box appears.
5. Enter the attributes of the subscreen.
6. Choose *Copy*.
   The system displays the group box on the screen as you specified.

Creating a Table Control

See Creating a Table Control [Page 321].

Creating a Tabstrip Control

See Creating a Tabstrip Control [Page 324].

Creating a Steploop

See Working with Step Loops [Page 328].

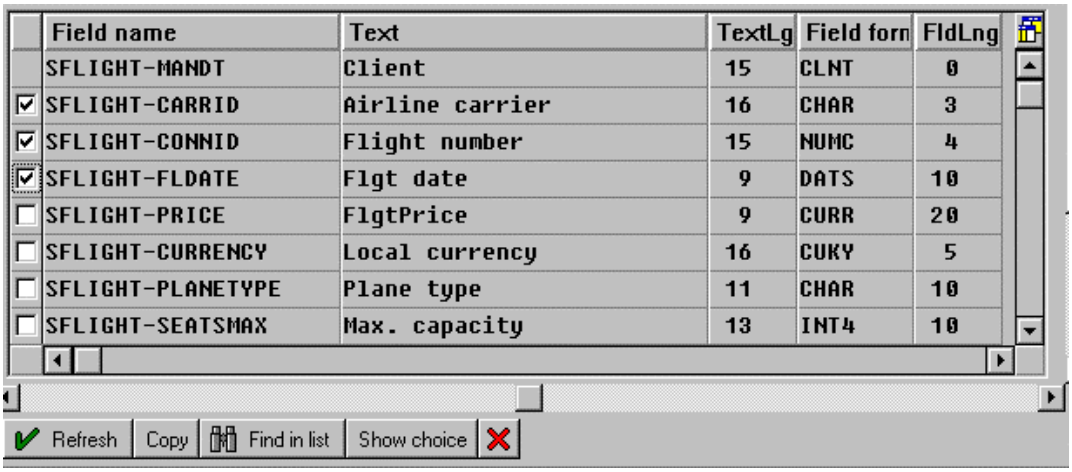
Using Dictionary and Program Fields on a Screen

You can add an element to a screen by using an existing ABAP Dictionary or program field.

Procedure

To use a selection of ABAP Dictionary fields on a screen:

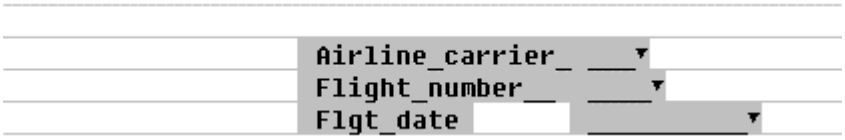
1. Position the cursor where you want to insert the fields.
2. Choose *Goto* → *Dict./Program fields*.
3. Enter a table name.
   (You must specify the table name in full.)
4. Get the table fields from the ABAP Dictionary or the program.
   To search for a table or field in the ABAP Dictionary, choose *Get from Dict*. To search for a table or field in the ABAP module pool, choose *Get from program*. The system displays the fields.
5. Select one or more fields.



6. Choose *Transfer*.
   The system returns you to the fullscreen editor.
7. Choose *Select*.

Result

The fields that you chose are displayed in a block in the fullscreen editor.



You can now work further with the fields, either individually or as a block.

See also

Modifying Screen Elements [Page 330]

Creating and Modifying Table Controls

For further information about defining and using table controls, see Table Controls [Page 299].

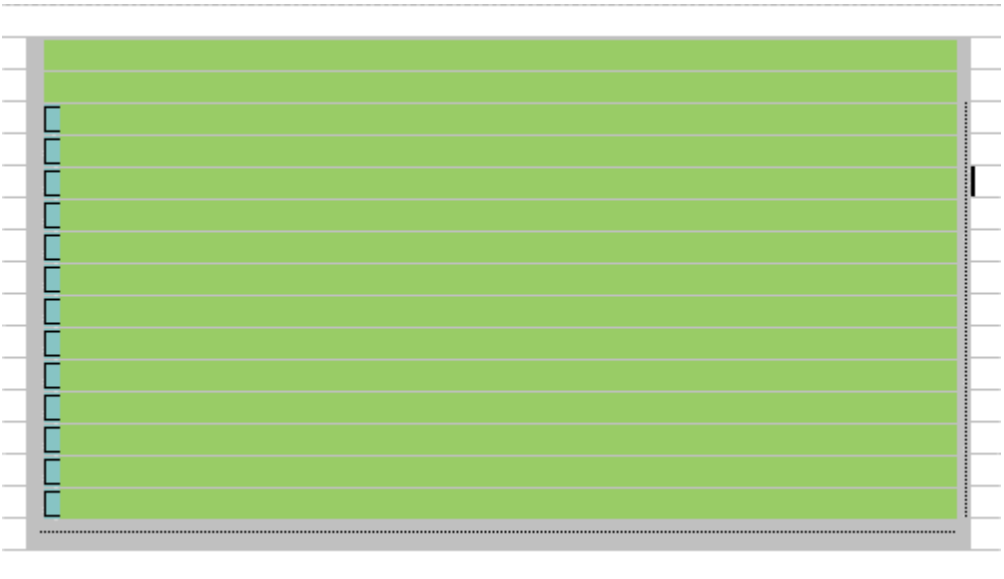
There is also a table control wizard, which you can use instead of the method described below. The wizard takes you step by step through the procedure required to create a working table control.
For further information, refer to Using the Table Control Wizard [Page 303].

Creating a Table Control

1. Position the cursor where you want the top left-hand corner of the table control to appear.
2. Choose *Edit* → *Create elements* → *Table control*.

   The system shows the maximum possible size for the table control in the fullscreen editor.
3. Position the cursor where you want the bottom right-hand corner of the subscreen area to be.
4. Choose *Select Ctrl end* to mark the end of the table control.

   The attribute dialog box appears.
5. Enter the table control attributes [Page 346].
6. Choose *Copy*.
   The table control appears in the fullscreen editor according to the size and attributes you have specified.



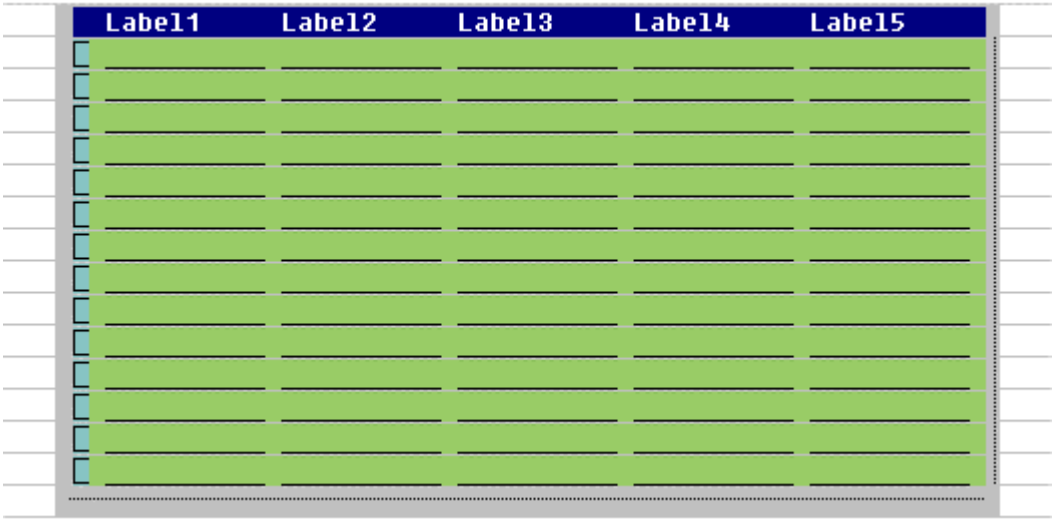
Adding Elements and Headings

The following procedure assumes that you are creating new screen elements without reference to ABAP Dictionary or program fields [Page 339]. If you are, choose *Dict./Prog fields* in step 5.

Creating and Modifying Table Controls

1. Position the cursor within the table control area.
2. Choose *Select* or *Edit* → *Table control*.
   The application toolbar now contains special functions for editing table controls.
3. Choose *Ctrl elements*.
   A dialog box appears, in which you can enter the definition of the table columns and their headings.
4. Place the cursor in the input field for the relevant column.
5. Choose *Create element.*
   Another dialog box appears, in which you can set an element type for the table column. You can create a text field, an input/output field, a radio button, a checkbox, or a pushbutton.
6. Select a field type.
   The *Screen Element Attributes* dialog box appears.
7. Enter the attributes and choose *Copy*.
   If you are defining an input/output field, you must specify the defined length as well as the field name.
8. Enter the column heading and press **ENTER**.
9. In the next dialog box, enter a field type and choose *Continue*.
10. Enter the field type attributes and choose *Copy*.
11. Repeat steps 4-10 for each additional column.
12. Choose *Copy*.

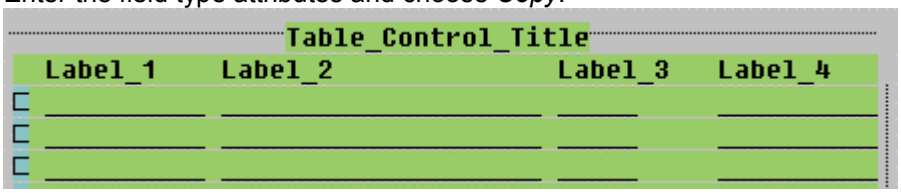
The table control, with its elements and column headings, is now displayed in the fullscreen editor.



Adding a Table Title

1. Open the attribute window in the table control.
2. Select *With title*.

3. Enter a field name for the title and choose *Copy*.
4. In the next dialog box, enter a field type and choose *Continue*.
5. Enter the field type attributes and choose *Copy*.



Editing Functions for Table Controls

function	Explanation
Ctrl attributes	Changes table control attributes.
Ctrl Elements	Changes table control columns. (Adding, changing, and moving columns.)
Select Ctrl end	Moves the bottom right-hand corner of the table control.
Deselect control	Ends an editing session for the table control.
Convert TC	Converts the table control into a fixed or variable step loop [Page 328].
Move TC	Moves a table control to a new position.
Dissolve TC	Dissolves the table control into individual elements. However, the system can only place the elements in the original table area. If dissolving a table control is likely to produce an unsatisfactory result, the system asks you whether you really want to proceed.
Delete Ctrl	Deletes the table control.

Creating a Tabstrip Control

For further information about defining and using tabstrip controls, see Tabstrip Controls [Page 291].

There is also a tabstrip control wizard, which you can use instead of the method described below. The wizard takes you step by step through the procedure required to create a working tabstrip control.
For further information, refer to Using the Tabstrip Control Wizard [Page 297].

Procedure

Creating a tabstrip control in alphanumeric mode has two steps:

- Create a tabstrip area.
- Define tabstrip elements (pushbuttons and subscreen areas).

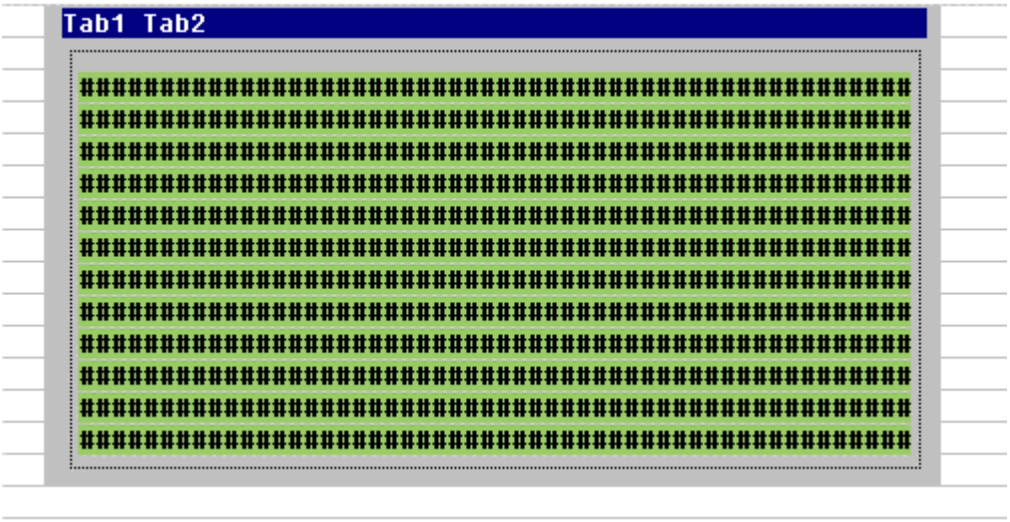
Defining the Tabstrip Area.

1. Position the cursor where you want the top left-hand corner of the tabstrip control to appear.
2. Choose *Edit* → *Create elements* → *Tabstrip control*.

   The system displays the maximum possible size of the tabstrip control in the fullscreen editor.
3. Position the cursor where you want the bottom right-hand corner of the tabstrip control to be.
4. Choose *Select Ctrl end* to mark the end of the tabstrip control.

   The attribute dialog box appears.
5. Enter the tabstrip control attributes [Page 345].
6. Choose *Transfer*.
   The tabstrip control appears in the fullscreen editor according to the size and attributes you

have specified. By default new tabstrip controls come with two tab titles.



Defining Tabstrip Control Elements

You add further tab titles using pushbuttons. A subscreen area must be assigned to each tab page.

1. Select the tabstrip control.
   The tabstrip control is now ready for editing.
2. Choose *Edit* → *Ctrl elements*.
   The tabstrip element dialog box appears.
3. Under *tab title*, enter the field name for the new pushbutton that you want to create.
4. Choose *Attributes*, or press **ENTER**.
5. Enter the attributes and choose *Copy*.
6. Enter the function code and function type of the pushbutton.
7. Repeat steps 3-6 for each additional pushbutton.
8. Under *Subscreens*, enter the field name of the subscreen area and confirm its attributes.
9. Under Reference subscreen, assign the appropriate subscreen areas to the pushbuttons.
10. Choose *Transfer*.

Result

You have now created a tabstrip control and assigned extra tab titles and subscreen areas to it. For an impression of how the tabstrip looks at runtime, you can simulate the screen.

(See Testing Screens [Page 350]**.)**

Creating an SAP Custom Container

The “custom control” is actually a container for a control, which allows the screen framework to support resizing and compression of controls.

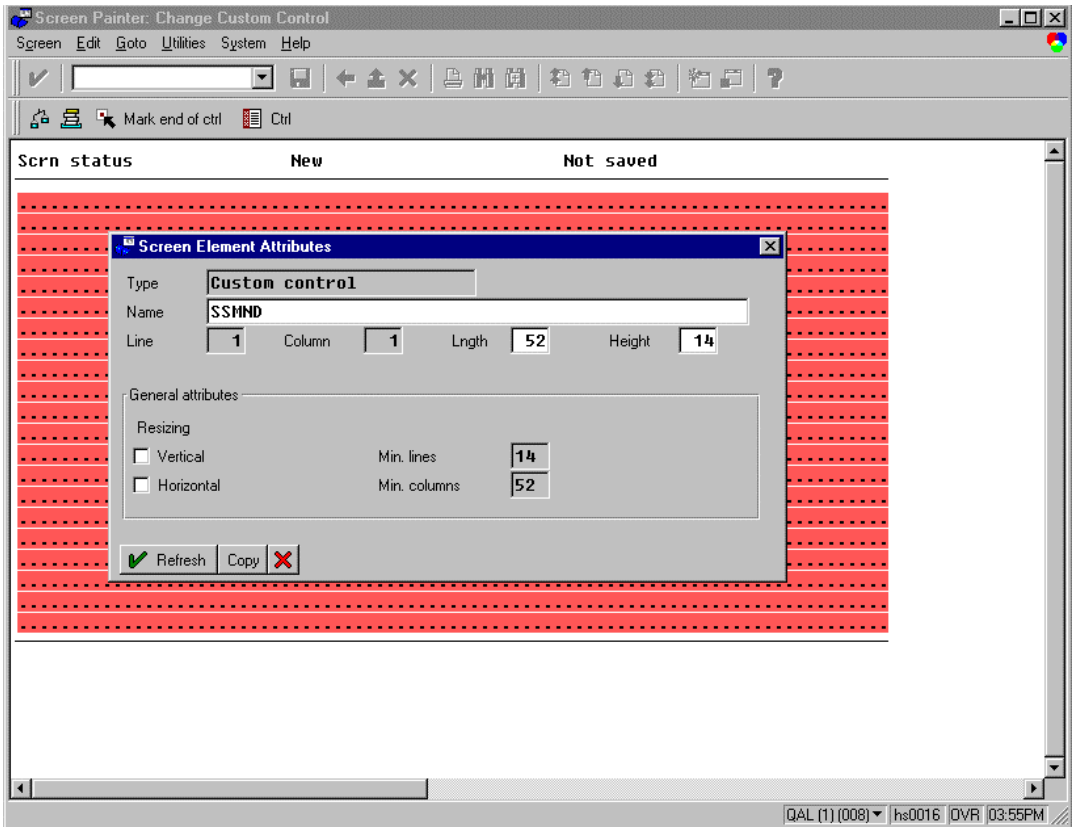
For information about control programming, refer to the SAP Control Framework [Ext.] documentation.

Requirements

You must have opened the layout editor of the alphanumeric Screen Painter in change mode.

Procedure

1. Place the cursor at the position you want the top left-hand corner of the container to occupy.
2. Choose *Edit* → *Create element* → *Custom control*.
   The system displays the change screen for the container in its maximum dimensions.
3. Position the cursor at the position you want the bottom right-hand corner of the container to occupy.
4. Choose *Mark ctrl end* to fix this position.
   The attribute dialog box for custom controls appears.
5. Enter the custom control attributes [Page 347].
6. Choose *Copy*.
   The system displays the control container with the dimensions that you specified in the dialog box.



Functions for Editing the Container

Function	Description
Ctrl Attributes	Changes the attributes of the custom control
Mark ctrl end	Changes the bottom right-hand corner of the control
Deselect ctrl	Deselects the control
Delete ctrl	Deletes the control
Move ctrl	Moves the control to a new position

See also

Creating a Control Using the SAP Picture as an Example [Ext.]

Creating and Modifying Step Loops

The procedure for creating step loops differs depending on whether you are using the alphanumeric or the graphical Screen Painter.

Creating a Step Loop

1. Create the elements you want in the loop.

   The elements can appear on one line or on several lines. You can separate the elements by blank lines.

2. Place your cursor on the element that will make the upper-left corner of the loop.
3. Choose *Loop*

   The Screen Painter changes to the *Change Loop Definition* screen. You can create several separate loop blocks on one screen, but nesting is not allowed.

4. Place your cursor on the lower right corner of the final element in the loop.
5. Choose *Loop block end.*

   The system automatically generates a fixed loop block with two repetitions. You can now either edit the step loop further, or choose *Back* to return to the normal fullscreen editor.

6. Save your changes.

Step Loop Editing Functions

To edit a step loop, you must be in the *Change Loop Definition* screen. To edit an existing loop block, place the cursor on the block and choose *Loop*. The following functions are available for editing a step loop:

function	Explanation
End line of loop	Defines the length of the loop area. The system automatically repeats the block within the defined area.
Variable loop	Defines a variable loop. With a variable loop, the system adjusts the number of repetitions dynamically in the screen program to suit the size of the current window. If the screen includes several loop blocks, you can define only one of these as variable. The maximum number of lines is 200. In order to be able to react to the variable loop size, the system always places the current number of loop blocks in the system field SY-LOOPC. With a fixed loop, the lower limit of the loop area always remains as originally defined.
Dissolve	Dissolves the loop by placing its fields in the screen.
Move loop	Moves the loop to a new location.

You can make changes only in the definition block. Changes in the definition block automatically affect the repetition blocks. In the actual definition block, you can change the attributes of

elements as usual. You can also define new elements or delete elements. However, you cannot insert or delete lines within the loop block.

If you choose either *Edit* → *Select* or *Edit* → *Temporary storage* → *Copy to temp.storage*, you can work in the same way as for fields not in a loop. However, you can select only elements in the loop definition block. The selected block cannot exceed the loop limits.

You can copy elements from outside to the loop block or vice versa. In both cases, the system adapts the element attributes accordingly. When you copy to a loop block, the elements are automatically added to the corresponding repetition elements. When you copy from a loop block, the same repetition elements are deleted.

Modifying Screen Elements

To edit a graphical element in alphanumeric mode you must first open the element's change screen. To open a change screen, place your cursor on an element and choose *Edit* → *Graphical element*. From a change screen, you can:

- Combine a series of logically associated radio buttons into a radio-button group.
- Dissolve a radio-button group into a series of individual radio buttons.
- Convert radio buttons to check boxes and vice versa.
- Convert elements

See Converting elements [Page 332].

- Move an element or a radio button group.
- Change the size of group boxes and subscreens.

Choose *Back* to leave the edit mode of a graphical element and return to the normal alphanumeric fullscreen.

Selecting

To select an element, double-click it. Alternatively, you can place the cursor on the element and choose *Select*. When you select an element, the system displays the *Change Select* screen. To return to the normal fullscreen editor screen, choose *Back*

If you select elements joined in a group, the entire group is selected. To select a block of elements do the following:

1. Select the first element in the block.

   The system places you in the *Change Select* screen.
2. Place your cursor on the element at the end of the block.
3. Choose *Mark end of block*.

   The system highlights the entire block.

Moving

To move a selected element or block of elements, do the following:

1. Select the element.

   The system places you in the *Change Select* screen.
2. Select a point on the screen where you want to move the element to.

   If you are moving a block, the new position can be within the old block. The system simply shifts the entire block to the new position.
3. Choose *Move*.

Copying and Inserting Elements

You can copy a selected element into the clipboard and later insert the copy at another point on a screen. To copy an element, select it and choose *Copy to t. stor.* To insert the clipboard contents, place the cursor at the desired destination and choose *Edit → Clipboard → Insert frm t. stor.*

To cut and paste a selection of the screen, do the following:

1. Select the element.
2. Copy the element to the clipboard.
3. Delete the original element.
4. Position the cursor at a new location.

   To paste the clipboard's contents into another screen, you must first call the other screen by choosing *Screen → Open*. Choose *Screen → Other screen*.
5. Insert the copy from the clipboard.

The advantage of this procedure is that you do not have to specify element attributes; the clipboard retains the original attributes. The clipboard is maintained only for a single Screen Painter session. To view the clipboards contents, choose *Edit → Clipboard → Display clipboard*.

Loop information is not retained. If you need loop blocks after inserting, you must recreate them.

Converting Elements

Just as in graphical mode, you can convert elements in the alphanumeric mode. The procedure is slightly different in alphanumeric mode. To convert an element, do the following:

1. Place your cursor on an element.
2. Choose *Graphical element*.

   The system displays the change screen for the element.
3. Choose *Edit* → *Convert* and a new element type.

   Combine the radio buttons to form a radio-button group.

Converting Elements into Pushbuttons

Before converting pushbuttons, make sure that you use only keywords or output fields. You cannot convert input fields into buttons. After you convert an element, you can assign it a function code using one of *Field list* screen or *Attributes for 1 field* dialog.

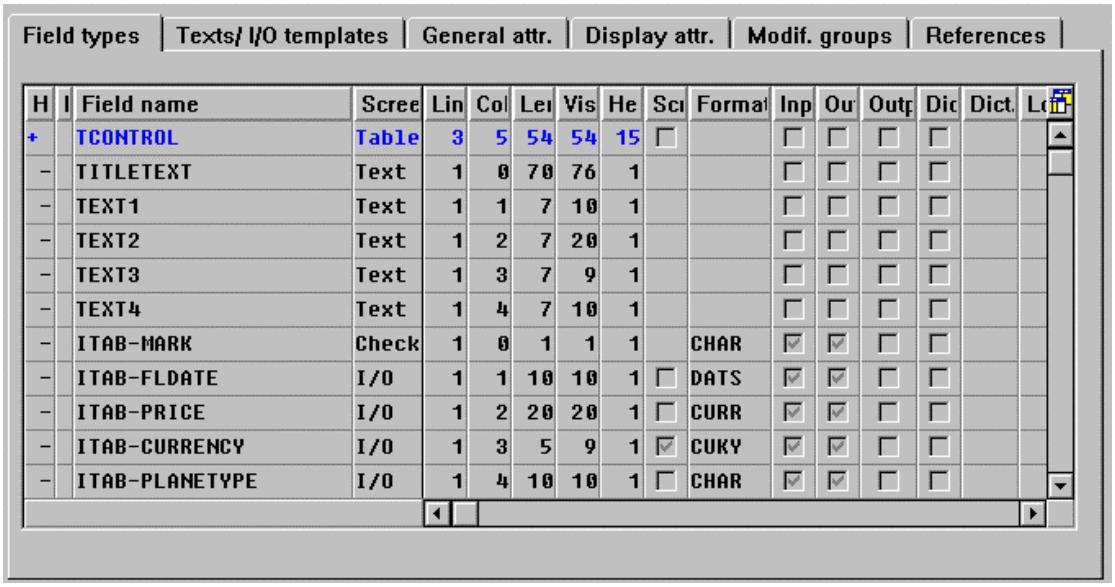
Converting Tables into Step Loops

If you want to convert a table control into a variable or fix step loop, the system again tries to use the columns and lines given. If this is impossible, the system creates multiple-line step loops. The system always creates as many repetition lines as possible, with two as a minimum.

Using the Field List View

You can use the field list to display or change the attributes of any element on the screen.

Use the *Goto* menu to display the *Field List* screen or choose *Field list* from the fullscreen editor. Example:



Components of the User Interface

View	Description
Field types	Contains attributes that describe the field. This includes attributes specific to the visual element and attributes connected to the underlying field.
Texts and I/O templates	Contains attributes related to the text and icons that are associated with an element.
General attr.	Contains attributes related to how the user interacts with the field. This screen includes attributes like whether a field is required or not.
Display attr.	Contains display attributes such as whether a field is visible or right-justified.
Modif. groups	Contains modification group attributes.
References	Contains the attributes for search helps and references.

The first column in each field list view displays the hierarchy of the element. For complex elements, that is, step loops, table controls, and subscreens, this column contains a + (plus sign). For elements contained in the directly preceding complex elements, this column displays a - (minus sign).

Defining the Element Attributes

After adding screen elements to your screen, you must provide attributes for each element. The system sets some of these attributes automatically. You can set the attributes either in the *Attributes* window or in the element list [Page 312].

The Attributes Dialog Box

To display the *Attributes* dialog from the fullscreen editor, choose *Attributes* or double-click an element. The *Attributes* dialog lists the attributes for a specific element.

These can be divided into

- General attributes [Page 336]
- Dictionary attributes [Page 339]
- Program attributes [Page 341]
- Display attributes [Page 343].

When you use table controls, tabstrip controls, or other controls, you should also remember the following sets of attributes:

- Table control attributes [Page 346]
- Tabstrip control attributes [Page 345]
- Custom Control Attributes [Page 347]

Any changes you make in the element attributes window using radio buttons become active at once. Text changes become active when the text editor ceases to be the active field (for example, when you click on another field).

General Attributes

For other attributes, refer to Working with Element Attributes [Page 335].

The general attributes comprise the following:

Attribute	Description and Ergonomic Guidelines
Field type	Identifies a screen element type (for example, keyword or group box).
Name	Identifies an element. You use this name to address a field from a module pool. All input/output fields require a name. Text fields require a field name only if they are translated into another language. An input/output field and the keyword that labels it can share the same field name. Otherwise, field names must be unique. Field names can not exceed 40 characters and must begin with a letter or an * (asterisk). The only special characters allowed are # (pound sign), / (forward slash), - (dash), _ (underscore) and $ (dollar sign). If you enter a new field name or change an existing name so that it contains a - (dash), the system checks the ABAP Dictionary for information associated with the field. On the *Field attributes* screen, the system automatically copies any information into the field's attributes. On the *Attributes* screen, the system asks whether you want to use the Dictionary field and its attributes. **Note:** From Release 4.6C, the element name of an input/output field can refer to the ABAP data type **STRING**. Remember, though, that the maximum length of an input/output field on the screen is restricted to 132 characters. You can only define input/output fields in the Screen Painter that fit on a **single line**. Characters in a string that are longer than this are truncated. The element in the ABAP program or ABAP Dictionary can, however, be of any length. There are two new field formats in the ABAP Dictionary - STRG and RSTR. For further information, refer to Field Formats [Page 348].
Text	Specifies a field's text. If you want to use an icon instead of text, leave this field blank.
Dropdown	This attribute only applies to input/output fields. If you choose List box, you can display a list of entries for the field, from which the user can select one. The width of the list is determined by the VisLg (visible length) fields. The height is automatically set by the system. If the list box contains a large number of entries, vertical scrollbars automatically appear. You assign a value list to the output field using the ValueID. The default value of this is the name of the input/output field (*Name* attribute). How the value list is displayed depends on the entry in the Input help attribute. For further information, refer to the Input help section of Program Attributes [Page 341].

With icon	Sets an icon value. Only use this attribute for output fields. For a list of all existing icons including length and quick info texts, refer to the table ICON (or Transaction ICON).
Icon name	Identifies an icon for a keyword. If you specify an icon that is not allowed for a field, the system returns an error message. To display only the icon without a label, leave the *Field text* empty. For templates, you set the *Icon Name* attribute at runtime. **Usability guideline**: Icons do not need an explanatory text if the meaning of the icon describes the element exactly. The meaning of the icon is always available in the quick info text.
Quick info	Identifies an icon's info text. This text is visible when the user holds the cursor over an icon. The ICON table defines the default *Quick Info* value for each icon. To change this default text (or leave it blank), use the *Quick Info* field. **Usability guideline**: All icons need a quick info text, since no icon is self-explanatory.
Scrollable	Sets the scroll function. Use this attribute if the *DefLg* value is greater than the *VisLg* value. When *Scrollable* is set, the system activates scrolling for the field. If you use Dictionary or program fields that are too long for your screen, do not forget to set *Scrollable* and to reduce the field's visible length as needed.
Line	Specifies the line where the screen element appears. The system sets this attribute automatically.
Column	Identifies the column where the element begins. The system sets this attribute automatically.
Height	Specifies the height of an element in lines. Text labels and entry elements always have a height of 1.
DefLg	Defined length. Identifies the actual length of the field in the Dictionary or your program.
VisLg	The length of an element displayed on the screen. Set this attribute if you want the screen element to have a length different from its defined length. You must set the attribute *Scrollable* if you want to make visible length less than the defined length. You can only change the visible length for input/output templates or for elements in table columns. In the latter case, *VisLg* may exceed *DefLg*. For all other elements (except elements containing icons and input/output fields with the *Dropdown* attribute), the visible length is the same as the defined length and cannot be changed.
Groups	Modification groups This attribute allows you to update several fields at once. You can assign each field to up to four modification groups. To assign a field to a modification group, enter the three-character group name in the appropriate column.
FctCode	Function code: This attribute is only for pushbuttons and input/output fields with the *Dropdown List box* attribute. If the user presses a pushbutton, the system sets the command field to the 20-character code you enter here. This value is not checked against values you specify in the Menu Painter.

General Attributes

FctType	Specifies the event at which the field is processed (for example, AT EXIT-COMMAND).

Dictionary Attributes

For further screen element attributes, refer to General Attributes [Page 336].

The Dictionary attributes comprise the following:

Attribute	Description and Usability Guidelines
Format	Data type. The data type determines what checks (valid date, numeric values) the system should perform on field input and how to convert the field for input/output. This field is always blank for keywords. For a list of the available data types, see Defining the Field Format [Page 348].
From Dict.	Specifies the current ABAP Dictionary reference. The system sets this attribute if you created this field by copying it from the Dictionary. If you assign a name to a new element or change the name of an existing element so that the name contains a - (dash), the system checks whether the field exists in the Dictionary. If the name exists, the systems asks if you want to refer to the Dictionary field or not.
Modified	The system sets this attribute if it detects a difference between the Dictionary definition for the field and the way it is used in the screen. Set this attribute if want to use a keyword that diverges from its Dictionary definition.
Conv.Exit	If you want the system to use a non-standard conversion routine for the conversion of field input, specify a four-character code here. There are two supported conversion routines: CONVERSION_EXIT_**<name>**_INPUT CONVERSION_EXIT_**<name>**_OUTPUT See the SAPCNVE program documentation for more information.
Search help	Allows the user to specify a search help as input to the element. Enter a four-character search help file name or the name of a field that contains the file name. In the latter case, prefix your input with a : (colon). For more information on search helps, refer to the ABAP Dictionary [Ext.] documentation.
Reference Field	For **tabstrip controls**: Establishes the link between a tab title and a subscreen area. For **currencies** and **quantities**: Specifies a currency or unit key. This attribute is valid only for fields of type CURR (currency) or QUAN (quantity). If the screen element contains a currency and the field type is CURR, you must enter the currency key field belonging to the currency (CUKY). If the screen element contains a quantity and the field type is QUAN, you must enter here the unit key field belonging to the quantity (UNIT). If the screen element is from the Dictionary, the system takes over the reference field without allowing any changes.
Parameter ID	ID for a SET/GET parameter (up to 20 characters long). This attribute is used with either the *SET parameter* or *GET parameter* attribute.

Dictionary Attributes

SET parameter *GET parameter*	Set and display default in an element. If you choose *Set Parameter*, the system stores the value entered by the user in the relevant *Parameter ID* parameter. If you choose *Get Parameter*, the system displays the value in *Parameter ID* in the element instead of the initial value.
Foreign key check	Determines whether the system performs a foreign-key check for the field. The field's definition in the ABAP Dictionary defines the foreign-key check.
Upper/Lower case	Set this attribute if your program handles the user's input as a literal. If not, the input is converted to all upper-case. Usability note: Set this attribute wherever possible. Mixed case input is easier to read than uppercase.

Program Attributes

For further screen element attributes, refer to General Attributes [Page 336].

The program attributes comprise the following:

Attribute	Description and Usability Guidelines
Input field	Defines an element as an input field. If the output field attribute is not set, the data in this element is processed during the transaction, but not displayed. However, you can set both, as is the default when creating a new template.
Output field	Defines an element as an output field. Choose this attribute for text templates that the program can use to display output. You cannot input data in these elements unless the *input field* attribute has also been set.
Output field only	Prevents display-only elements from being changed into input elements at runtime. This attribute is useful when a program attempts to set all input elements globally back to "ready for input". In this case, the input templates and the input/output templates return to input readiness, but the elements that are marked *Output only* are not affected by such a change. (For example, in the Workbench, the *Display<->Change* function makes use of this attribute.) **Usability note**: Use this attribute for dynamically-set field names. The names are then displayed two-dimensionally and in a proportional font in the same way as static field labels.
Required field	If you set this attribute, the user must enter a value in the field. Required fields appear on the screen containing a question mark (?).
PossEntry	This attribute can only be set (and is only displayed) for input/output fields with the attribute Dropdown and the entry List box. You use it to determine how and when the value list for a dropdown list box is generated. There are two possible entries: • *Space* The system provides standard help. The help processor is started in the PBO and fills the value table automatically before sending it to the Value Request Manager. The sources are domain fixed values, value tables, and search helps. If the input/output field is linked to a PROCESS ON VALUE REQUEST module, this takes priority over the automatic process described above. • *A from program* The application itself determines the values in a PBO module and passes the table and ValueID (*Name* attribute) to the Value Request Manager using the function module VRM_SET_VALUES.

Program Attributes

Poss Entr. button	Specifies whether a possible entries pushbutton should appear beside the element. The attribute does not appear for *listbox* elements with the *dropdown* attribute. If the element has possible entries, you can enforce or suppress the display of the entries button using this field. The system sets this field automatically for fields that specify foreign key checking or value lists in the ABAP Dictionary. This field is also set automatically for fields of type TIMS or DATS. **Usability note**: You should only change the value of this field if the value set automatically is incorrect for technical reasons.
Right-justified	Right justifies numerical fields in an element. You can also display keywords in this way (for example, when defining headers).
With leading zeros	Left justifies values in numerical fields with leading zeroes.
**-entry*	Allows the user to enter an * (asterisk) in the first position of the element. The system ignores the asterisk and transports input starting from the second position. Transport is determined by the conversion guidelines in the field format. However, the first-character asterisk triggers a flow logic module you declare with: `FIELD... MODULE... ON *-INPUT.`
Without reset	Prevents the reset character (!) from being used to delete input from an SAP field.
Without template	Prevents special characters from being treated differently. If the user enters special characters as part of the input, they are transferred to the screen as regular text. If you set this attribute, you cannot set the *Req.entry* attribute.

Display Attributes

For further information about screen element attributes, refer to General Attributes [Page 336].

The display attributes comprise the following:

Attribute	Description and Usability Guidelines
Fixed font	Displays input/output fields and text fields in a nonproportional) font. To use *Fixed font* for output templates, you must also have defined the element as an output only field. **Usability note**: Do **not** set this attribute, since fixed fonts are more difficult to read than proportional fonts.
Bright	Highlights an element. **Usability note**: Do not set this attribute for more than 10% of the information on a screen.
Invisible	Set this attribute if you want the element to be invisible.
2-dimensional	Displays elements without the three-dimensional shading that normally appears around the element border. This field is automatically set when you use an icon. **Usability note**: Do **not** set this attribute, since field display is automatically optimized by the system.
As label left	This attribute is used for text fields and input/output fields that are only used for display purposes (*Input field* attribute not set). If this attribute is set, the text field is linked to the screen element on its right in the same line. At runtime, the text field appears as a label for that screen element, that is, it appears to the left of the screen element. If a field has an ABAP Dictionary reference, this attribute is preset. .
As label right	This attribute is used for text fields and input/output fields that are only used for display purposes (*Input field* attribute not set). If this attribute is set, the text field is linked to the screen element on its left in the same line. At runtime, the text field appears as a label for that screen element, that is, it appears to the right of the screen element.
Double-Click sensitive	Makes a screen element double-click sensitive (hotspot). You can only set this attribute for text fields and input/output fields. If you set this attribute, **double-clicking** the element at runtime triggers an action.

Display Attributes

The *Reverse video*, *Blinking* and *Underlined* attributes, as well as an individual color scheme for each element (all available in R/2), are not supported in R/3. The color scheme is now defined for the system as a whole.

Tabstrip Control Attributes

For further screen element attributes, refer to General Attributes [Page 336].

Attribute	Description and Usability Guidelines
Resizing *Vertical/horizontal*	Indicates that the tabstrip changes its size when you change the height (vertical resizing) or width (horizontal resizing) of the window. Set this attribute if you want the tabstrip control always to appear in proportion to the screen.
Min. lines	Specifies the minimum number of lines that can be displayed when you resize vertically. It is the height of the largest subscreen area in the tabstrip control.
Min. Columns	The minimum number of columns that can be displayed when you resize horizontally. It is the width of the largest subscreen area in the tabstrip control **Usability note**: By choosing the values carefully, you can avoid the bottom edge of the tabstrip control moving up or down when you choose a new tab page.

If the screen contains a variable step loop, vertical resizing is no longer available.

Table Control Attributes

For further screen element attributes, refer to General Attributes [Page 336].

The following table control attributes are available:

Attribute	Description and Usability Guidelines
Tab title	Number of tab titles in the tabstrip control. The field texts <field1> to <field#> are preassigned.
Table type	Indicates whether the table control is for data entry or selection.
W/ ColHe	This attribute allows you to create column headings as text fields or input/output fields.
W/ Title	This attribute allows you to create a table title as a text field or an input/output field. You must create the title separately.
Configbl	Allows you to save the current setting of the table control attributes to a file at runtime. The system retrieves the settings from this file whenever the table is displayed. **Usability note**: Switch off this attribute if there is nothing to configure (for example, when the table consists of a single column that is always displayed in full) or if the table control is modified in the program.
Resizing	Indicates a table supports vertical and/or horizontal resizing. Set this if you want the table to change with a window's size. For further information, refer to Tabstrip Control Attributes [Page 345]
Separators	Inserts vertical separator lines between the table fields (columns) and horizontal lines between the table rows. **Usability note**: Use horizontal separators with very wide table controls.
Line sel.	Allows line selection. Choose *None*, *Single*, or *Multiple*.
Column sel.	Allows column selection. Choose *None*, *Single*, or *Multiple*.
w/ SelColumn	Specifies whether a line selection column appears with the table. The column is stored internally as a checkbox. For this reason, the selection column requires a name. Enter a name in the space provided.
Fixed Columns	Excludes one or more columns from horizontal scrolling. Enter the number of columns from the left that you want to fix.

Custom Container Attributes

For further screen element attributes, refer to General Attributes [Page 336].

Attribute	Description and Usability Guidelines
Resizing *Vertical/horizontal*	This attribute determines how the custom control behaves when you change the height (**vertical resizing**) or width (**horizontal resizing**) of the window. Set this attribute if you want the custom control always to be displayed in the same proportions to the screen size.
Min. lines	The minimum number of lines of the control that can be displayed when you resize vertically.
Min. columns	The minimum number of columns of the control that can be displayed when you resize horizontally.

Choosing Field Formats

The field format determines how you edit the element attributes. A screen element can have one of the following formats:

Format	Description
ACCP	Posting period Format: YYYYMM. Internal format: C(6)
CHAR	Character Character string (in ASCII or EBCDIC format). Internal format: C(n)
CUKY	Currency key Internal format: C(5)
CURR	Currency field Corresponds to an amount field DEC, points to a field of type CUKY. Internal format: P
DATS	Date Output determined at run-time. Internal format: D(8)
DEC	Calculation field Calculation or amount field with decimal places and, if required, decimal point and plus or minus sign. Internal format: P
INT1	1-byte integer Internal format: X(1)
INT2	2-byte integer Internal format: X(2)
INT4	4-byte integer Internal format: X(4)
LANG	Language key Internal format: C(1)
NUMC	Numeric field Field of any length comprising only numeric characters (for keys, not for calculation). Internal format: N(n)

PREC	Accuracy Accuracy of a QUAN field. Internal format: X(2)
QUAN	Quantity Points to a field in UNIT format and to a field in PREC format, which specifies accuracy. Internal format: P
RAW	Hexadecimal representation of the internal value (bit sequence) of any field.
STRG	String with variable length
RSTR	Byte sequence with variable length in hexadecimal format
TIMS	Time in format HHMMSS. Internal format: T(6) If the user enters blanks or the reset character (!), the system returns blanks. The value must be numeric. The numbers on the left must not exceed 240000. The minutes and seconds should not be greater than 59. If you want to prevent use of blanks, you must prohibit reset. To do this, choose the attribute *No reset*.
UNIT	Unit key Field with unit key. Internal format: C(n)

Testing Screens

Testing allows you to check a simulation of the screen as it will appear at runtime. If you have already programmed the flow logic, you can choose whether to simulate the screen with or without it.

Procedure

1. Choose *Screen* → *Test...*
   The system displays a dialog box for the runtime simulation. :
2. If necessary, change the window coordinates. (These determine the size of the simulation.)
3. Set the scope of the simulation.
   If you want to include the flow logic in the simulation, select *Complete flow logic*. Otherwise, the flow logic is not included, and you can only test the screen layout.
4. Choose *Continue*.
   The system simulates the screen.

Checking Screens

In order to check a screen's syntax, data consistency, or layout, you use the *Check* function.

Syntax Checks

Choose *Screen* → *Check* → *Syntax* to check the syntax within the Screen Painter (attributes and flow logic). If an error occurs, the system displays the error type in the status line and positions the cursor at the relevant location. If the syntax check runs without error, the system returns a success message. You can then *Save* or *Generate* your screen.

Screen Consistency

Choose *Screen* → *Check* → *Consistency* to check data consistency between a module pool, the ABAP Dictionary, and a screen. The system checks whether the flow logic contains an object (field or module) or the field list contains a field that is not defined in the ABAP Dictionary or the module pool.

The *Consistency* function also checks the consistency of any screen attributes specified in the fields *Next screen* and *Cursor position*. In particular, it checks whether the next screen that you have specified actually exists, and whether the field specified as the cursor position exists.

The Consistency function also checks the search help, reference field, and other attributes. In this way, you can detect errors in advance and eliminate them. The system displays the errors in a hit list.

Violations of SAP Standards

If you choose *Screen* → *Check* → *Layout,* the system checks whether the layout of your screen (in the fullscreen editor) corresponds to the SAP ergonomic standards. Some of the checks include:

- whether the fields are aligned and positioned correctly on the screen
- whether boxes are positioned correctly
- how often you use display attributes

After a check, the system displays a hit list sorted by priority. For each error, the system offers a solution. However, these errors do not influence the processing of the screen. You can generate a screen regardless of layout errors.

Saving, Activating, and Deleting Screens

Saving

Each time you make changes to a screen, you should save it. To save the screen, choose *Save*. The system saves all components attached to the screen (screen attributes, layout, element list, screen flow logic).

Activating

You can only call a screen in a transaction if a runtime version already exists. You create a runtime version of a screen by activating it. To do this, choose *Screen* → *Activate* (or the *Activate* icon). Whenever you activate a screen, it is automatically saved as well. After activation, the status line contains the message:

Screen saved and activated.

You should check a screen for consistency before you activate it. Sometimes you might want to activate a screen even though it contains inconsistencies (for example, for testing purposes). If these inconsistencies are severe or the screen's flow logic contains severe syntax errors, the system will not create an active version of the screen.

Deleting

Before you delete a screen, you should check its where-used list.

To delete the screen, select it and choose *Screen* → *Delete*. The system deletes everything associated with the screen, including its flow logic.

Menu Painter

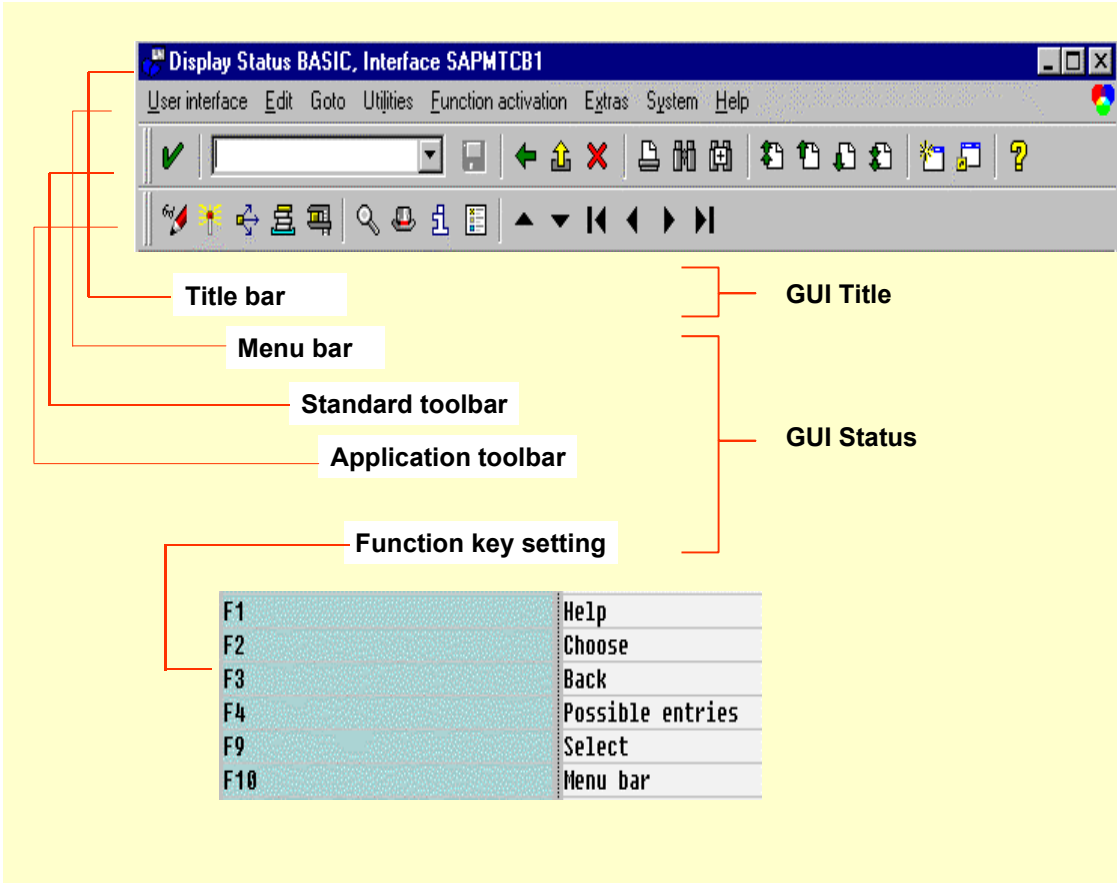
The Menu Painter is a tool with which you design user interfaces for your ABAP programs. This section tells you how to create and use the interface, and how to define the functions that you use within it.

If you want to use the Menu Painter in conjunction with the Modification Assistant, refer to Modifications in the Menu Painter [Ext.].

The Menu Painter: Introduction

ABAP programs contain a wide variety of functions, which fall into different categories within the user interface. It is important for users to be able to differentiate between these categories, and to choose the right function easily. In the R/3 System, you arrange functions using the Menu Painter.
An instance of the user interface, consisting of a menu bar, a standard toolbar, an application toolbar, and a function key setting, is called a **GUI status**. The **GUI status** and **GUI title** defines how the user interface will look and behave in an ABAP program.

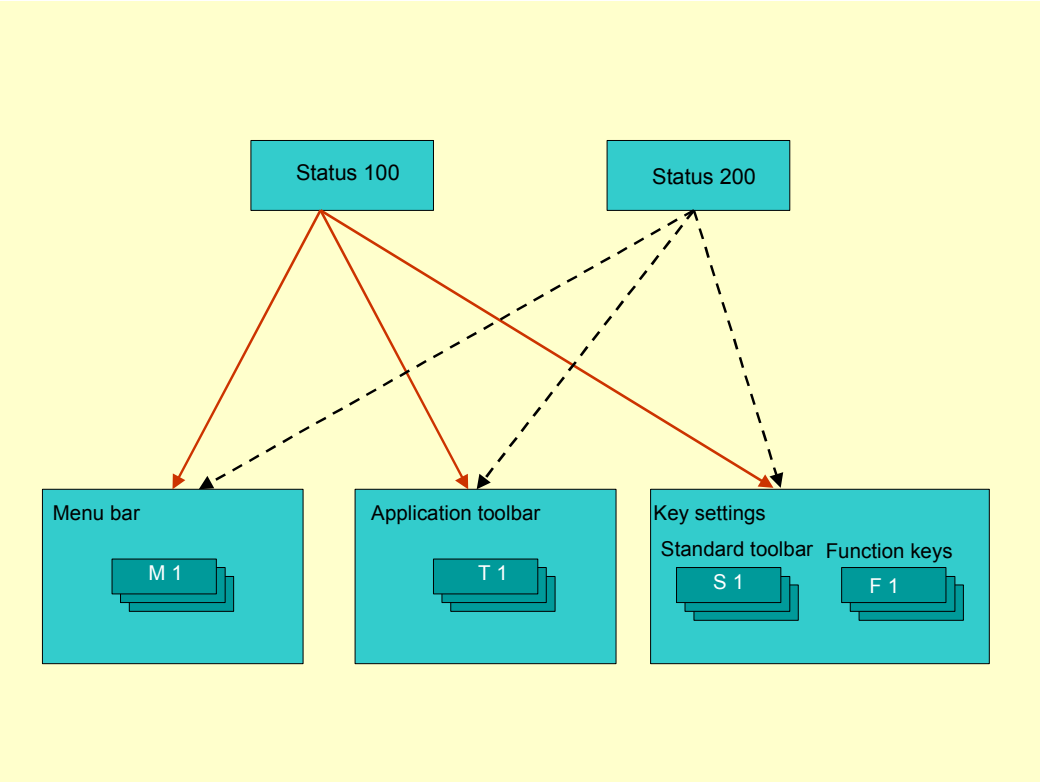
Components of the User Interface



Some statuses do not use all of these objects. For example, the status of a modal dialog box uses only function keys and an application toolbar.

Basic Concepts

- The principal object of a user interface is the GUI status. This can be assigned to any screen (screen, selection screen, or list). Technically, it consists of a **reference** to a menu bar, a standard toolbar, and a function key setting.



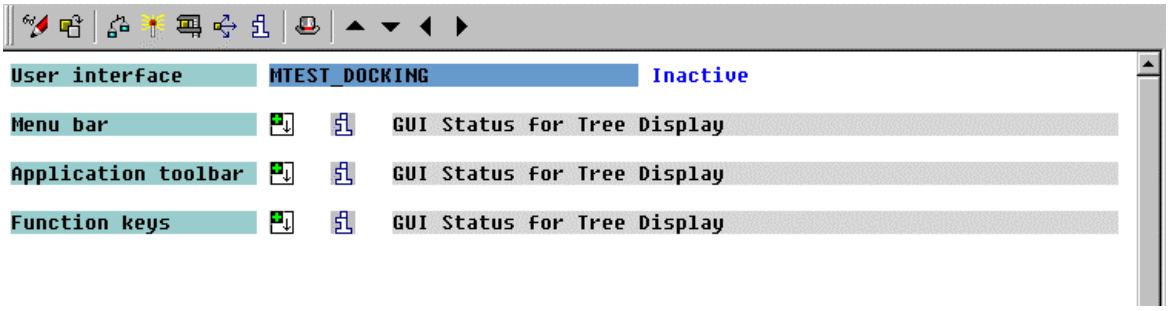
- Different GUI statuses can refer to common components.
- A program can have many GUI statuses and titles. These represent the different modes in which an application may operate.

  For example, a transaction might have two statuses - “Change” and “Display. In change mode, the delete function is active, but in display mode it is not.

- Several different screens can use the same status.
- You set the GUI title of a screen independently from its status.

The Menu Painter Interface

From the initial screen of the Menu Painter, you choose a status of a program, and then the work area is displayed. Here, you can edit all of the components of the screen.



From the worksheet, you can create a menu bar, define menu functions, assign F keys, customize the standard tool bar, and create an application toolbar.

To maintain the menu bar, application toolbar, or function key settings, click the *Expand* icon next to the relevant area.

Interface Object List

The Menu Painter keeps lists of all components of the GUI. These are used for component administration, and have functions such as *Create*, *Rename*, *Copy*, and so on. For example, the menu list contains the entries from all of the menu bars in the program. In all, there are six different lists (status list, menu list, function list, title list, menu bar list and function key setting list). You can access any of these lists from the *Goto* menu of the worksheet, or from the initial screen of the Menu Painter. Regardless of where you make changes, on the worksheet or through a list view, the Menu Painter updates and saves your changes in the same manner. You can also change menu texts, function texts, icons, title texts and short documentation if you access the list in change mode. The lists not only give a rapid overview of the interface components for the whole program, they also make it easier to maintain their text entries.

Saving Your Work

While you are within the Menu Painter, you do not have to save your work each time you move from the Worksheet view to a display list view. In principle, you only need to save your work before you leave the Menu Painter altogether. You do this using the *Save* function. However, if you do forget to save, the system will remind you by displaying a dialog box before you leave the Menu Painter.

Activating the Interface

None of the changes you make to your interface are visible at runtime until you activate it. Once you have completed and tested your changes, you must reactivate the interface. When you activate the interface, the system creates a load version of it. After activation, your last set of changes becomes visible in all programs that use the interface.

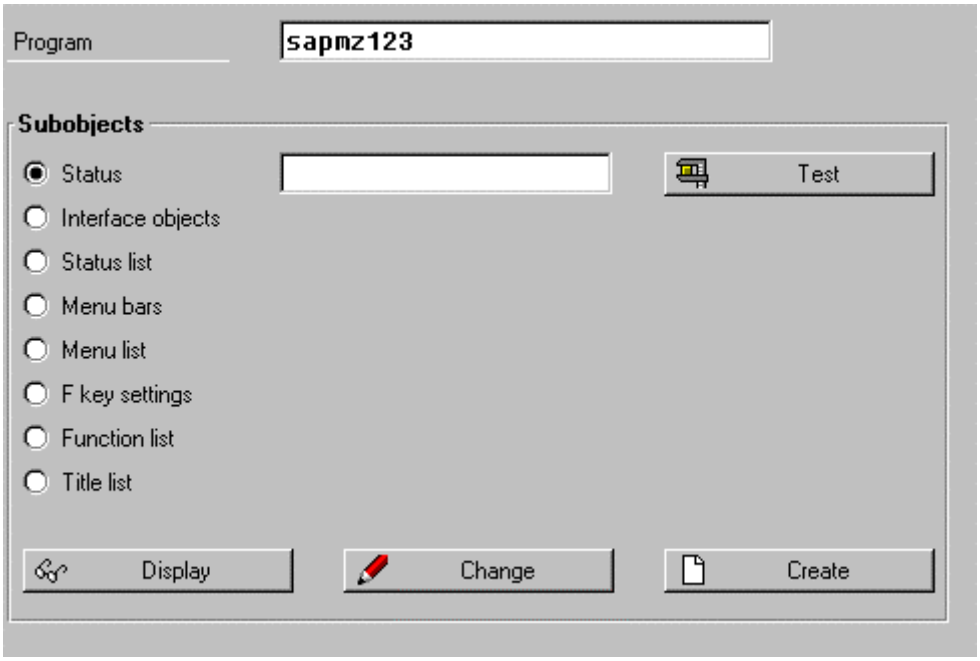
See also:

Inactive Sources [Page 509]

Menu Painter: Initial Screen

Starting the Menu Painter

To start the Menu Painter, choose the corresponding pushbutton on the initial screen of the ABAP Workbench, or enter Transaction **SE41**.



Use

From the initial screen of the Menu Painter, you can quickly gain an overview of all of the components of the user interface of an ABAP program, and take advantage of the efficient navigation functions. In particular, you can restrict yourself to a section of the user interface for editing.

From the initial screen you can:

- Create new statuses
- Test existing interfaces
- Display or change components of the interface

Components

Choose	To
Status	Open the Menu Painter work area
Interface objects	Display all user interface objects for the current program
Status list	Display a list of all GUI statuses for the current program

Menu Painter: Initial Screen

Menu bars	Display a list of menu bars, sorted by status
Menu list	Display a list of all menus
Function key settings	Display the list of function key settings
Function list	Display a list with all function codes
Title lisrt	Display a list of all GUI titles for the current program.

You can switch from any of the above lists into any other using the *Goto* function.

Creating a GUI Title

You should create a GUI title for each screen in a program. Titles help to orient the user: this is especially useful for transactions with several screens. To create a GUI title from the Repository Browser:

Procedure

1. Choose *Status* from the object list.

   The system displays a list of possible program objects.
2. Select *GUI title.*
3. Enter a title code (up to 20 characters).
4. Choose *Create*.

   The system displays a dialog box for the title bar information.
5. Enter the title you want to appear at the top of your screen.
6. Choose *Save*.

Result

You have now created a GUI title that you can set in an ABAP program using the following statement:

```
SET TITLEBAR <TITLECODE>.
```

If you do not set a title, the system uses a standard title.

Using Variables in Titles

To determine a title at runtime, you can use an & (ampersand) with your title text.

At runtime, these variables are replaced with the values that you specify. To set a titlebar containing variables, use the following ABAP statement:

```
SET TITLEBAR <titlecode> WITH <var1> <var2>... <varN>.
```

You can use up to nine variables in a title.
The variables are replaced with values according to their numbering (or simply from left to right if the variables are not numbered). `For further information, see the F1 help in the ABAP Editor.`

GUI titles remain set until you explicitly change them. At runtime, the system stores the current title in the SY-TITLE system field.

Defining a Status

To create a new status using the Menu Painter:

- Create the status
- Create the menu bar (or create a reference to an existing menu bar)
- Add menu entries
- Define new function key settings
- Define the standard toolbar (or create a reference to an existing one)
- Define the application toolbar (or create a reference to an existing one)
- Test the status

- Activate the status

Navigation

You can maintain a status from various points within the ABAP Workbench.

- By forward navigation from the ABAP Editor
- From the Object Navigator, under *Program objects*
- From the initial screen of the Menu Painter

Creating a GUI Status

Procedure

1. Enter the name of your ABAP program.
2. Choose *Status* from the object list.
3. Enter a status name.

   A status name can consist of up to 20 alphanumeric characters.
4. Choose *Create*.

   The *Create Status* dialog box appears.
5. Enter a short description.
6. Select a *Status Type*.

   The status type enables the Menu Painter to display the correct work area for the kind of status you want to create. The following status types follow the standards in the SAP Style Guide.

Status type:	References to
Dialog status	Menu bars, standard toolbar, function keys and application toolbar
Dialog box	Function keys and application toolbar. Dialog boxes do not have menu bars or a standard toolbar.
Context menu	A context menu. Context menus contain a set of functions that you can use to construct a context-sensitive menu. See also Creating a Context Menu [Page 369].

7. Choose *Continue*.

   The system displays the work area of the Menu Painter.

   Define the components of the interface (or refer to existing interface components).

   See also:
   Defining Key Assignments [Page 377]

   Creating Menu Bars [Page 372]

   Defining Pushbuttons [Page 381]

   Creating Standard Toolbars [Page 386]
8. Save your new status.

Result

The new status object appears in your program object list.

You cannot call a status in your program until you have activated it. (**See also:** Testing and activating statuses [Page 387])

Creating a Context Menu

Use

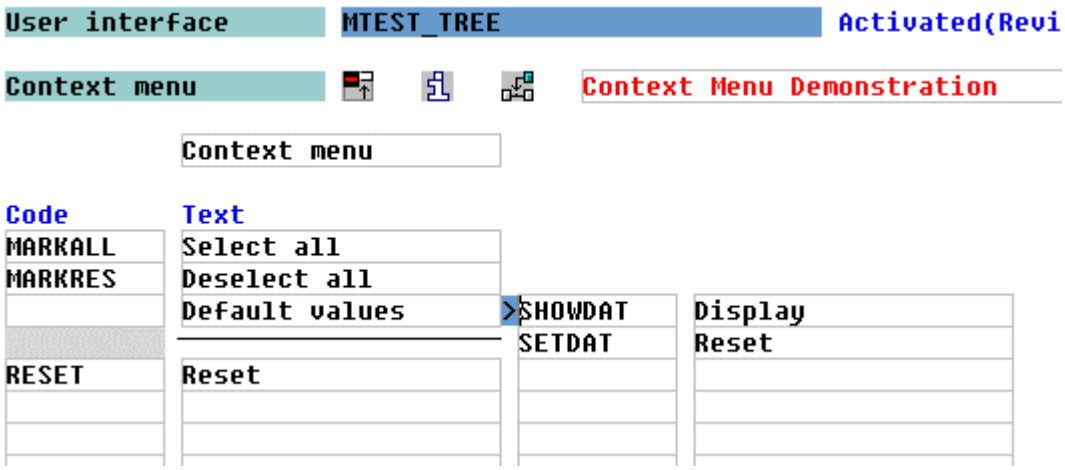
You can use the entries from a context menu that you define in the Menu Painter to construct a context menu on a screen or list using the method **LOAD_GUI_STATUS** from the global class **CL_CTMENU**.

Procedure

To create a context menu from the Object Navigator:

1. Select *Program objects* and choose *Edit*.
2. Enter the name of the ABAP program.
3. Choose GUI Status and enter the name of your context menu.
   This is the name that you will pass to the importing parameter **STATUS** of the method **LOAD_GUI_STATUS**.
4. Choose *Create.*
   The *Create Status* dialog box appears, containing fields for the status attributes.
5. Enter a short text.
6. Select the status type *context menu*.
7. Choose *Continue.*
   The work area for the context menu appears.
8. In the *Code* column, enter a function code, and under *Text*, the corresponding text for the menu entry.
9. Repeat step 8 for each further function you want to add to your context menu.
10. If you want to enter a separator, choose *Edit* → *Insert* → *Separator*.
11. If you want to create a cascading menu, leave the *Code* field empty, and enter the menu text for the cascading menu. You can then open it by double-clicking, and enter the required entries.

Example:



Result

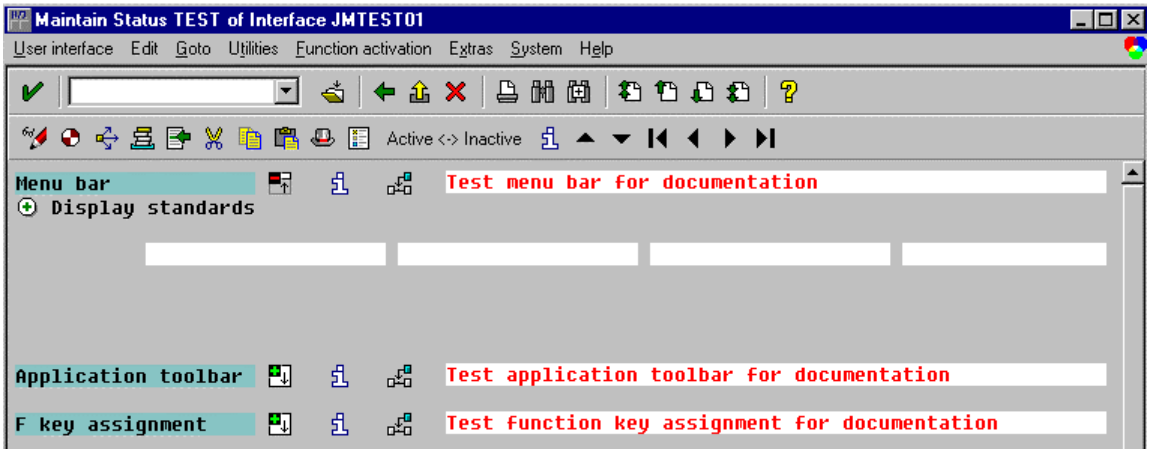
You have created a set of functions that can be used as a template for constructing a context menu.

For information about how to define the relevant processing logic in ABAP, refer to Context Menus [Ext.].

Working with Menu Bars

Each new status that you create can have a reference to its own menu bar. You can either create a new menu bar, or link to an existing one.

A menu bar may contain up to six menus. The system then adds the two standard menu bars - *System* and *Help*. These two menu are always displayed at the right-hand end of every menu bar in the R/3 System.
If you want to create a new menu bar, the status work area starts with an empty menu bar area.



Using Standard Proposals

When you create a menu bar, you can use the SAP ergonomic standards as a template. To display this template, choose *Display standards*. You can adopt all of these menus as a starting point and edit the list as you like.

To revert to a blank menu bar, choose *Hide standards*.

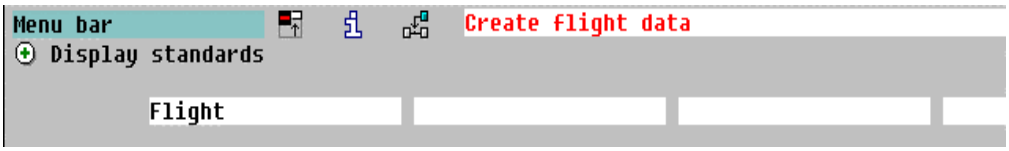
In a status for a list, you can also adopt individual default functions. Choose *Display list functions* to display the defaults. You can either adopt the defaults, or change them to suit your own requirements.

Creating a Menu Bar

Procedure

1. With the Menu Painter in change mode, open an empty menu bar using the Expand icon to the right of the *Menu bar* text field.
2. If you want to use the default settings, choose Display standards.
3. If you are using the standards, change the first menu title <Object> to something relevant to your application.
4. If you are not using the standards, enter the menu titles as required.

   For example:



5. Add menu entries [Page 369] to the menus.

Observing Standards

The Menu Painter supports the user interface standards outlined in the SAP Style Guide by proposing names for GUI objects. You can display the proposals either from the initial screen of the Menu Painter, or from the work area itself.

Choose *Utilities* → *Help texts* → *Standards/Proposals*.

Following these proposals helps you to create transactions with the look and feel of the standard R/3 applications.

Adding Functions to a Menu

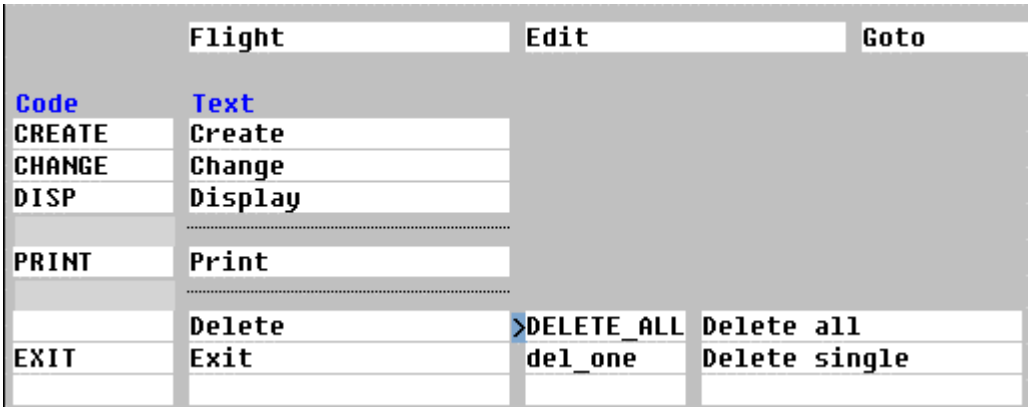
Once you have created a menu bar, you enter the individual menu entries. Each menu can contain up to 15 entries.
A menu can contain any of the following:

- Function names (with function code and text)
- Submenus (pull-down menus)
- Separators

You may include submenus up to three levels deep.

Menu functions that logically belong together are grouped together using separators. This makes the menu easier to use. Separators also make long menus easier to read by dividing them into smaller parts.

Example:



Defining Menu Functions

To add functions to a menu that is already open in the Menu Painter:

1. Open a menu list in the menu bar by double-clicking the menu title.

   The system opens the menu. The menu entries list contains the two columns *Code* and *Text*.

2. In the Code column, enter a function code (this may be up to 20 characters long).

   If you want to enter a function code that is longer than the input field, you must first change the displayed length of the field in the user settings. (Choose *Utilities* → *Settings* → *User-specific*.)

3. Enter the function text in the *Text* column.

   The name you enter here appears in the menu at runtime. You can also determine the contents of function texts at runtime (see Defining Dynamic Function Texts [Page 403]).

4. Repeat steps 2 and 3 for each item in the menu.

Creating Cascading Menus

To add a cascading menu (sub-menu) to a menu:

1. Leave the *Code* column blank.
2. Enter a menu name in the *Text* column.
3. Double-click the cascading menu to open it.

   The system opens the menu entry list for the cascading menu.
4. Complete the menu as you would any other.

Inserting Separators

1. Place the cursor on a line.
2. Choose *Edit* → *Insert* → *Separator*.

If you want to insert a separator between two existing menu entries, place the cursor on the line before which you want to insert the separator.

Editing Menu Entries

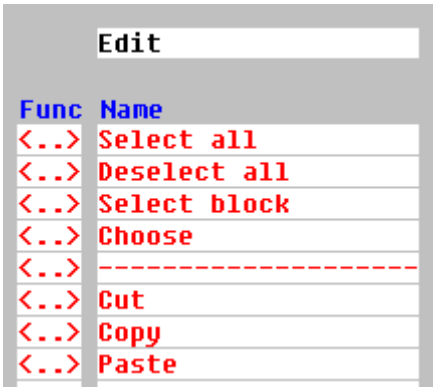
You can cut, copy, paste, and delete menu entries.

1. Place the cursor on a line.
2. Choose *Edit* → *Entry* → *Cut* (or *Copy*, or *Paste*, or *Delete*).

If you double-click a function code, the Function Attributes dialog box appears. Here you can, for example, change the icon assigned to a function.

Using Style Guide Proposals

Some functions are standard on SAP menus. If you are starting with the SAP standard menus, these standard functions appear on your menus:



Adding Functions to a Menu

To use a standard function, activate it by entering a function code next to it. If you do not enter a function code, the function remains inactive, and is not displayed at runtime.

Defining Function Key Settings

The R/3 System uses function keys to allow users quick access to commonly-used functions. The function keys are special keys on the keyboard (F keys) which enable you to trigger functions without having to use the menu.

To add function keys to your interface, you must assign each function code a specific function key.

In the R/3 System, there is a distinction between:

- Reserved function keys
- Recommended function key settings
- Freely assigned function keys

The following function keys are reserved by SAP, and cannot be changed:

F1	*Help*
F3	*Back*
F4	*Possible entries*
F12	*Cancel*

The reserved or recommended function keys depend on the type of status. For further information, see the SAP Style Guide. Choose *Utilities* → *Help texts* → *Standards/Proposals*.

Procedure

To create a function key setting:

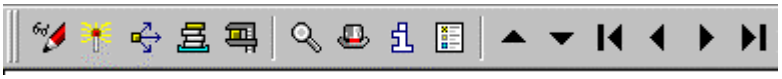
1. With the Menu Painter in change mode, open the work area.
2. Expand the *F key assignment* section of the work area.
3. Enter a function code in the first input field for the corresponding function key.
   If you want to link to an existing function key setting, see Linking objects in a GUI status [Page 393].
4. In the second input field, enter a text for the function. This text is then visible as documentation whenever the user presses the right mouse button.

When you define a function key in the *Recommended function key settings* group, the system proposes a default function description. However, you can overwrite this text. You should only assign functions to these function keys that correspond to the standard description, since these are intended to be used uniformly throughout the R/3 System.

Defining an Application Toolbar

You can create pushbuttons for the most useful functions in your programs. These are arranged in the application toolbar, which appears below the standard toolbar.

The following is an example of an application toolbar:



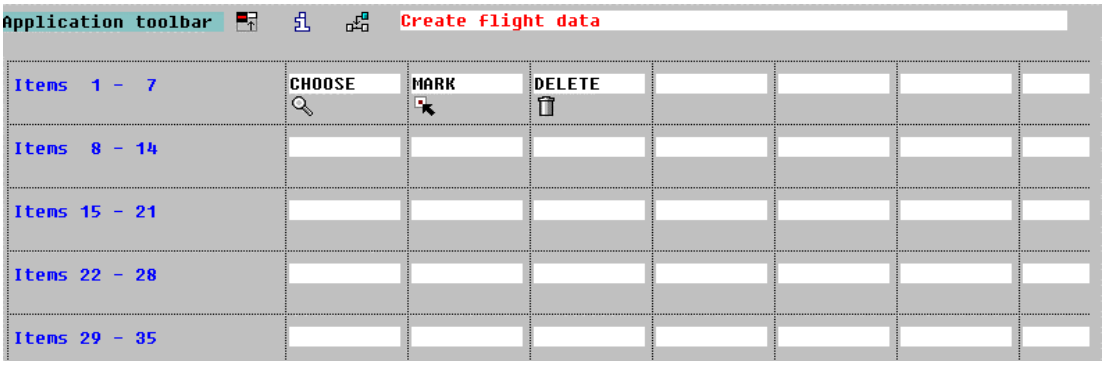
Notes:

- You must have assigned a function to a function key before you can include it in the application toolbar.
- You can include up to 35 pushbuttons in the application toolbar.
- Application toolbar functions can contain an icon, a text, or both together.
- You can display inactive functions within the application toolbar if you have defined fixed positions [Page 384] for it.
- You can also assign a dynamic text to a pushbutton at runtime. To find out how to assign dynamic texts, see the section Defining Dynamic Function Texts [Page 403].
- You can group functions in the application toolbar using separators. See also Inserting Separators [Page 385].
- The Style Guide also recommends you should only create pushbuttons for functions that are also available on a menu. For further information, choose *Utilities* → *Help texts* → *Standards/Proposals*.

Procedure

1. Assign a function to a function key. For details of how to do this, see Defining a function key setting [Page 377]
2. Expand the application toolbar and enter a name for it.
3. Enter the function code (that you assigned to the function key in step 1) in one of the pushbutton fields in the application toolbar, and press ENTER.

   The system automatically fills in the text for the function definition, and an icon, if one exists.



4. Repeat step 3 for each additional pushbutton.
   When composing the application toolbar, you should adhere to the standards in the SAP Style Guide.
5. If required, you can add icons to the pushbuttons. To find out more about assigning icons to pushbuttons, refer to the section Icons in the Application Toolbar [Page 381].

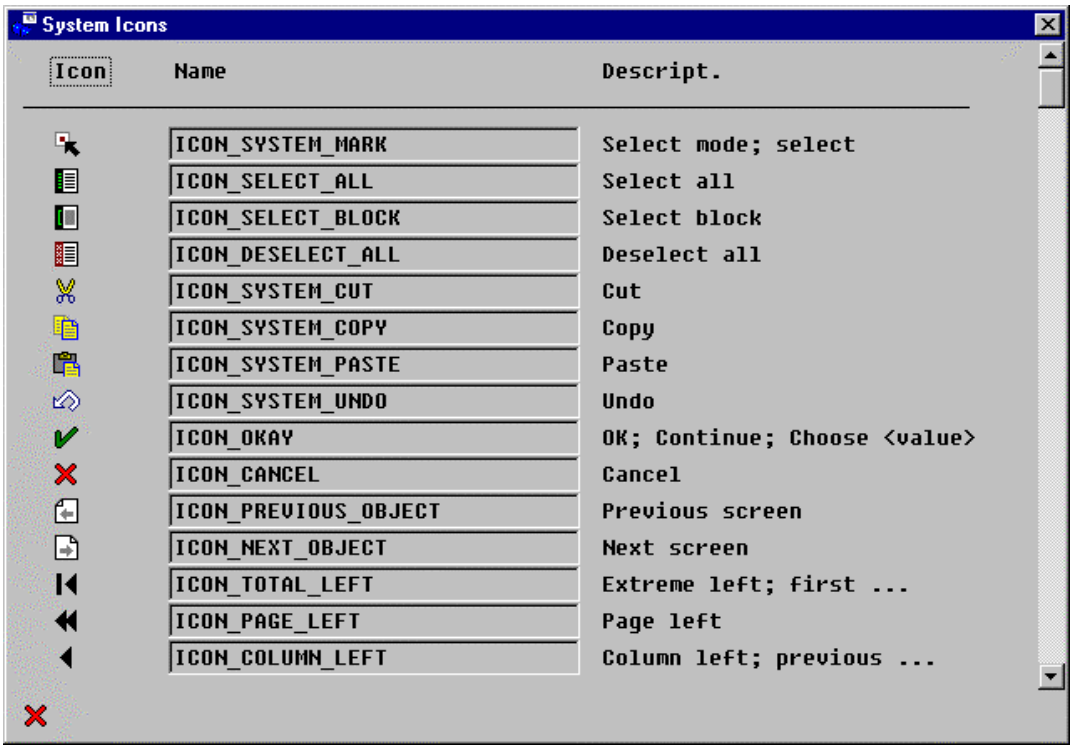
Defining Icons in the Application Toolbar

You can create icons for functions in the application toolbar. An icon can appear alone or with text.

The set of icons available for use in the Menu Painter is not exactly the same as that available in other editors (for example, in the Screen Painter). In the icon pool, all icons are classified into different groups according to where they are used.

You can assign an icon to a function as follows:

- Enter an icon in the *Icon name* field in the *Function list.*
- Double-click a function in the function key setting.
  The *Function attributes* dialog box appears, in which you can enter an icon in the *Icon name* field.
  You can display a list of possible icons using the possible values help.



Further Options

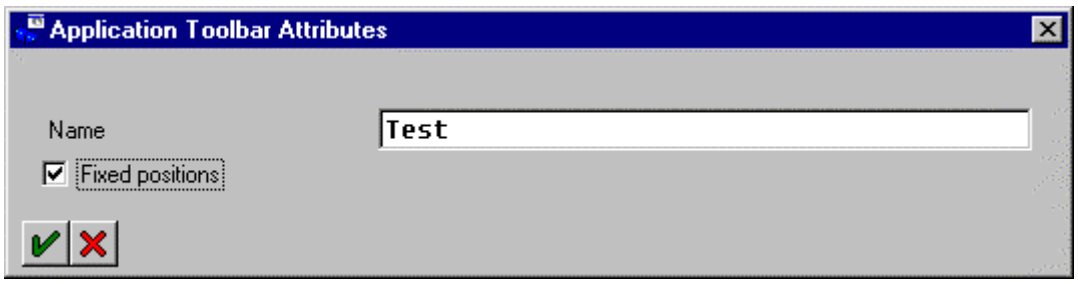
- If you want the icon to appear with an explanatory text, enter a text.
- Enter an *Infotext* if required. The system displays the quick info text when the user places the cursor on the icon or holds down the right mouse button.

Fixed Positions

If you set the Fixed positions attribute, the pushbuttons in the application toolbar are not shifted when you activate or deactivate functions dynamically.

Procedure

1. Start the Menu Painter [Page 353].
2. Switch to change mode.
3. Choose *Goto* → *Attributes* → *Application toolbar* or choose the padlock icon to the right of the application toolbar
4. Set the *Fixed positions* option, and choose *Continue*.



Result

The padlock icon next to the application toolbar appears 'locked'..

Inactive functions in the application toolbar will now appear grayed out, regardless of whether they are inactivated in the Menu Painter or dynamically in the program.

Inserting Separators

Use

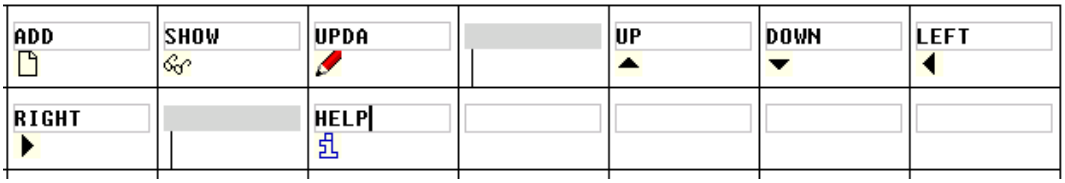
As in menus, you can divide up application toolbars into groups of related functions. The divisions are marked by separators.

Procedure

To insert separators into an application toolbar:

1. Open the work area of the Menu Painter.
2. Switch to change mode.
3. Position the cursor where you want to insert a separator.
4. Choose *Edit* → *Insert* → *Separator*.
   The system inserts a separator at the cursor position.

Example:

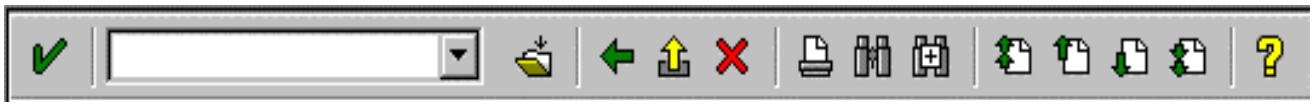


Result

The application toolbar designed above would look like this at runtime:

Creating the Standard Toolbar

- The standard toolbar is identical for all R/3 interfaces with status type *Screen* or *List*.



- If a function is inactive, its icon is grayed out.
- Interfaces with the status type *Dialog box* or *List in dialog box* do not have a standard toolbar.

Prerequisites

The functions that you want to assign to the standard toolbar must already have been assigned to function keys. See also Assigning Function Keys [Page 379].

The SAP Style Guide recommends that you always activate at least the *Back*, *Exit*, and *Cancel* functions in the standard toolbar.

Procedure

To create a new standard toolbar in the Menu Painter:

1. Open the work area of the Menu Painter.
2. Switch to change mode.
3. Enter the required function codes above the relevant icons in the *Standard toolbar* section.

Testing and Activating a Status

Testing allows you to check a simulation of the status as it will appear at runtime. The system always uses the inactive version of the status where one exists.

Changes to a GUI status do not become visible in your application until you have reactivated the status.

Testing a Status

1. Choose *User interface* → *Test status*.

   The system displays the *Status Simulation* dialog box.

2. Enter a *screen number* and *title* if you want to simulate a whole screen. If you do not enter a screen number, the system simulates the status using an empty test screen.

3. Choose *Execute*.

   A dialog box appears, containing the window coordinates.

4. Choose *Continue*.

   The system simulates your status.

Activating a Status

To activate a status, choose *User interface* → *Activate*. When you activate a status, the system automatically checks for syntax errors.

To check a status for syntax errors without activating it, choose *User Interface* → *Check syntax*. This function checks the user interface and then produces a list of syntax errors that you must correct before activating your status.

See also:

Inactive Sources [Page 509]

Using the Extended Check

The Menu Painter's extended check examines all the GUI statuses in your program for correctness, syntax errors, and completeness. You can correct any errors from within the extended check. The extended check is based on the standards defined in the SAP Style Guide [Ext.]. These are checked against the attributes that you have assigned to the components in your interface.

Starting the Check

To start the extended check, choose *Utilities* → *Extended check* either from the initial screen of the Menu Painter, or from the work area.

The system displays the result of the check procedure in the form of a hierarchical tree structure. There are two error categories:

- Warnings
- Violations of SAP standards

Warnings include oversights such as undefined function keys or missing fastpaths. Violations of SAP standards include incorrect function key settings or inadequate assignment of functions to a menu,

Further Information 

Choose the information icon to display more information about an error. Select a node in the hierarchy tree and choose *Information* Here, the system also displays information about how to correct the error. If you select the *Warnings* or *Standard violations* node, the system displays the general documentation.

Correcting Errors

To correct errors in the extended check:

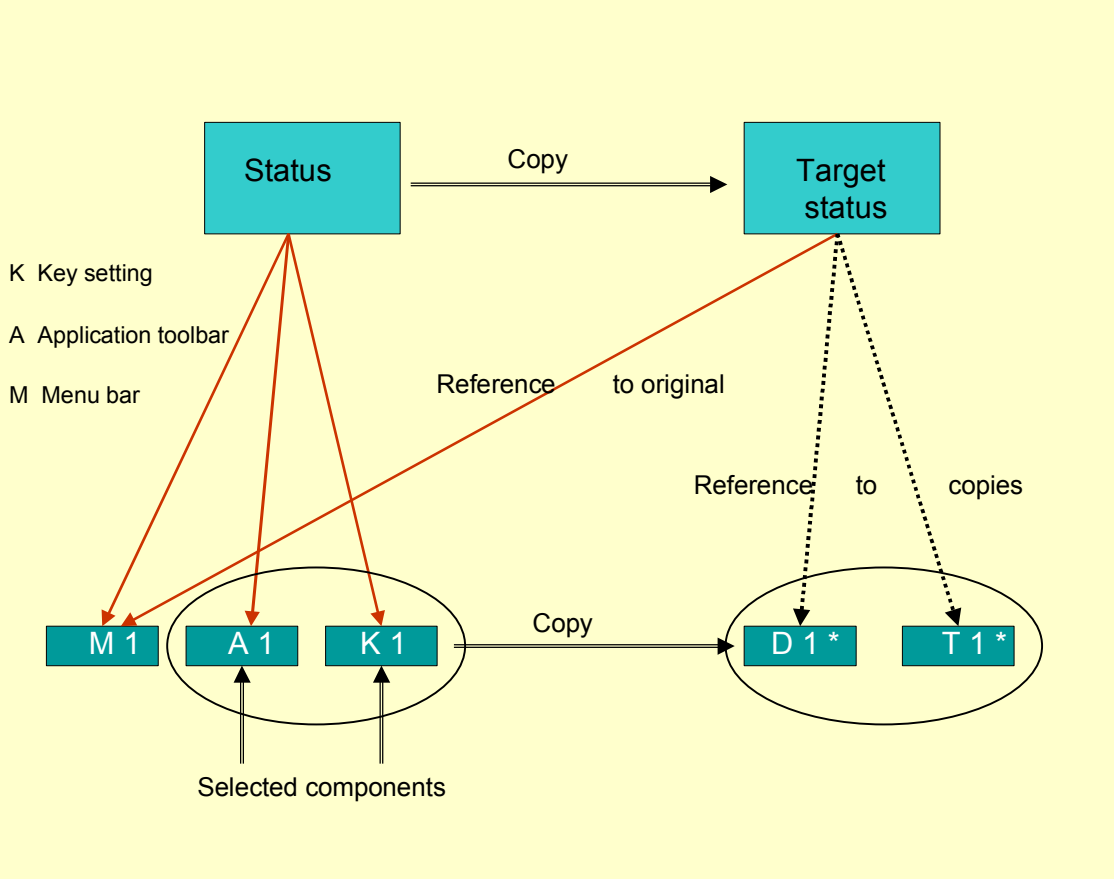
1. Open the appropriate node in the tree display.
2. Choose an object from the check list by double-clicking it. The system opens the corresponding work area in the Menu Painter.
3. Ensuring that you are working in change mode, correct the error.
4. Choose *Back* to return to the tree display.

Copying a Status

You can create a new GUI status by making a copy of an existing status.

There are two ways to do this.

- You can copy a status within a program. If you do this, there are two further possibilities:
  - The target status uses the same components. In this case, the status references are maintained, so any changes to components are visible in both statuses.
  - The target status only uses some of the same components. In this case, you choose which components are to be used. These are copied into the new status. In the new status, all references to the components are replaced with references to the copy. This means that changes to these objects are only visible in the target status. The references to the other (unselected) components are kept. This principle is explained in the following diagram.



Copying a Status

- You can copy a status from a different program. In this case, the system copies all components.

  If you want to use a standard interface for your program, you should reference these components to the original.

Procedure

This procedure tells you how to create a new status by copying an existing status from the same program. You start on the initial screen of the Menu Painter.

1. Choose *Status*.

   The *Copy Status* dialog box appears. By default, the current program and status appear as the *To* and *Frm* program and status.

2. Enter a new status for the *To* status.

3. Choose *Copy*.

   In the next dialog box, you can choose whether the copy should use all of the same or some of the same components. In the second case, you can choose a set of components. The default option is *Same components*. However, you can also choose *Partly the same...* (see above diagram). In this case, you must then select the components that you want to use.

4. Select the components that you want to use.

5. Choose *Copy*.

   The system creates the new status in the same program.

When you copy the status into another program, step 4 is omitted, since all components are automatically copied.

Linking Objects in a GUI Status

There are two ways of creating a status:

- By creating new components with a unique reference to the status.
- By referring to existing components, and using them more than once in several different statuses.
  When you do this, you can create individual menu bars, menu functions, and function key settings either for an arbitrary basic status, or from the appropriate overview list. New statuses can then use these components.

Copying a Menu Bar

1. Create a new status that you want to link to a menu bar.
2. In the Menu Painter, choose *Edit* → *Copy* → *Menu bar* (or the Assign icon ).
   The system displays a list of all of the existing menu bars for the current program.
3. Choose a menu bar by double-clicking it.
   The status links to the functions on the list. However, the functions are not yet active.
4. Activate the functions.

Copying a Function Key Assignment

1. Create a new status that you want to link to a function key setting.
2. In the Menu Painter, choose *Edit* → *Copy* → *Function key setting* (or the *Assign* icon ).
   The system displays a list of all of the existing function key settings for the current program.
3. Choose a function key setting by double-clicking it.
   The system creates the link to the function key setting. However, the functions are not yet active.
4. Activate the functions.

Copying an Application Toolbar

1. Create a new status that you want to link to an application toolbar
2. In the Menu Painter, choose *Edit* → *Copy* →*Pushbutton settting* (or the *Assign* icon ).

   The system displays a list of all of the existing application toolbars for the current program.
3. Choose an application toolbar by double-clicking it.
   The system creates a reference to the application toolbar. However, the functions are not yet active.
4. Activate the functions.

Removing Links

There are several ways you can remove a link. The simplest method is to simply overwrite the link with new information.

Linking Objects in a GUI Status

You can also use the *Edit* → *Initialize* function to reset a function key, menu, or pushbutton setting.

Working with Overview Lists

Reusable components of the program interface are grouped together in overview lists. These are sorted by category into a menu list, title list, and so on. You can choose individual objects from these lists and edit them. Objects that are **not used** within the interface are also included in the lists.

Unused objects are not assigned to a status. For example, if you delete a menu from the work area, the system does not delete it from the menu list, even though it no longer appears in the interface. The menu remains in the menu list, and you can link it in a new status reference later on if required.

Accessing Lists

To access the display lists from the worksheet, select the *Goto* menu and an object (*Menu list*, *Function list*,...) You can switch back to the worksheet screen from a display list by selecting *Goto → Current status.* You can also access the lists from the initial screen of the Menu Painter.

Displaying All Objects in a Category

If, for example, you want to maintain one or more menus within a user interface, you can access the complete menu list by selecting *Goto → Menu list*. The menu list contains all the program's menu functions. Within this list, you can create, delete, or copy menus as required. You can also select a menu for editing by double-clicking its name.

Deleting Interface Objects

To delete a component, you must delete it explicitly from its display list. You can delete any sub-object in a display list by choosing *User interface → Delete → Sub-object.* If the object is still used somewhere in the program's interface, the system issues a warning. You can either confirm the deletion or choose the *Where-used list* to display a list of all links to the object.

When you delete an object from a list, the system deletes the object and any links from your program's interface.

Locating and Deleting Unused Objects

The *Unused objects* function generates a list of all objects that are defined in your interface but not used. You can do this by choosing *Utilities → Unused objects*. The system displays a list of the unused objects. If the Menu Painter is in change mode, you can delete any unused objects directly from this list.

Area Menu Maintenance from Release 4.6A

Functions

You execute a function whenever you choose a menu entry or press a function key or pushbutton. The element in the interface is linked to the ABAP program itself by a unique function code, which you assign to the interface element when you create it in the Menu Painter. When you choose a function, its function code is passed either to the system field SY-UCOMM (in type 1 programs) or to the OK_CODE field (in transactions). You can address these fields in your programs to find out which function the user has chosen.

Defining a Function

The definition of an interface function contains the following elements:

- **Function code**: Unique key for the function, which can be interpreted by the ABAP program.
- **Function Type**: is used to determine processing control. Function types can, for example, tell the system when or how to carry out a function. (**See also:** Using function types [Page 398])
- **Function text**: A text that describes the function (such as *Save*).
- **Icon name**: Name of the icon to be displayed on a pushbutton.
- **Icon text**: Text to be output on the pushbutton in addition to the icon.
- **Infotext**: Text to be displayed in the status bar.
- **Fastpath**: Letter combination that allows users to choose functions without using the mouse. See also Defining a fastpath [Page 399]

Specifying Further Options

- When you define a status, you can set functions to either active or inactive (**see also** Activating and deactivating functions [Page 400]).This allows you to use a single predefined menu bar, application toolbar, and function key setting in all statuses of an application, since you can deactivate any functions that are not supported on the current screen.
- You can also deactivate functions dynamically. This means that you do not need to create a new status if you only want to deactivate individual functions (**See also:** Deactivating functions at runtime [Page 402])
- You can also change function texts and menu texts dynamically (**See also:** Defining dynamic function texts [Page 403] and Defining dynamic menu texts [Page 405]).

Using Function Types

When you create functions, you define a function code and a name. When a user chooses a function, the system stores the function code in the SY-UCOMM field. The code tells the system which function a user chose.

You can also assign a type to your functions. Function types can, for example, tell the system when or how to carry out a function. The system uses the following function types:

Type	Meaning
	Normal function code processing (for example, in a PAI module).
E	Triggers an "at exit-command" module in the Process After Input (PAI) processing block (MODULE <xxx> AT EXIT-COMMAND). When the user selects an E type function, the system enters the "at exit-command" module before carrying out any input checks.
T	Calls another transaction. Activating a T type function has the same effect as the LEAVE TO TRANSACTION statement. The user cannot return to the original transaction.
S	Triggers system functions used internally in SAP standard applications. You should not use type 'S' when creating your own functions.
P	Triggers a function defined locally at the GUI. This function is not passed back to the ABAP program (no SY-UCOMM or OK_CODE). Instead, it is processed at the presentation server. This type of function can only currently be used for **tabstrip controls** (Screen Painter).

Assigning Function Types

To assign function types:

1. Double-click a function in the GUI status.
   The *Function Attributes* dialog box appears, which displays an overview of all of the function attributes.
2. Choose a function type in the *Function type* field.
3. Choose **ENTER** to continue.

Defining a Fastpath

Fastpaths allow you to enter a menu path using a single letter or sequence of letters instead of the mouse.

Example:

You can choose the function Delete from the Edit menu by entering **L**.

You can also enter the entire menu path using the letter combination preceded with a period. In our example above, you would enter.**BL** in the command field.

Rules

- The fastpath for a menu must be unique within the menu bar. The fastpath for a function must be unique within its menu.
- The letter must be a part of the complete function name. For example, you could not choose *Z* as the fastpath for the *Create* function.

In **double-byte languages**, the English fastpath keys are used. The frontend displays the double-byte language text in the menu, with the English letter for the fastpath in parentheses afterwards.

Maintaining a Fastpath

You do not have to maintain fastpaths explicitly. As soon as you generate the interface, the system creates a set of default fastpaths automatically. You can then change the fastpath if required.

1. To change the fastpath, open the status in the Menu Painter.
2. Choose *Goto* → *Object lists* → *Fastpath*.
   The fastpath maintenance screen appears. Here, you can enter a fastpath for each function.
3. To display a default set of fastpaths, choose *Propose fastpath.*
4. Choose *Save*.

Activating and Deactivating Function Codes

Within a status, functions can be active or inactive. Active functions are executable functions and inactive functions are not executable. You should deactivate a function if the mode of your application changes and the function is no longer available. For example, in view mode a function like delete would not be active.

If inactive functions are assigned to the application toolbar, the system does not display the function unless you have set the fixed positions [Page 384] option for the application toolbar.

In the standard toolbar, the system grays out inactive functions at runtime.

Inactive functions are grayed out in the menu. You can activate or deactivate menu options from within the Menu Painter. You can also deactivate functions at runtime. To find out more about dynamic deactivation, see Deactivating a Function at Runtime [Page 402]

You can activate or deactivate functions for:

- a single function in one status.
- several functions in one status (for example, after adopting a menu bar, application toolbar, or function key setting).
- single functions in more than one status in an application (for example, when you want to add new functions to a menu that is used more than once).

Procedure

To activate or deactivate a function code in the status work area of the Menu Painter:

Individual Functions

1. Position the cursor on the corresponding function in the function list.
2. Choose *Extras* → *Function Active <-> Inactive*.
   The function code and description are now displayed in a different color. The system has activated deactivated functions or deactivated active functions.

Several Functions

1. To activate several functions in the current status, select *Extras* → *Active functions in current status*. The system displays a list of all functions used in the status.
2. You can now deactivate a set of functions by deselecting the corresponding checkbox. .
3. Choose *Copy*.

For Several Statuses

1. To activate several functions in the current status, choose *Extras* → *Active functions in multiple statuses*.
2. Enter a function code
3. Choose *Continue*.
   The system generates a list showing all the statuses where the function is used.
4. Select the statuses in which you want to make the change.
5. Choose *Copy*.

Deactivating Functions at Runtime

You can deactivate menu functions dynamically at runtime. To do this, use the EXCLUDING addition when you set the GUI status. You can deactivate either individual functions, or a whole group of functions.

Deactivating a Single Function

Suppose you have a GUI status called **CREATE**. This GUI status could have a *List reservations* menu option with the function code **LIST**. If *List reservations* is active by default, you can deactivate it at runtime with the following statement.

```
SET PF-STATUS 'CREATE' EXCLUDING 'LIST'.
```

If *List reservations* is included in a menu, the system grays out the function text. If the function is assigned to a pushbutton, the pushbutton is not displayed unless you have set the fixed positions [Page 384] option for the application toolbar.

Deactivating a Group of Functions

You can deactivate several functions at once by filling an internal table with all the function codes you want to deactivate. Following our example above, and using an internal table called **itab**, you would do this as follows:

```
SET PF-STATUS 'CREATE' EXCLUDING itab.
```

For further information about how to deactivate functions dynamically, see the online documentation for the **SET PF-STATUS** statement.

Defining Dynamic Function Texts

If you want a menu entry or a function to have a variable text at runtime, you can define dynamic texts. To do this, you must define a field in your ABAP program that will contain the required text at runtime.

When you define the text, use the structure **SMP_DYNTEXT**.

For example:

```
DATA: TEXT_1 LIKE SMP_DYNTXT,
      TEXT_2 LIKE SMP_DYNTXT.
```

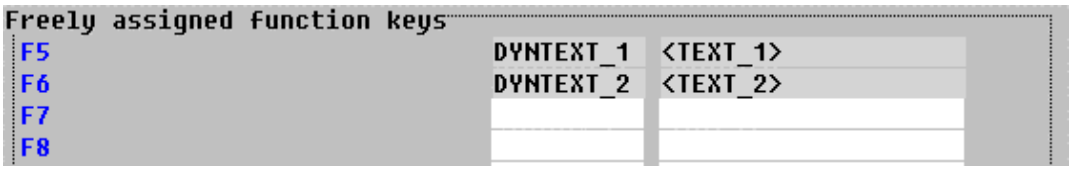
Procedure

1. Place the cursor on an empty function line.
2. Choose *Edit* → *Insert* → *Func.with dyn. text.*

   The *Insert Function with Dynamic Text* dialog appears.
3. Enter a function code
4. Choose *Continue*.

   The *Enter function text* dialog box appears.
5. Enter a program or ABAP Dictionary field name.
6. Choose *Continue*.

   The system display the field in <> (brackets) as follows:



For information about how to create a dynamic text for a menu, see Defining dynamic menu texts [Page 405]

Changing Function Texts

You can change static function texts to dynamic ones, and vice versa. To change a static text into a dynamic text:

Defining Dynamic Function Texts

1. Double-click the function that you want to change.

   The *Function Attributes* dialog box appears.

2. Choose *Change text type*.
3. Enter the name of the program or ABAP Dictionary field in the *Field name* field.
4. *Choose Continue*.

   The system display the dynamic text in brackets <>.

   Since the field name you specify to store the function name can be up to 132 bytes long (as on the screen), the system can not always display the field in its full length. You can change the field name by double-clicking the function or menu name.

Defining Dynamic Menu Texts

If you want to change menu texts dynamically in your program, you can assign a dynamic menu text as follows:

1. With the Menu Painter in change mode, open the menu bar.
2. Position the cursor on a blank field in the menu bar.
3. Choose *Edit* → *Insert* → *Menu with dyn. text.*
4. Enter a short description in the *Short documentation* field, and the name of a program or ABAP Dictionary field in the *Field name* field.
5. Choose **ENTER**.

Setting a GUI Status and GUI Title

The GUI status and GUI title defines how the user interface will look and behave in an ABAP program. Users cannot choose functions until you have set the status in the program. You set a status and title for a screen in the PBO (Process Before Output) module using the ABAP keywords

- SET PF-STATUS
- SET TITLEBAR

For further information, see the F1 help in the ABAP Editor.

Example

Suppose you have a program with a screen 100. If you want this screen to appear with the menus, standard toolbar, and application toolbar from GUI **CREATE** status, you must insert the following code into a Process Before Output (PBO) module of screen 100:

```
SET PF-STATUS 'CREATE'.
```

To call screen 100 with the title bar '100', insert the following code into the same PBO module:

```
SET TITLEBAR '100'.
```

When you set a GUI status or a GUI title, it remains set until you explicitly set a new GUI status or title. In the above example, if you call screen 200 without setting a new status, screen 200 will appear with GUI **CREATE** status and title bar **100.**

Using the Same User Interface in Several Programs

If you want to use a GUI status or GUI title from another ABAP program, you can use the `SET` statement with the addition `OF PROGRAM`. For further information, see the F1 help for the `SET` statement.

Evaluating Function Codes in the Program

When you create new menu and toolbar functions, you must assign a unique code to every function. When the user chooses a function, its function code is placed in the system field SY-UCOMM and the screen field OK_CODE.

The SY-UCOMM field always contains the current function code. You do not need to declare this field in your module pool.

OK_CODE

The OK_CODE field stores the function code in your program. It is always the last field in the field list of a screen. You need to assign a name to this field in the Screen Painter. Traditionally, this field is always called “OK_CODE”. However, you can call it anything you like. Once you have assigned a name to the OK_CODE field, you need to declare a field with the same name in your module pool.

Example:

If your GUI status contains the function codes 'BACK', 'EXIT', and 'SAVE', for example, you need the following code in your PAI module.

```
MODULE USER_COMMAND_0100.
      CASE OK_CODE.
            WHEN 'BACK'.
                  ...
            WHEN 'EXIT'.
                  ...
            WHEN 'SAVE'.
                  ...
      ENDCASE.
ENDMODULE.
```

Function Builder

The Function Builder allows you to create, test, and administer function modules in an integrated environment.

If you want to use the Function Builder in conjunction with the Modification Assistant, refer to Modifying Function Modules [Ext.].

Overview of Function Modules

Function modules are ABAP routines that are stored in a central function library. They are not application-specific, and available systemwide. The ABAP Workbench comes with a large number of standard function modules.

Like form routines, function modules encapsulate program code, and provide an interface for data exchange.

However, there are significant differences between function modules and form routines:

- Function modules must belong to a pool called a function group.
- They possess a fixed interface for data exchange. This makes it easier for you to pass input and output parameters to and from the function module.
  For example, you can assign default values to the input parameters. The interface also supports exception handling. This allows you to catch errors and pass them back to the calling program for handling.
- They use their own memory area. The calling program and the function module cannot exchange data using a shared memory area - they must use the function module interface. This avoids unpleasant side effects such as accidentally overwriting data.
- You call a function module by its name (which must be unique) in a CALL FUNCTION statement.

The Function Builder allows you to develop, test, and document new function modules. You can also use it to display information about existing function modules:

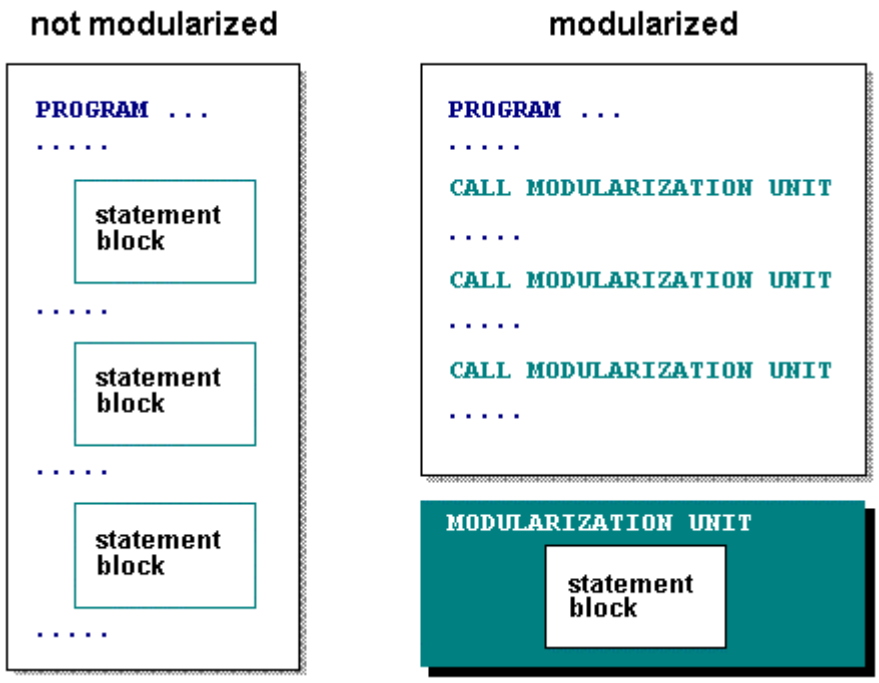
Administration	Specifies information like the development class a function belongs to, the person responsible for the module, a short description of the module.
Import	Contains a list of the formal parameters that are used to pass data to a function module. For further information, refer to Displaying Information about Interface Parameters [Page 417]
Export	Contains a list of the formal parameters that are used to receive data from a function module. For further information, refer to Displaying Information about Interface Parameters [Page 417]
Changing	Contains a list of the formal parameters that are used both to pass data to and receive data from a function module. For further information, refer to Displaying Information about Interface Parameters [Page 417]
Tables	Specifies the tables that are to be passed to a function module. Table parameters are always passed by reference. For further information, refer to Displaying Information about Interface Parameters [Page 417]
Exceptions	Shows how the function module reacts to exceptions. For further information, refer to Displaying Information about Interface Parameters [Page 417]
Documentation	Provides information about the interface and exceptions
Source code	Program code of the function module
Global data	The global data used by the function module.

Main program	Program code of the main program.

Function modules play an important role in the modularization of applications. You can use them to encapsulate a particular function or group of related functions.

Modularization allows you to avoid redundancy. It also makes your programs easier to read and understand.
Modularized programs are easier to maintain and keep up-to-date.

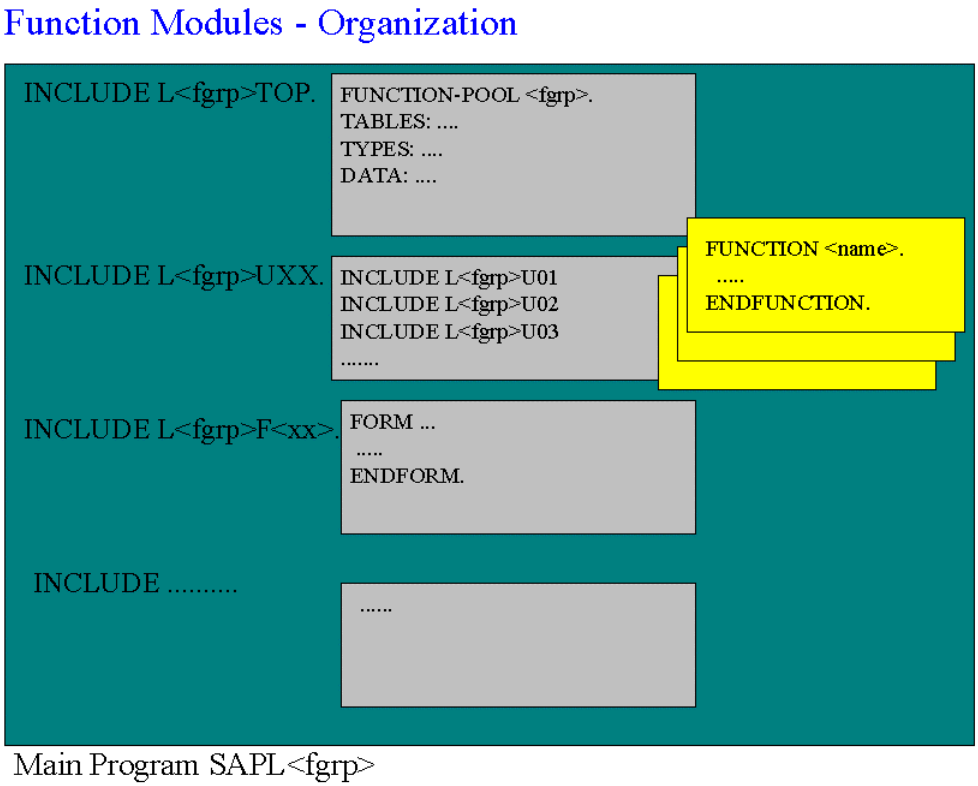
The modularization principle:



Function Groups

The Function Builder administers function modules that logically belong together in function groups. Function groups are containers for function modules. They can also contain global data declarations and subroutines that are available to all of the function modules in the group.

The following illustration shows the organization of function modules within a function group:



For each function group <fgrp> there is a main program, generated by the system, called SAPL<fgrp>.
The main program contains INCLUDE statements for the following programs:

- L<fgrp>TOP. This contains the global data for the function group.
- L<fgrp>UXX. These includes contain the function modules themselves. The numbering XX indicates the chronological order in which the function modules were created. This includes L<fgrp>U01 and L<fgrp>U02 contain the first two function modules in the function group.
- L<fgrp>F01, L<fgrp>F02... These includes can be used to write subroutines (forms) that can be called as internal forms by all function modules in the group.

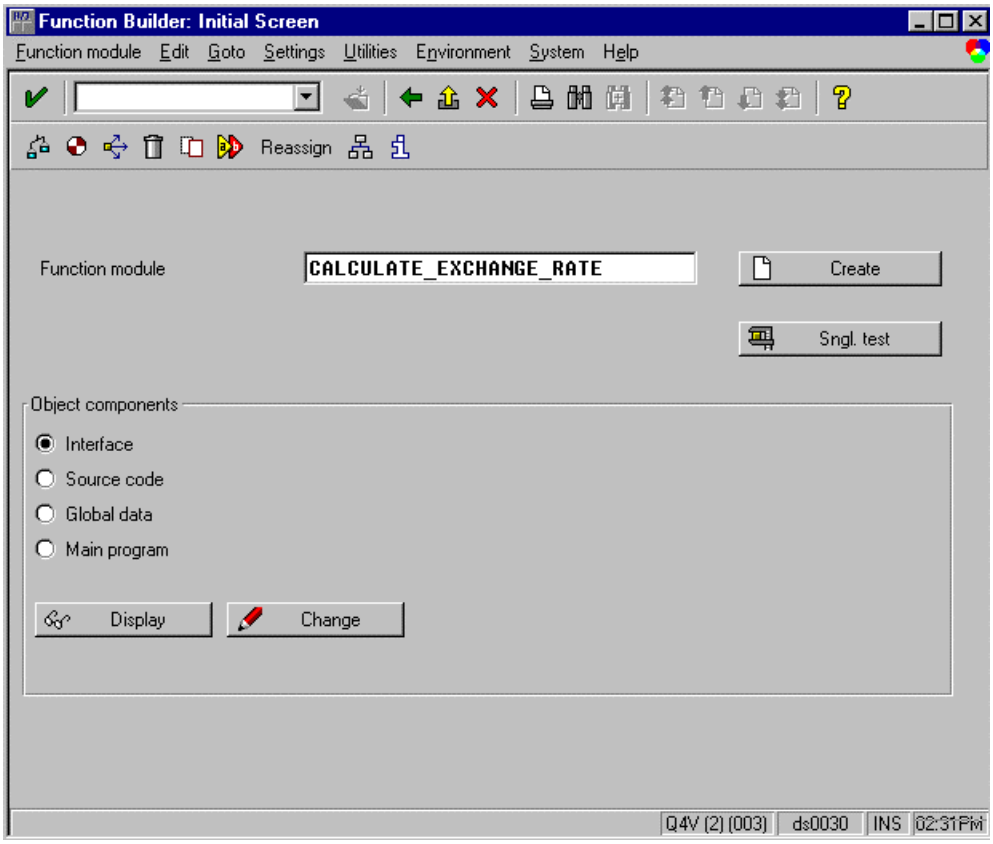
Displaying a Function Group

To display a function group, choose *Goto* → *Function groups* → *Display group*. A dialog box appears in which you can enter the name of the function group.

Initial Screen of the Function Builder

1. Start the ABAP Workbench.
2. Choose *Function Builder.*

   The Function Builder initial screen appears:



3. Enter the name of the function module that you want to create, change, display, or test. You can select any of the following components:

Interface	Lists the interface parameters (import, export, tables, exceptions), attributes, and parameter documentation. If you are working in change mode, you can extend the parameter list.
Source code	Displays the source code between the `FUNCTION` and `ENDFUNCTION` statements.
Global data	Displays the TOP include of the function group, containing the global data declarations.
Main program	Displays the main program with its list of includes.

4. Select one of the above components.
5. Choose *Display* or *Change*.

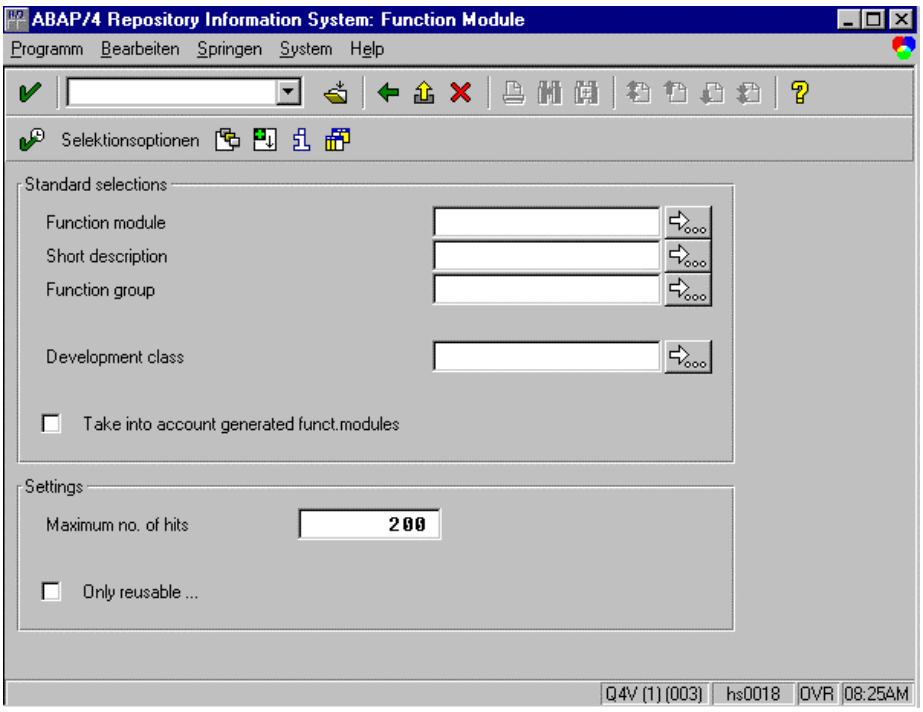
Looking Up Function Modules

Before creating a new application, you can search in the Function Builder to see if suitable function modules already exist. You can do this in the Repository Information System or in the Application Hierarchy.

Using the Repository Information System

To search for a module, choose *Find* from the initial screen of the Function Builder. The system displays the standard *Function Module* search screen.

The Repository Information System's search screen offers you a number of selection options. Only some of these options are displayed when you first call the screen. To view the rest of the selection options, choose *Edit* → *All selections* :



Enter either a function group or a development class for a quick search. You can also limit your selection to function modules of a particular type. You can search for remote function call (RFC) modules or for those that are used in update routines.

You can also generate a list of only those function modules that are released for customers. For more information about searching with the Repository, see The Repository Information System [Page 49 .

Using the Application Hierarchy

You can also search for function modules using the Workbench's Application Hierarchy. The Application Hierarchy provides an overview of all the applications in your R/3 system. You can use this hierarchy to display function modules associated with particular applications.

The Application Hierarchy [Page 502]

Looking Up Function Modules

1. In the *Function module* field of the initial screen of the Function Builder, or in the *CALL FUNCTION* field of the insert statement dialog box, press F4.
2. Choose SAP applications in the *Input Help Personal Values List* dialog box.
   The system branches to the SAP application hierarchy.
3. Open the application hierarchy down to the lowest level, and double-click a function module to choose it.
   The system jumps back to the point where you started your search, and inserts the name of the function module in the corresponding field.

Getting Information about Interface Parameters

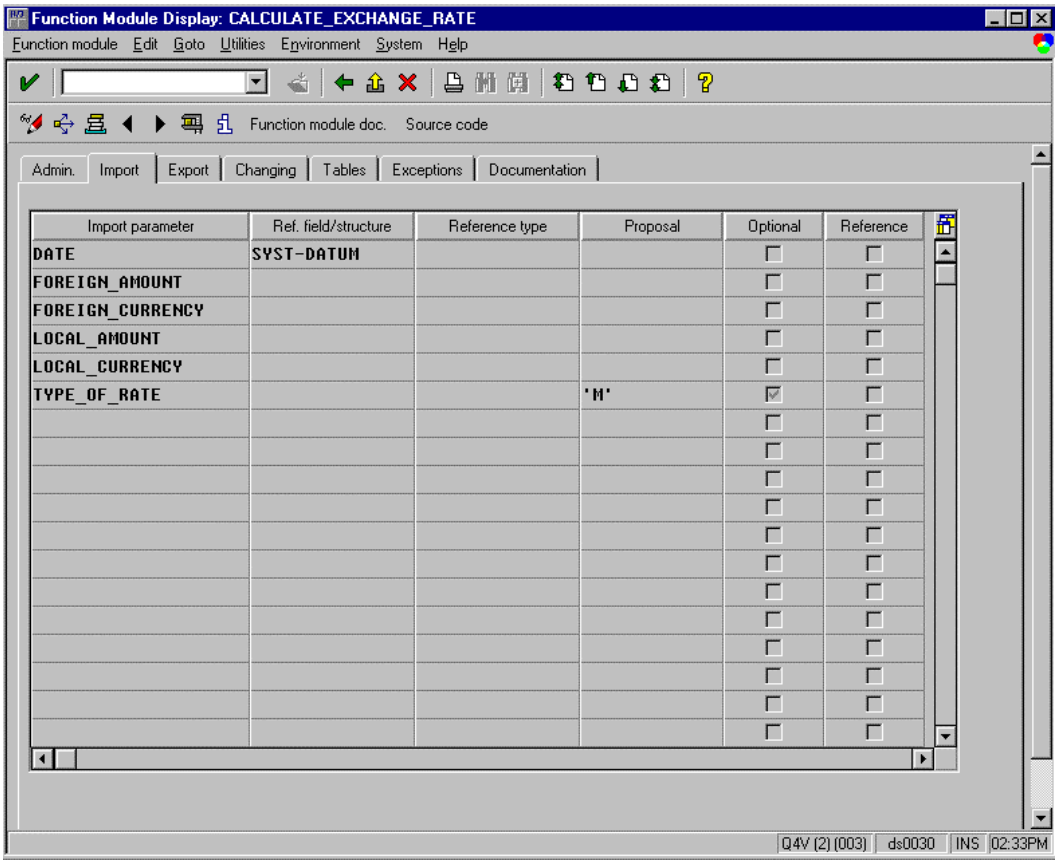
A function module's interface determines how you can use the module from within your own program. It is important that you understand a module's programming interface before you use the module. There are five different interface parameters:

Name	Explanation
Import	Values transferred from the calling program to the function module. You cannot overwrite the contents of import parameters at runtime.
Export	Values transferred from the function module back to the calling program.
Changing	Values that act as import and export parameters simultaneously. The original value of a changing parameter is transferred from the calling program to the function module. The function module can alter the initial value and send it back to the calling program.
Tables	Internal tables that can be imported and exported. The internal table's contents are transferred from the calling program to the function module. The function module can alter the contents of the internal table and then send it back to the calling program. Tables are always passed by reference.
Exceptions	Error situations that can occur within the function module. The calling program uses exceptions to find out if an error has occurred in the function module. It can then react accordingly.

To find out what parameters are needed to call a function module, enter the module's name in the initial screen of the Function Builder and display the object component *Interface*:

The following display uses a tab, with separate pages for various parts of the interface information (administration, formal parameters, exceptions, and documentation). Our example contains a list of all import parameters and their further attributes:

Getting Information about Interface Parameters



Particularly important here is the information about which parameters must be passed when the function module is called. If a parameter does not have to be passed, select the *Optional* checkbox.
If the parameter does have to be passed, the checkbox is deselected.

Choose *Changing* or *Tables* to see the changing and tables parameters and whether they are optional or required.
There is no *Optional* checkbox for export parameters, since they are always optional.
You can also specify the data type of a formal parameter by linking it to a data type in a type pool. Type pools are ABAP Dictionary objects that allow you to define your own global types. If you want to use the types in a type pool for formal parameters, you must declare the type pool in the TOP include of the function group. You can then enter the types in the *Reference type* field for formal parameters.
In the *Reference field/structure* field, you can enter ABAP Dictionary reference structures, against which the system checks the actual parameters at runtime.
Use the *Default* field to assign a default value to a parameter. In the *Reference* field, you can indicate whether you want to pass the parameter by reference or by value.

For more information about setting the attributes of parameters, see Specifying Parameters and Exceptions [Page 428]

The *Documentation* tab page contains short descriptions of the parameters and exceptions. To display further information about a particular parameter or exception, double-click its name.

For a full description of the task of the function module, double-click the *Short text* field, or choose *Function module doc.*

Displaying Function Module Attributes

The *Administration* feature of the Function Builder shows you a function module's attributes. Administration information includes the function module's:

- Function Group
- Process type
- status

If you want to print out all a module's interface information, choose *Function module* → *Print*. This option lets you specify the aspects of the function module (documentation, code, and so on) you want.

Calling Function Modules From Your Programs

You can call a function module from within any ABAP program by using the following statement:

```
CALL FUNCTION <function module>
     [EXPORTING  f1 = a1.... fn = an]
     [IMPORTING  f1 = a1.... fn = an]
     [CHANGING   f1 = a1.... fn = an]
     [TABLES     f1 = a1.... fn = an]
     [EXCEPTIONS e1 = r1.... en = rn
                 [ERROR_MESSAGE = rE]
                 [OTHERS = ro]].
```

You enter the name of the function module as a string. You pass parameters by assigning the actual parameters to the formal parameters in lists following the **EXPORTING**, **IMPORTING**, **CHANGING**, and **TABLES** options.

You assign parameters in the form *<formal parameter> = <actual parameter>*
If you assign more than one parameter within an option, separate them with spaces (or by starting a new line).

- The EXPORTING options passes the actual parameter a_i to the formal parameter f_i. The formal parameters must be declared as **import** parameters in the function module. The parameters may have any data type. If you specify a reference field, the system checks the parameter.
- The IMPORTING option passes the formal output parameter f_i of the function module to the actual parameter a_i. The formal parameters must be declared as **export** parameters in the function module. The parameters may have any data type.
- The CHANGING options passes the actual parameters a_i to the formal parameters f_i. After the function module has been processed, the system returns the (changed) values of the formal parameters f_i to the actual parameters f_i. The formal parameters must be declared as **changing** parameters in the function module. The parameters may have any data type.
- The TABLES option passes internal tables between the actual and formal parameters. They are always passed by reference. The parameters in this option must always be internal tables.
- The EXCEPTIONS option contains a list of special parameters that allow you to react to errors in the function module. When an exception occurs, the function module processing terminates. To take a concrete example: If exception e_i is triggered, the system stops processing the function module and does not pass any values back to the program. The calling program receives the exception e_i by assigning the value r_i as a return code to the system field SY-SUBRC. r_i must be a numeric literal. You can then evaluate the contents of SY-SUBRC in the calling program.

  You can change the error handling in the function module by specifying an ERROR_MESSAGE in the EXCEPTIONS list. Normally, you should only call messages

in function modules using the exception handling method (using the MESSAGE... RAISING or RAISE statements within the function module). For more information, see Understanding Function Module Code [Page 430]
When you use ERROR_MESSAGE, the system treats the message that are called without explicit handling as follows:

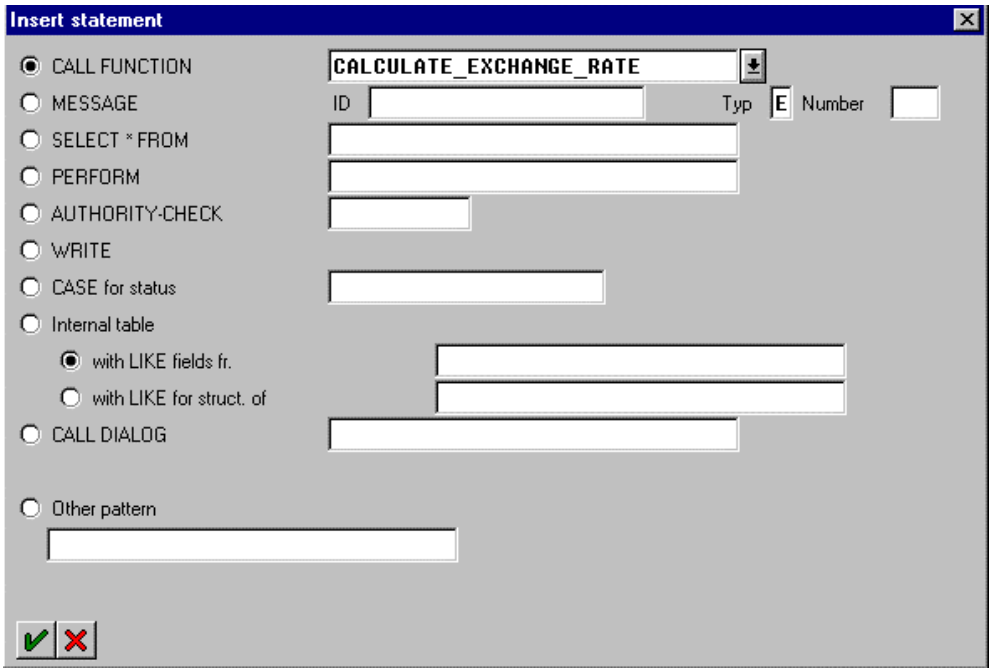
- Messages with type S, I and W are ignored (but entered in the log if you are running the program in the background).
- Messages with type E and A cause the function module to terminate as though the ERROR_MESSAGE exception had been triggered (SY-SUBRC is set to r_E).

If you enter OTHERS in the exception list, you can allow for all exceptions, even though they are not listed. It acts as a default exception.

You can use the same number r_i for different exceptions, as long as the exceptions do not have to be more specific.

You can call a function module from an ABAP program by using the *Insert Statement* function in the ABAP Editor as follows:

1. Position the cursor at the point where you want to insert the CALL FUNCTION statement.
2. Choose *Pattern*.
3. In the *Insert Statement* dialog box, select CALL FUNCTION.



4. Enter the name of the function module.

   If you do not know the name, you can search using the possible values help.

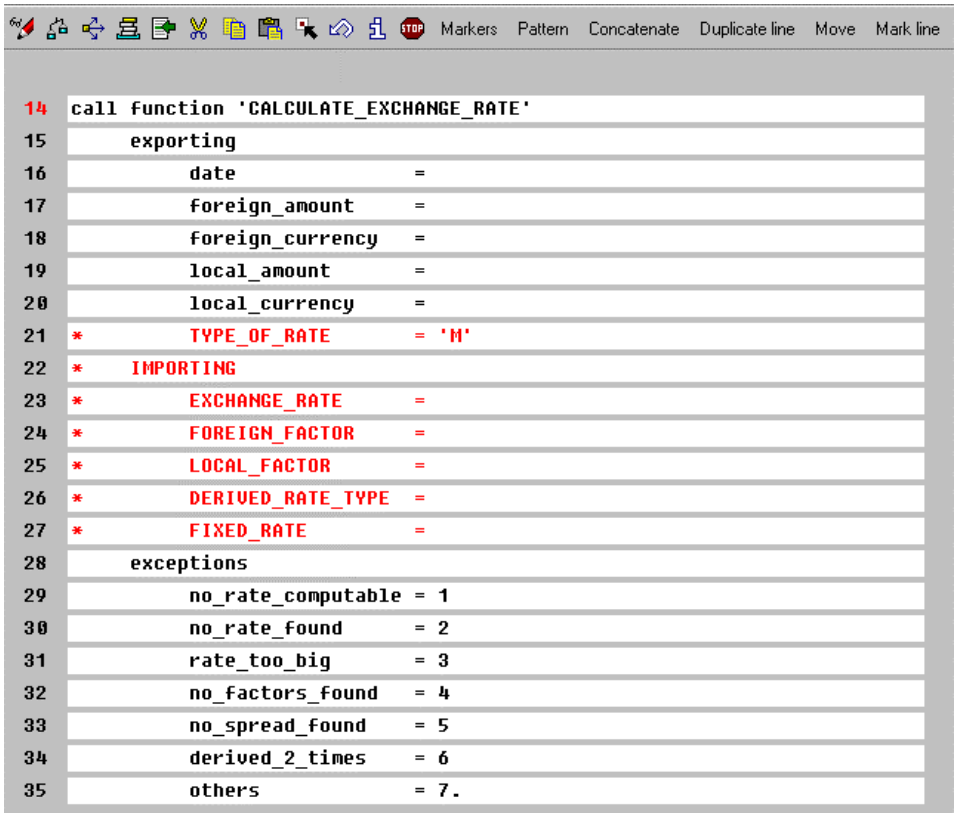
5. Choose *Continue*.

   The system inserts the CALL FUNCTION statement, complete with the interface of your chosen function module.

6. Add the parameters, and program any exception handling.

   In our example, the function module is inserted into the ABAP Editor as follows:



7. If you need more information about the function module, you can use the ABAP Editor help (F1). The Help dialog box appears. Select *Function module*, and enter the function module name. Choose *Continue*.
   The system displays the interface definition for the function module. From there, you can display all of the other function module elements.

   You must enter values for any parameters do not appear as comment lines in the ABAP Editor (constants or parameters).

The importing interface is commented out. Remove the asterisks (*) before the IMPORTING addition and the relevant parameters, and enter variables against the parameters from which you want to receive values from the function module.

There are other parameters that you can use with the CALL FUNCTION statement if the function is running in the update task or on a remote host.

When a function module runs in the update task, the system processes it asynchronously. Instead of processing it straight away, the system waits for the next database update to be triggered by a COMMIT WORK statement. Running a function module on a remote host means that you call a function module within another SAP system or a non-SAP system.

For further information about how to call function modules from programs, see the *Function modules* section of the ABAP User's Guide [Ext.]

Creating new Function Modules

Function modules perform tasks of general interest to other programmers. Usually these tasks are well-defined functions that all users need, regardless of application. Some well-defined tasks include performing tax calculations, determining factory calendar dates, and calling frequently-used dialogs.

When you write ABAP routines that other programmers might use, you should define these routines as function modules. This means that you develop them in the Function Builder as follows:

1. Check whether a suitable function module already exists. If not, proceed to step 2.
2. Create a function group, if no appropriate group exists yet.
3. Create the function module.
4. Define the function module interface by entering its parameters and exceptions.
5. Write the actual ABAP code for the function module, adding any relevant global data to the TOP include.
6. Activate the module.
7. Test the module.
8. Document the module and its parameters for other users.
9. Release the module for general use.

Runtime Considerations

There are some runtime considerations you should be familiar with when writing function modules:

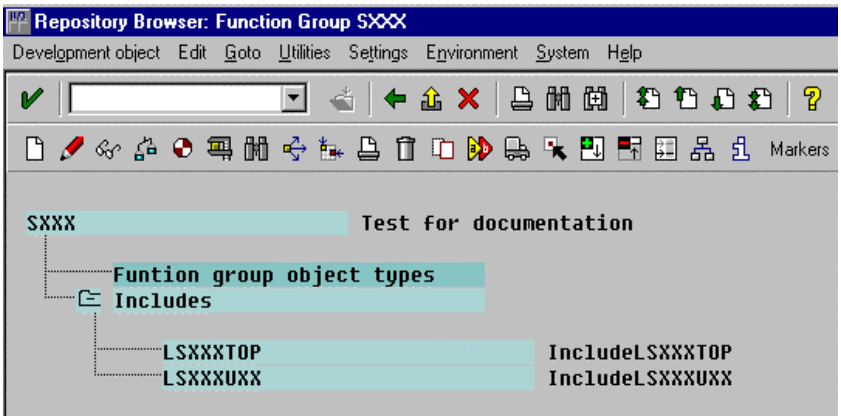
- The CALL FUNCTION statement can pass import, export, and changing parameters either by value or by reference. Table parameters are always transferred by reference.
- If you declare the parameters with reference to ABAP Dictionary fields or structures, the system checks the type and length when the parameters are transferred. If the parameters from the calling program do not pass this check, the calling program terminates.
- At runtime, all function modules belonging to a function group are loaded with the calling program. As a result, you should plan carefully which functions really belong in a group and which do not. Otherwise, calling your function modules will unnecessarily increase the amount of memory required by the user.

Creating a Function Group

1. Choose *Goto* → *Function groups* → *Create group*.
2. Specify the function group name and a short text.
3. Choose *Save*.

   Function group names are freely definable up to a maximum length of 26 alphanumeric characters. Remember to observe the normal naming conventions for the first character (A-X for SAP development, Y and Z for customers).

When you create a new function group, the system automatically creates a *main program* containing two includes. Like any other programs and includes, you can display them in the Repository Browser.

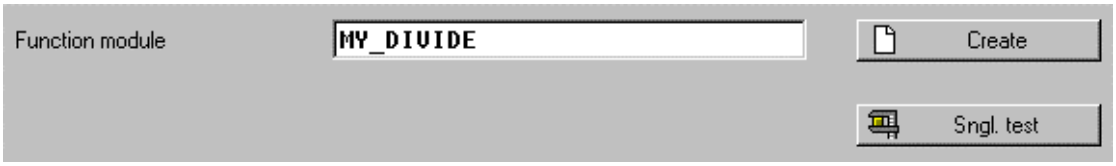


The name of the main program is assigned by the system. This is made up of the prefix **SAPL** followed by the function group name. `For example, the main program for function group SXXX is called SAPLSXXX.`

The names of the include files begin with **L** followed by the name of the function group, and conclude with **UXX** (or **TOP** for the **TOP** include). The TOP include contains *global data* declarations that are used by all of the function modules in the function group. The other include file within the main program is used to hold the function modules within the group.

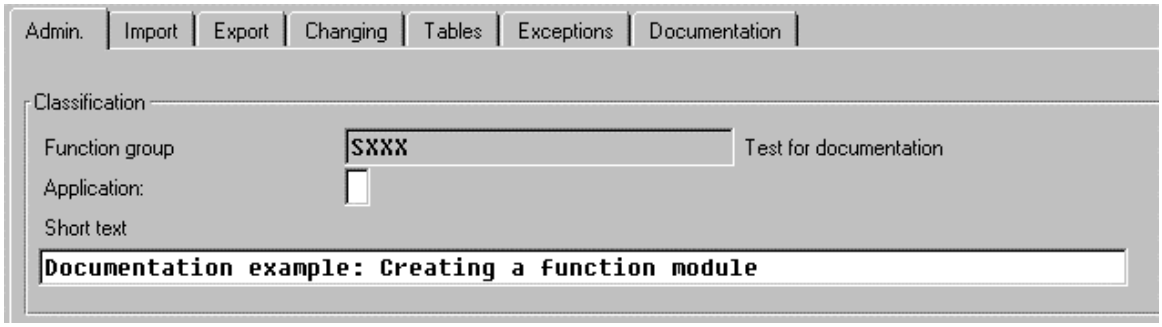
Creating a Function Module

1. Enter the function's name in the field *Function module.*
2. Choose *Create*.



3. In the *Enter function group* dialog box, enter the function group to which you want to assign the function module.

   The *Administration* page of the *Create function module* screen appears.



4. Enter the following attributes

Attribute	Explanation
Application	Specifies the function's application group. Leave the field blank ("For all applications") if your function is not application-specific.
Short text	Describes the function module.
Process type	Specifies the function type. Choose *Normal* unless you are writing a function to be run in remote systems or in an update task.

General data	Lists general administration information

5. `Choose` *Save*.

   The Workbench automatically creates an include file for your function module, whose name is eight letters long. For example, for the first function module in the function group FGRP, the include file is called LFGRPU01. The next function modules will have include files LFGRPU02, LFGRPU03, LFGRPU04, and so on.

For further information about the include programs, see Understanding Function Module Code [Page 430]

Specifying Parameters and Exceptions

The parameters and exceptions for a function module constitute its interface. The Function Builder contains a tab page for each of the following interface components: tables, exceptions, import, export, and changing parameters.

To set the parameters and exceptions of your function module:

1. Enter the function module name on the initial screen of the Function Builder.
2. Select *Interface* and choose *Change*.
3. Enter any further required information for the parameters (import, changing, export, or table).

Field	Explanation
Parameter	Name of the formal parameter, for identification
Reference field/ structure	A database field, component of an ABAP Dictionary structure, or an entire ABAP Dictionary structure. This is the same as the ABAP Dictionary field name in the Reference field/ reference structure column. Use this field to create a field based on an ABAP Dictionary field. You should always use a reference structure if the data in the parameter must have the same structure as the reference field (for example, when you want to add new entries to the database).
Reference type	You can enter any system type, either generic or fully typed. For further information, refer to the Data Types [Ext.] section of the ABAP User's Guide.
Default	This is the parameter's default value. Applies to import and changing parameters only. The system transfers this value to the function module if the caller sets its own value for that parameter.
Reference	The parameter reference. Specify this if you want the parameter to be called by reference instead of by value. When a parameter is called by reference, the system points to the original parameter without making a copy of it. The function module works with and, if necessary, alters the original parameter and not a copy. Table parameters are always passed by reference.

4. Enter the exceptions.

   The exceptions screen only allows you to enter a text with which the exception can be triggered in the function module.

5. You can also document the interface on this screen.

   On the *documentation* screen, you can enter short descriptions of the parameters and exceptions.
   You can also write full documentation of the entire function module from here.
   For further information, see Documenting and Releasing Function Modules [Page 441]

6. Save your entries.

Understanding Function Module Code

The system predefines the organization of objects for a function group and its function modules. When you create a function group, the Workbench automatically generates a main program, global data, and source code. The system uses the format **SAPL<fgrp>** to name the *Main program*. The **<fgrp>** variable is the function group's name.

For each successive function module in a function group, the Workbench automatically creates an include file. You can view this include file by selecting *Source Code* on the Function Builder initial screen. The system gives the include file a name using the form **L<functgrp>U<nn>**. For example, in the function group FGRP, the first function module resides in include file LFGRPU01. The subsequent function modules are in include files LFGRPU02, LFGRPU03, LFGRPU04, and so on.

The Main Function Program

The Function Builder generates the main function program includes for you. The system uses the **L<functgrp>UXX** form to name a main function program. So, for the function group FGRP, the main functions include would be **LFGRPUXX**.

Writing Function Modules

Once you have defined the interface of your function module, you can start writing the code itself.

On the initial screen of the Function Builder, select *Source code*. The ABAP Editor appears, in which you can write the code of the function module between the FUNCTION and ENDFUNCTION statements.
The parameters and exceptions of the function module appear in the Editor as commented lines.

```
1  function my_divide.
2  *"----------------------------------------------------
3  *"*"Local interface:
4  *"       IMPORTING
5  *"             VALUE(Z1) TYPE  F
6  *"             VALUE(N1) TYPE  F
7  *"       EXPORTING
8  *"             VALUE(RES) TYPE  F
9  *"       EXCEPTIONS
10 *"              DIV_ZERO
11 *"----------------------------------------------------
12
13
14
15 endfunction.
```

A few features to bear in mind when writing function modules:

Data Handling in Function Modules

- You do not declare export and import parameters in the source code of the function module. The system does this automatically, using an INCLUDE program that inserts a list of the parameters as comment lines in the source code.
- You can declare local data types and objects in function modules in the same way that you would in subroutines.
- You can declare data using the TYPES and DATA statements in L<fgrp>TOP. This data is then available to all of the function modules in a function group. The system creates the data the first time a function module in the group is called. It always saves the values from the last function module to be called.

Calling Subroutines from Function Modules

You can call various subroutines from function modules.

- You can write internal subroutines, adding them after the ENDFUNCTION statement. These subroutines can be called from all function modules in the group. However, we recommend that you only call them from the function module in which you wrote the function module. This makes your function group easier to understand.
- If you want to create internal subroutines and call them from all of the function modules in the function group <fgrp>, use the special INCLUDE programs L<fgrp>F<XX>.
- You can call any external subroutines from a function module.

Triggering Exceptions

Within a function module, you can address all exceptions using the names you defined in the interface. Exceptions can be handled either by the system or by the calling program. You decide this when you call the function, by assigning a numeric value to the exceptions that you want to handle yourself. For further information, see Calling Function Modules From Your Programs [Page 420].

Exceptions must be explicitly triggered.
There are two ABAP statements that may only be used in function modules that you can use to trigger exceptions:

Syntax

```
RAISE <Exception>.

MESSAGE..... RAISING <Exception>.
```

The effect of these statements depends on whether you handle the exception in the calling program or let the system process it.

- If you trigger the exception in the RAISE statement and the calling program is to handle it, the function module processing is terminated, and the numeric value assigned to the exception is placed in the system field SY-SUBRC. Further processing then takes place in the calling program.

  If the calling program fails to handle the exception, the system triggers a runtime error.
- If you use the MESSAGE... RAISING statement, the processing is similar if you want to handle the exception in the calling program. If you want the system to handle the

exception, there is no runtime error generated in this case. Instead, processing continues, and the system displays a message with the defined type. To do this, you must specify the MESSAGE-ID in the first statement of the include program L<fgrp>TOP. The MESSAGE... RAISING statement also enters values in the following system fields:

- SY-MSGID (message ID)
- SY-MSGTY (message type)
- SY-MSGNO (message number)
- SY-MSGV1 to SY-MSGV4 (contents of the fields <f1> to <f4> that are included in the message).

For further information, see the keyword documentation for the MESSAGE statement.

Here is an example for raising exceptions:

Suppose we have the following function module:

```
1  function my_divide.
2  *"----------------------------------------------------------
3  *"*"Local interface:
4  *"       IMPORTING
5  *"             VALUE(Z1) TYPE  F
6  *"             VALUE(N1) TYPE  F
7  *"       EXPORTING
8  *"             VALUE(RES) TYPE  F
9  *"       EXCEPTIONS
10 *"              DIV_ZERO
11 *"----------------------------------------------------------
12   if n1 eq 0.
13     raise div_zero.
14   else.
15     res = z1 / n1.
16   endif.
17 endfunction.
```

If N1 is unequal to zero, it divides Z1 by N1. Otherwise, it triggers the exception DIV_ZERO.

Example: Program MDTEST calls the function MY_DIVIDE:

```
 8
 9 report  mdtest.
10
11 data: result type f.
12
13 call function 'MY_DIVIDE'
14      exporting
15           z1   = 6
16           n1   = 4
17      importing
18           res  = result
19      exceptions
20           div_zero = 1
21           others   = 2.
22
23 if sy-subrc eq 0.
24       write: / 'Result =', result.
25 else.
26       write 'Division by zero'.
27 endif.
```

When you run the program, the output looks like this:

```
Result = 1,500000
```

If you replace N1 = 4 with N1 = 0 in the EXPORTING list, the program MDTEST processes the exception DIV_ZERO by assigning the value 1 to SY-SUBRC. The output now looks like this:

```
Division by zero
```

Checking and Activating Modules

Before you can activate a function module, you must check to make certain that the module's syntax is correct. Open one of the module's interface screens and use *Function module → Check* to check your module. Checking Source Code [Page 111] explains in detail how to use this feature.

Newly created function modules are automatically set to *Inactive*.

If a function module is inactive, the normal syntax check only checks that module. If you want to check the module as part of the whole function group without activating it, choose *Function module → Check → Main program*. The check program checks all of the function modules and include programs regardless of whether they are active or inactive.

To activate a completed function, choose *Function module → Activate*. An activated function module is included in all syntax-checking for the module's entire function group. When the system checks an activated function module's syntax, the system actually applies the check to all activated members of the function group. When you activate a function module, syntax-checking is performed automatically.

See also:

Inactive sources [Page 509] in the ABAP Workbench.

Restoring the Active Version of a Function Module

You can replace the inactive version of a function module with its last active version by choosing *Function module → Return to active version*. The inactive version is deleted.

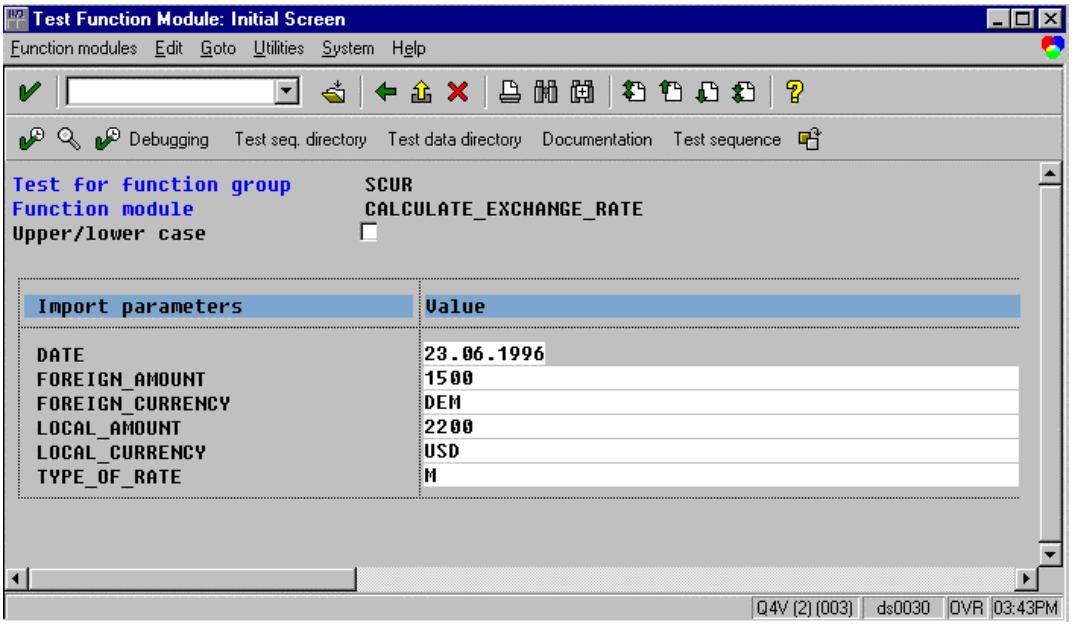
Testing Function Modules

Debugger [Page 444]

You should use the test environment in the Function Builder to test new function modules before releasing them for general use. You can also use the test environment to examine functions written by other developers before calling these modules from within your own programs. The library's testing options let you determine if a function performs as it should and shows you if the module returns expected results. To run a test:

1. Choose *Test* from the Function Builder initial screen.

   The *Test Function Module* screen appears. It displays all of the import and changing parameters of the function module.



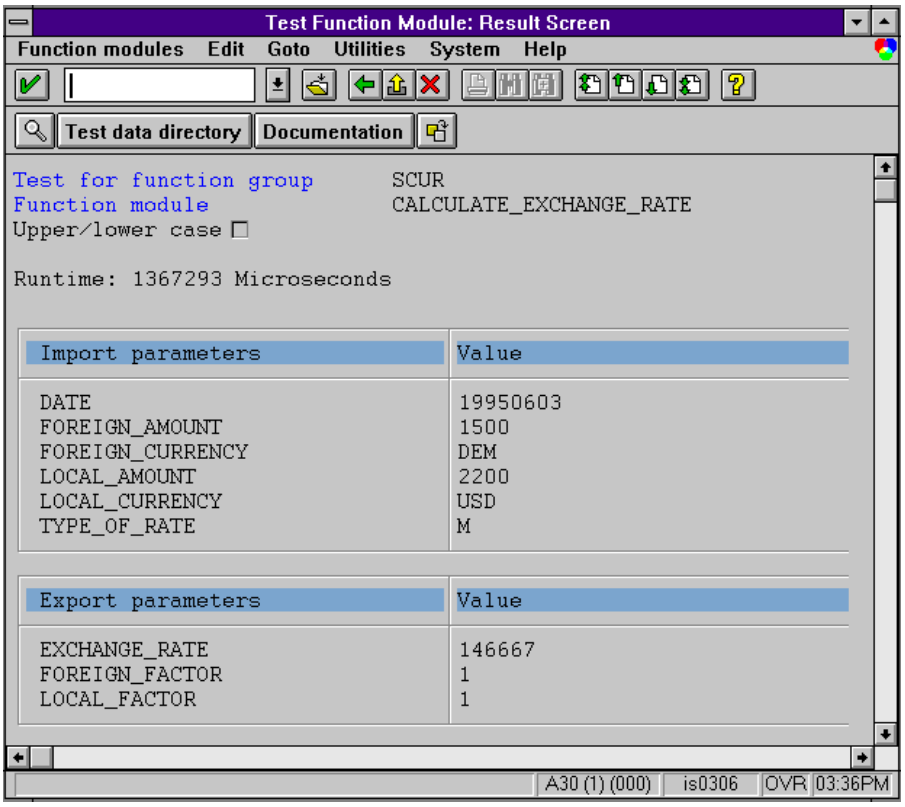
2. Specify the data you want to transfer from your program to the function module.

   Fill in values for the relevant import, changing, and tables parameters. To fill in single-field parameters, enter the value in the displayed field. To fill in table/ structure parameters, double-click on the parameter name.

3. Choose *Execute*.

   The system runs the function module using your input and displays the values of the export parameters that result:

Testing Function Modules



When you test a function module, the system displays any exceptions. The system also identifies the time required to execute the module in microseconds. This is an elapsed-time measurement. The measurement includes interrupt time as well as processing time. You should view this elapsed-time as an estimate only.

Other Test Options

The Function Builder contains more test options than simply running a function module. You can also run a test in the debugger or create a runtime analysis performance file for a function. To investigate a function module in debugging mode:

1. Choose *Test* from the Function Builder initial screen.
2. Specify the data you want to transfer from your program to the function module.
3. Select *Debugging*.

   The system executes the function module in debugging mode. You can step through the function's code and use all the options offered within the Debugger

To test a function module's performance, select *Runtime analysis* (Transaction SE30) from the *Test Environment for Function Modules* screen. The system executes the function module and records the function's performance in a special performance data file. Select *Eval.rtime analysis* to display the results of the analysis.

Saving Tests and Test Sequences

The Function Builder offers you several further options that can be helpful when you are developing your function. These options include:

- Saving test runs.
- Displaying old tests and their results.
- Comparing previous test results with results from a fresh test run.
- Composing a test sequence for repeated testing.

Saving a Test Run

execute the test and then select the *Save* icon. Enter a short description of the test so that you can identify it better later on.

Displaying and Rerunning Old Tests

To display old tests that you have saved, enter the test screen in the Function Library and choose *Test data directory*. You can view either the test parameters or the actual results.

Rerunning old tests shows you if the changes you have made in your function module affect the data received by the calling program. To rerun a test using the same parameters:

1. Choose *Test* from the Function Builder initial screen.

   The system displays the test environment.
2. Choose *Test data directory*.
3. Place the cursor on the test you want to reexamine.
4. Choose *Test.*

   The system immediately reruns the same test and displays any differences in the results.

Using Test Sequences

1. Open the function module test environment.
2. Choose *Goto* → *Test sequence*.
3. Run a series of tests as you would normally, entering relevant parameters and selecting *Execute*.

   You can re-run the same function module several times or test different, related function modules. For example, you can test a module that creates a new table record and then test one that deletes the same record. If you want to test a series of function modules, choose *Function modules* → *Other FM* after each test.
4. End the sequence by choosing *Edit* → *New sequence*.

   The system lets you save your test sequence.

To display an existing sequence, choose *Test seq. directory* to display the sequence. You can also run a test sequence by selecting *Edit* → *Enter sequence*. The system provides a window where you can enter the names of all the modules you want to test. Select *Execute* and the system lets you run each module in test mode one after the other.

Documenting and Releasing a Function Module

You document your function modules in the Function Builder. There are two kinds of documentation - parameter documentation, and full function module documentation.

Interface Documentation

The parameter documentation must provide users with information about the different parameters and exceptions.

1. Open the Function Builder and enter the name of the function module that you want to document.
2. Select *Interface*.
3. Choose *Change*.
4. Choose the *Documentation* tab.
   A list of all parameters and exceptions appears.
5. Enter a short text for each entry.
6. Save your entries.

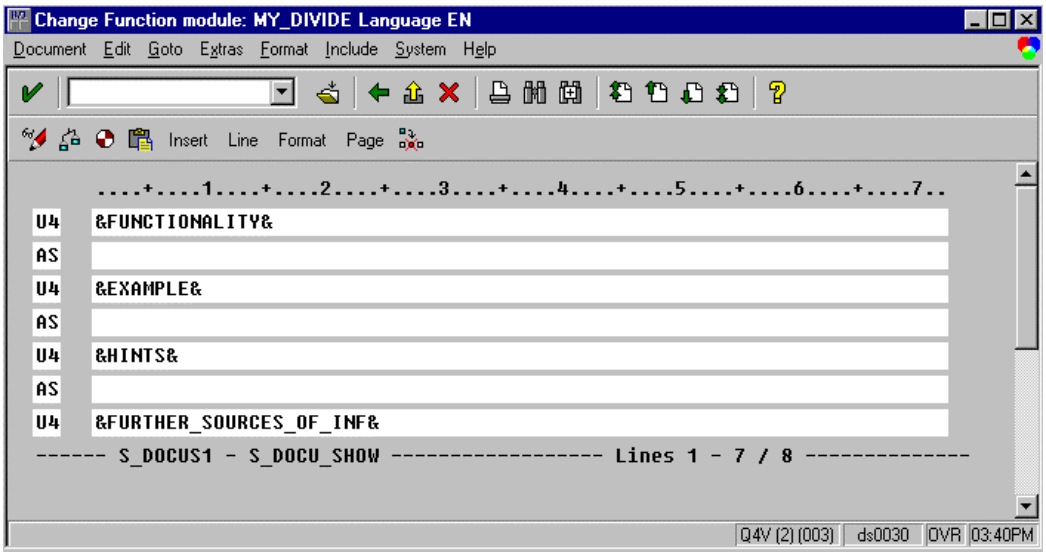
To add extra documentation for a parameter or exception, double-click the corresponding line on the *Documentation* page.
The long text editor appears. Enter your text and save it. You can now return to the function module maintenance screen.

Function Module Documentation

Function module documentation contains important detailed information about the task of the function module. It should be detailed enough for other users to be able to understand your function module without having to examine its source code.

To create the documentation:

1. Open the Function Builder and enter the name of the function module that you want to document.
2. Select *Interface*.
3. Choose *Change*.
4. Choose *Function module doc.*
   The system opens the SAPscript editor. Here, you can enter comprehensive function module documentation, including examples, tips for using the function module, and any other relevant information.



The SAPscript editor differs considerably from the ABAP Editor. Firstly, the menus and key settings are different. Secondly, you can use special formatting in SAPscript documentation. For further information about the SAPscript editor, see Text Processing with the SAPscript Editor [Ext.]

5. Save the documentation

Releasing a Function Module

Releasing a function module is a purely administrative gesture with no effect on the function or its usability. When you are satisfied that a function module is ready for general use, you can release it to the system. Releasing a function module signals that a developer has tested it.

Remember too, that when you release a function module, its documentation is released for translation and appears in the relevant translator's worklist. You should therefore only release the function module when you are sure that no more changes will occur. It is a good idea to inform your translator in good time that you are releasing your documentation.

Normally, only one person is responsible for a function group and only this user can release the module. Check the attributes for a function group to find out who is responsible for the group. Then, notify this person that your function module is ready for release. To release a function module, choose *Function module → Release options → Release.*

Releasing function modules was designed primarily for developers at SAP. By releasing a function internally, the SAP developer tells other developers at SAP that they can use this particular function module safely. SAP developers can also release their functions for customers. The precise function of an SAP function module may change but SAP ensures that the function remains backwards compatible.

Debugger

This documentation describes how to use the Debugger to find errors in the source code of an ABAP program.

Contents

Debugger

Releasing Database Locks [Ext.]

Settings and Warnings [Ext.]

Runtime Analysis

This documentation describes the runtime analysis tool in the ABAP Workbench. Runtime analysis shows you how long it takes to process ABAP code, from a single statement to a complete transaction.

To start the runtime analysis, choose *Tools* → *ABAP Workbench*, *Test* → *Runtime analysis* (or transaction **SE30**). . On the initial screen, you can choose one of the four main functions:

- Measurement in the current session
- Measurement in a new session
- Select measurement restrictions
- Analyze performance data

The following documentation explains how to perform an analysis, display and interpret the results, and use the information to optimize your program.

Contents

Runtime Analysis

Performance Trace

The performance trace tool contains a range of trace functions that you can use to monitor and analyze the performance of the system in database accesses, locking, and remote calls of reports and transactions.

Contents

Performance Trace

Performance Trace: Overview

Use

The Performance Trace allows you to record database access, locking activities, and remote calls of reports and transactions in a trace file and to display the performance log as a list. It also provides extensive support for analyzing individual trace records.

Integration

The Performance Trace is fully integrated in the ABAP Workbench, and can be called from the ABAP Workbench initial screen.

Prerequisites

To use the Performance Trace, you need authorization to start Transaction ST05 and the system authorizations “Change trace switches” (authorization STOM for authorization object S_ADMI_FCD) and “Analyze traces” (authorization STOR, also for authorization object S_ADMI_FCD).

Features

From Release 4.0B, the Performance Trace contains the following traces:

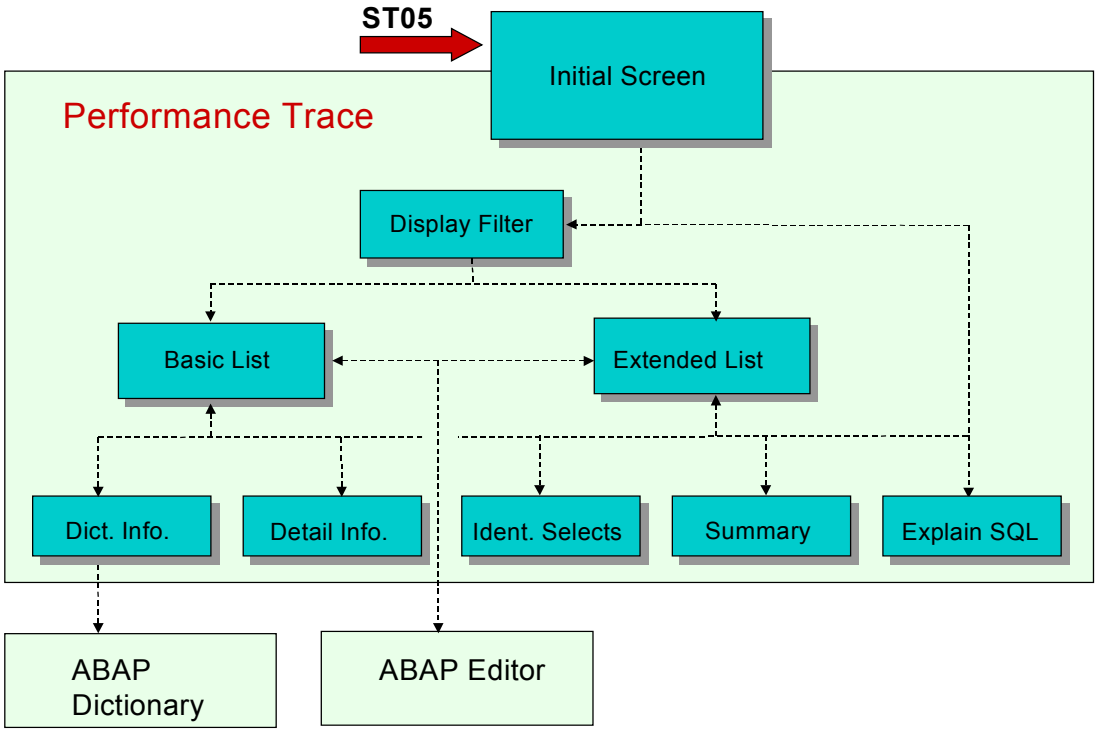
1. SQL Trace: This allows you to monitor the database access of reports and transactions.
   See also: SQL Trace Analysis [Page 469].
2. Enqueue Trace: This allows you to monitor the locking system.
   See also Enqueue Trace Analysis [Page 481].
3. RFC Trace: This provides information about Remote Function Calls between instances.
   See also RFC Trace Analysis [Page 484].

Activities

To start the Performance Trace, choose *Test* → *Performance Trace* from the initial screen of the ABAP Workbench (Transaction **ST05**).

Architecture and Navigation

The following diagram shows the architecture of the Performance Trace. It contains the most important components and the navigation options:



Navigation

On the initial screen, you can start and stop trace recording. The measurement results are saved in a trace file.

You can restrict the quantity of performance data displayed by setting a display filter.

You can display the performance data either as a *basic list* or in an *extended list*.

It is also possible to make the display more specific:

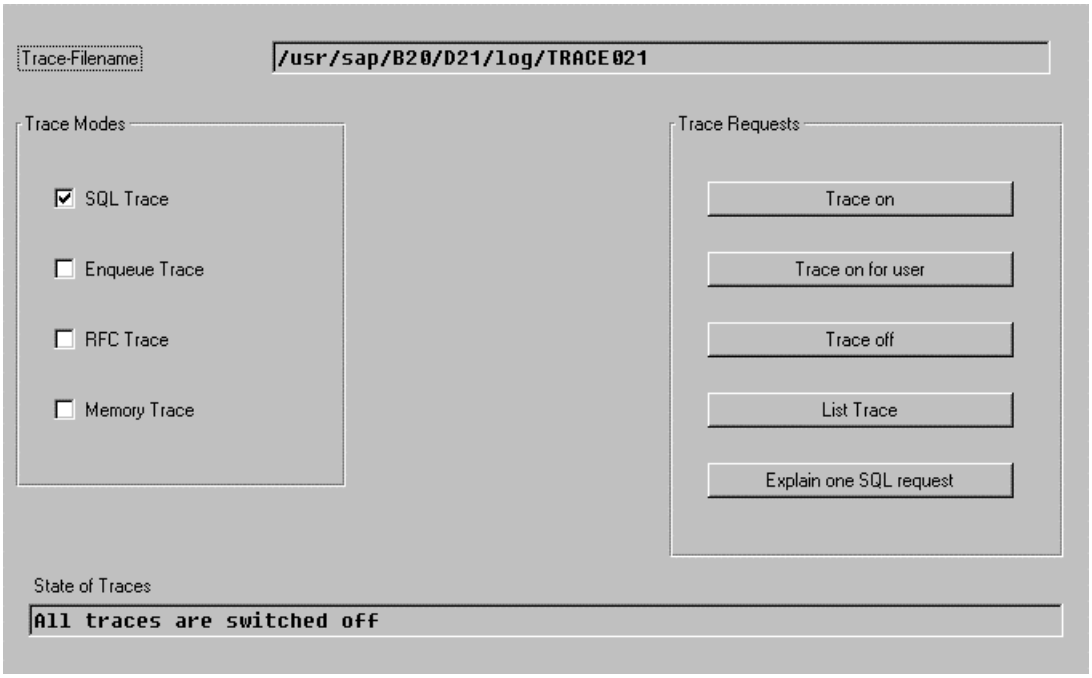
Use the *Dict. Info*. display to display information about ABAP Dictionary objects. You can also branch from this display to the object definition in the ABAP Dictionary. Use the *Detail info.* option to display information for the statement you want to analyze. Choose *Ident. selects* to display a list of identical select statements. Use the *Summary* function to summarize the trace list. The *Explain SQL* displays an execution plan for a selected SQL statement.

You can branch directly from the performance data for a statement to display the statement itself in the ABAP Editor.

Initial Screen

Access

You can start the test tool using Transaction **ST05** or by choosing *Test* → *Performance Trace* from the initial screen of the ABAP Workbench.



Functions

The initial screen contains the following functions:

- Trace on (starts recording)
- Trace off (stops recording)
- Trace function selection *SQL Trace, Enqueue Trace*, or *RFC Trace*
- Display the basic or extended list
- Start the *Explain SQL* to analyze an SQL statement or an explicit trace file.

Constraints

The *Memory Trace* function is not yet available.

Recording Performance Data

Preparation

By default and for performance reasons, the Performance Trace in the R/3 System is normally switched off. When you decide to analyze a report or transaction using the Performance Trace, you must first establish whether you want to analyze the interaction of reports and transactions, their effects on one another, the behavior of one or more individual reports and transactions, or only program sections.

Trace records are written into a **Ring file**. This means that trace records can be overwritten. When the file is full, recording continues from the beginning. It is therefore a good idea to log essential procedures separately. To avoid data being overwritten, you can also shorten the trace interval or enlarge the ring file.

See also: Configuring the trace File [Page 488]

Essential Information

Before you can record trace records, you must switch on the Performance Trace for an instance of the R/3 System.

Here, you can determine which trace functions (SQL Trace, Enqueue Trace, RFC Trace) you want to switch on, and for which user or users they should be activated.

You can then run the reports or transactions that you want to analyze, before switching the trace off again.

This process generates a trace file, containing all of the trace records, which you can then analyze, either immediately or later on. If you decide to repeat the trace or analyze the results later on, remember that the data in the trace file can be overwritten (see above).

See also

Starting the Trace [Page 455]

Stopping the Trace [Page 456]

Starting the Trace

Prerequisites

You can only switch on the Performance Trace for a single instance. You should already have decided the scope of your performance analysis.

Procedure

1. From the ABAP Workbench, choose *Test* → *Performance Trace*.
   The initial screen of the test tool appears. In the lower part of the screen, the status of the Performance Trace is displayed. This tells you whether any of the Performance Traces are switched on, the users for which they are enabled, and the user that switched it on.
2. On the initial screen, select the trace functions that you want to switch on (SQL Trace, Enqueue Trace, RFC Trace).
3. If you want to switch on the trace under your user name, choose *Trace on*.
   If you want to switch on the trace for another user or user group, choose *Trace on for user*. To enter a single user, specify the user name. To enter a user group, specify a search pattern (you can use the normal wildcards). If you want to change the user or user group, switch off the performance trace and then restart it, entering the new users or user group.
4. Run the program you want to analyze as normal.

You will normally analyze the performance trace file immediately. In this case, it is a good idea to use a separate session to start, stop, and analyze the Performance Trace.

During a Performance Trace interval (the time between switching the Performance Trace on and off), you can change the trace types at any time. The user (or user group) must remain unchanged during this period.

Result

The results of the trace recording are contained in a trace file. If trace records are overwritten during the trace interval, the system displays a message to inform you when you analyze the trace file.

The Performance Trace records all database access, remote calls, or lock activity. The measurement itself can affect the performance of the application server on which the trace is running. You should therefore switch off the trace [Page 456], as soon as you have finished with it.

Stopping the Trace

Prerequisites

You have switched on the Performance Trace and finished running the program that you want to analyze.

For performance reasons, you should switch of the trace as soon as you have finished recording.

Procedure

1. From the ABAP Workbench, choose *Test → Performance Trace.*
   The initial screen of the test tool appears. It contains a status line displaying the traces that are active, the users for whom they are active, and the user who activated them.
2. Select the trace functions that you want to switch off.
3. Choose *Trace off.*
   If you switched on the trace recording yourself, you can now switch it off immediately. If another user switched it on, a confirmation prompt appears.

Result

The results of the trace recording are stored in a trace file, which you can now analyze. For further information, refer to Analyzing Performance Data [Page 457].

Analyzing Performance Data

Prerequisites

Once you have switched off the performance trace, you can analyze the data. The data is available until its trace records are overwritten in the trace file. You can ensure that the file is not overwritten by changing the name of the trace file (see Configuring the Trace File [Page 488]) or saving the trace file at operating system level (see Saving Lists Locally [Page 490]).

Procedure: Overview

You must switch off the Performance Trace before analyzing the trace file. For further information, refer to Stopping the Trace [Page 456].

Before displaying the trace records, you can use a display filter to specify the records and information that you want to look at. For further information, refer to Display Filter [Page 458].

When you display the trace records, you can choose between a basic list and an extended list. Both lists display an overview of the logged actions and performance data. For further information, refer to Displaying Lists of Trace Records [Page 462].

Both lists contain the same range of functions for analyzing the statements and other performance data listed. For further information, refer to Analyzing Performance Data [Page 457].

Other analysis options depend on the trace types that you are using.

See also:
SQL-Trace [Page 469]

Enqueue-Trace [Page 481]

RFC-Trace [Page 484]

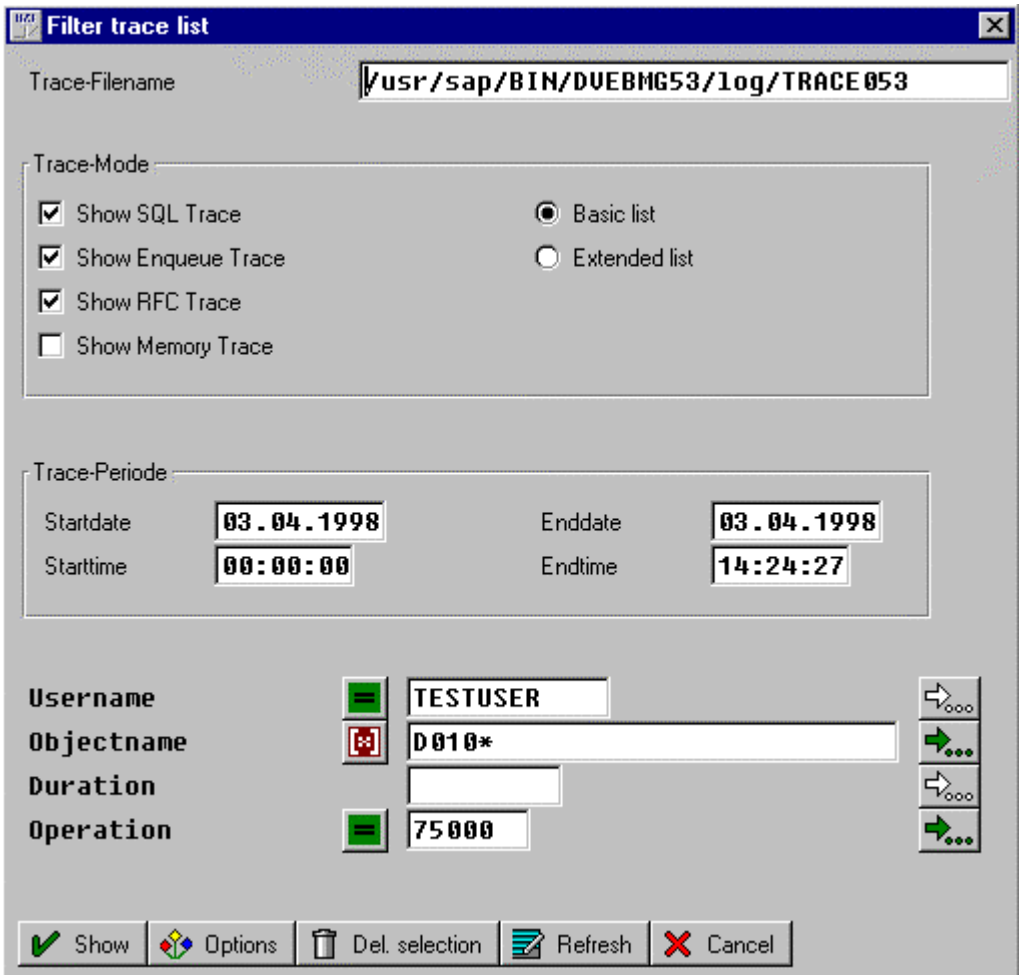
Display Filter

Prerequisites

You have switched off the performance trace and have opened the display filter by choosing *List trace* from the initial screen of the Performance Trace.

Use

You can use the display filter (“Filter trace list”) to restrict the number of logged trace records that are displayed on the basic list or extended list.



Features

If you do not enter any selections, all of the trace records are selected.

Enter the Name of the Trace File

The system initializes this parameter from the system environment. The current trace file is proposed by default. However, you can also specify a different filename. For further information, refer to Configuring the Trace File [Page 488].

Specifying the Trace Type to Display

This parameter is also initialized from the system environment. The default trace type is SQL Trace. If you start the display filter directly after the recording, the trace type appears as it was last configured.

Specifying the Trace Interval

The default trace interval is from 00:00:00 to the current system time on today's date. However, if you start the display filter directly after the recording, the trace interval is set from the starting time to the ending time of the recording.

Note that if you are working on a distributed system where the clocks on the database server and the application servers are not synchronized, any times determined automatically by the system may be inaccurate, which in turn may mean that not all trace records are displayed.

Enter Further Selections

For further information, refer to Other Filters [Page 460].

Options

Place the cursor on a parameter and choose *Options* to change the default option and choose a new operator.

Del. Selection

To delete the selection criteria for a parameter, choose Del. selection. See also Other Selection Options [Page 460]).

Refresh

To update the display filter screen, choose *Refresh* (or press **ENTER**).

Initialize the Filter Parameters

If you repeat the Performance Trace during an R/3 terminal session, the default filter parameters are set to your last selections. Use the *Initial filter trace list* function (right mouse button and choose from popup menu) to restore the original defaults.

Specify the List Type

Select either *Basic list* or *Extended list* and choose Show to display the corresponding list.

See also Displaying Lists of Trace Records [Page 462].

Other Filters

There is a range of further display options that you can set:

- User name
- Object name
  - SQL Trace: Name of the table in the SQL statement
  - Enqueue Trace: Name of the lock object
  - RFC Trace: Instance on which the function is called
- Duration
- Operation
  - SQL Trace: Database operation
  - Enqueue Trace: Lock object operation
  - RFC Trace: Execution type (client/server)

The following defaults are already set:

- For *Username:* The current user
- For *Objectname:* System tables (D010*, D020*, DDLOG) are not displayed.

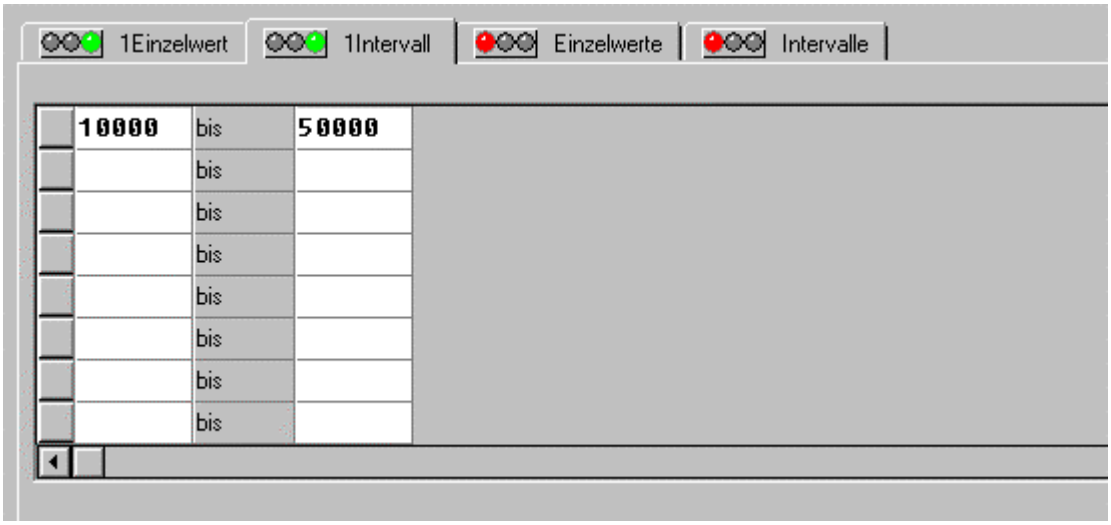
The *Duration* and *Operation* parameters are not initialized.

You can specify a numeric value for the *Duration*. This is measured in microseconds. For all other parameters, you can enter a pattern or name. You can use the wildcard characters ‘*’ and '+’ in patterns.

You can also change the operator in a specification. To do this, position the cursor on the corresponding parameter and choose *Options*. A dialog box then appears in which you can specify whether records satisfying the condition should be included (green stoplight) or excluded (red stoplight) from the selection.

If you want to enter more than one value, name or pattern for a parameter, use the *Multiple selection* option (“=>” icon on the right of the screen).

Example: You might want to restrict the duration to a particular interval:


1Einzelwert
1Intervall
Einzelwerte
Intervalle
10000
bis
50000
bis
bis
bis
bis
bis
bis
bis
bis

Displaying Lists of Trace Records

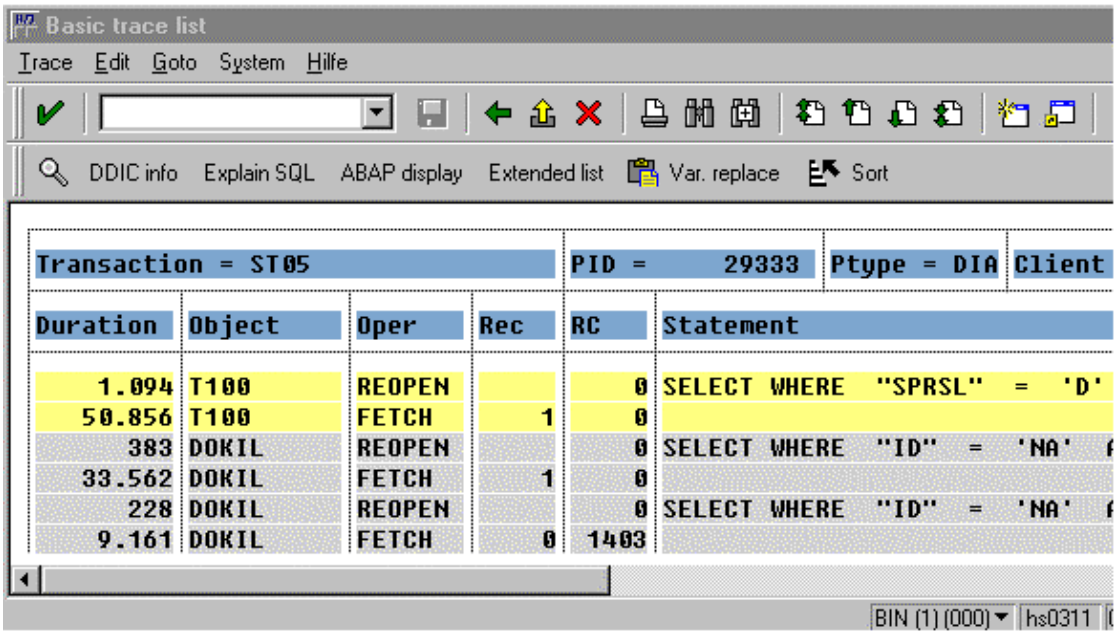
Prerequisites

You must have switched off the Performance Trace, set the display parameters in the display filter [Page 458] and displayed the basic or extended list.

Lists

You can display trace records in either a *basic list* or an *extended list*. The extended list contains everything from the basic list, along with three extra display columns. There is a range of analysis functions that you can use both on the basic list and on the extended list. The functions are the same in both lists. You can switch between the two list displays using a pushbutton in the application toolbar.

Basic List



The first line of the list contains a header containing the following information:

- Name of the transaction, process identification number, process type, client, and user name.

The next line contains the following headers:

- **Duration** Runtime for the statement in the form milliseconds.microseconds.
- **Object**
  - SQL trace record: Name of the database table
  - Enqueue trace record: Name of the lock object
  - RFC trace record: Shortened name of the instance on which the function module was executed
- **Oper**
  - SQL trace record: Name of the operation performed in the database. See also Measured Database Operations [Page 471]
  - Enqueue trace record: Name of the lock operation
  - RFC trace record: Client | Server (client means that a remote function was called, server means that the function was made available and executed)
- **Rec** Number of records
  - SQL Trace: Number of records retrieved or processed and passed between the R/3 System and the database.
  - Enqueue trace: Number of granules
  - RFC Trace: Not used
- **RC** Return code of the statement

- **Statement** Short form of the logged statement
  - Depends on the trace form.

    The runtime (**duration)** is highlighted in the list if it exceeds a given threshold value (100000 microseconds). This is declared in the type group “SQLT” as the constant “SQLT_DURATION_NEG”. You can change this value if you want to use a different threshold.

    Note that the **duration** can only be as precise as clock of your hardware platform. If the execution time of the statement is less than the smallest unit of time supported by the hardware clock, the duration will be zero.

Extended List

The extended list (*Extended list* pushbutton) contains three extra display columns:

- **hh:mm:ss.ms** The time at which the record was executed (in the form hours: minutes: seconds: milliseconds).
- **Program** Name of the program that executed the logged statement.
- **Curs**
  - SQL trace record: Number of the cursor (link to cursor cache) used to find the database entries.
  - Enqueue and RFC trace records: Not used.

Other Functions

Analyzing the Trace Records

- Sort list
- Display formatted logged statements
- Definition of the corresponding ABAP Dictionary object for SQL and Enqueue trace
- Display the logged statement “in situ” in the program code
- Display the access plan for a logged SQL statement
- List identical select statements in the trace list
- Summarize the trace list
- Switch between the two lists

For further information, refer to Analyzing Trace Records [Page 466]

Different Trace Types

The system displays different trace types in different colors.

Standard Functions

A range of standard R/3 list functions is also available to help you navigate and search in the list or save the list to an operating system file.

See also Saving Lists Locally [Page 490].

Analyzing Trace Records

Prerequisites

You have displayed the trace records that you want to analyze in a basic or extended list.

Functions

Sorting the Lists

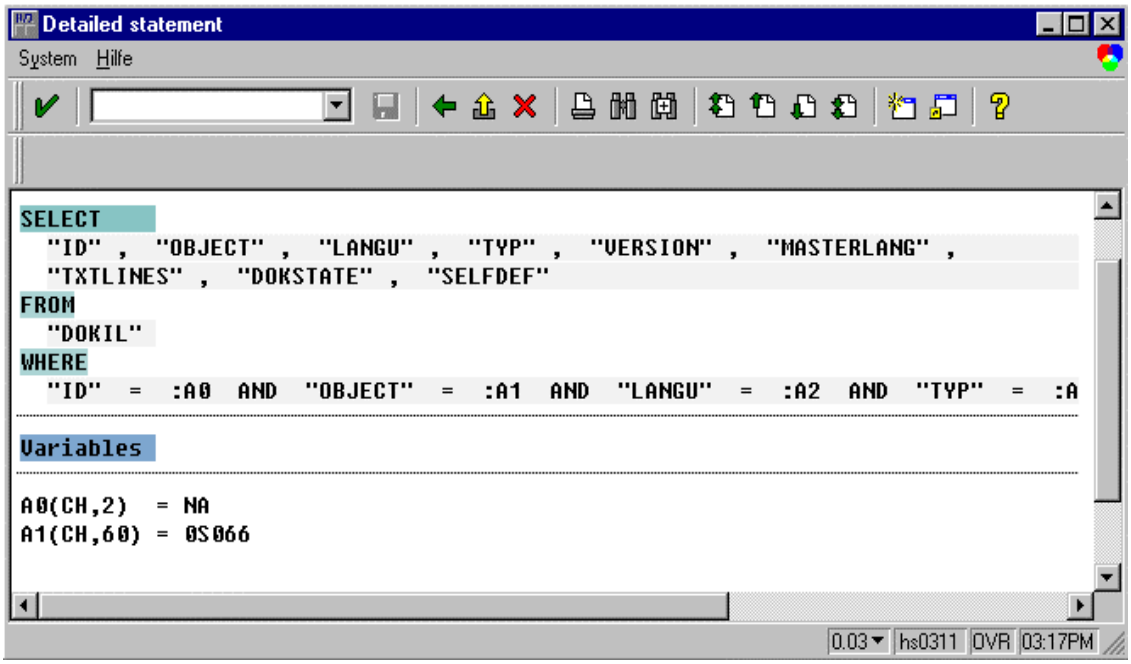
You can sort the list by any of the parameters in the list heading, that is, transaction name, process identification number, process type, client, and user name. To sort the list, position the cursor on the relevant column and choose *Sort*.

Switching Between Lists

To switch from the basic list to the extended list, choose *Extended list*. To return from the extended list to the basic list, choose *Back*.

Detailed Display and Replacing Placeholders

When the logged statement is formatted, you can specify whether to replace the placeholders in the statement by the current variables or leave them in the statement and list the variables separately. If there are no variables, the two display forms are identical. To display the statement, double-click its short form, or position the cursor on it and click the magnifying glass icon.

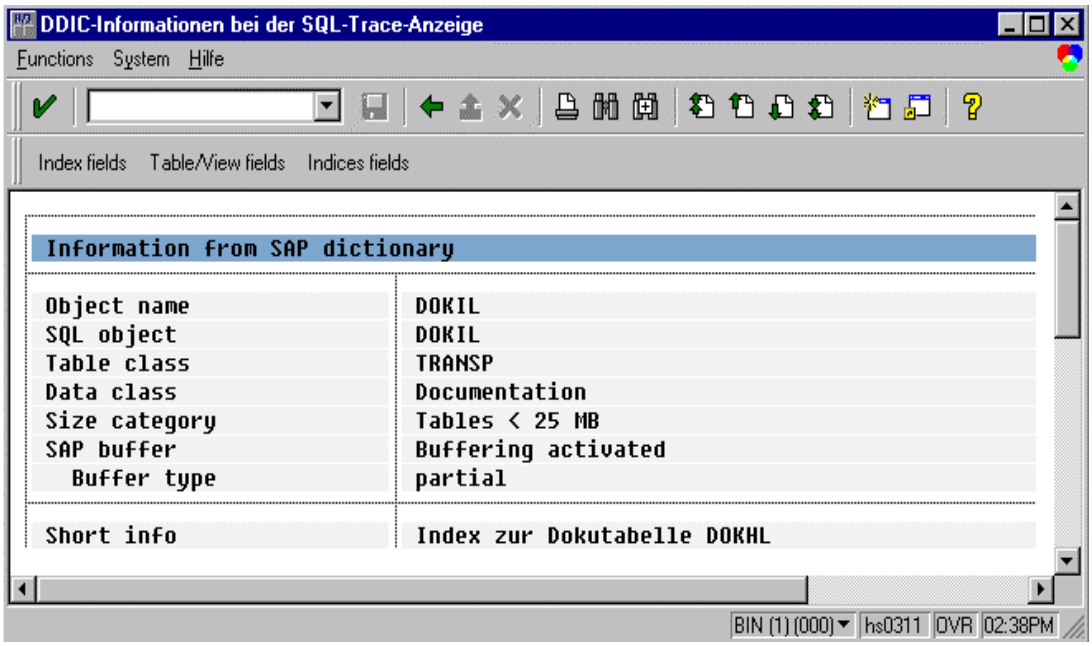


To replace the placeholders with the current variables, choose *Var. replace*.

Displaying Information about ABAP Dictionary Objects

To display ABAP Dictionary information for an object (table or lock object), position the cursor on the object and choose *DDIC info*. If the current statement contains more than one ABAP

Dictionary object (for example, a join), the **Object** column contains the first object to appear in the statement.

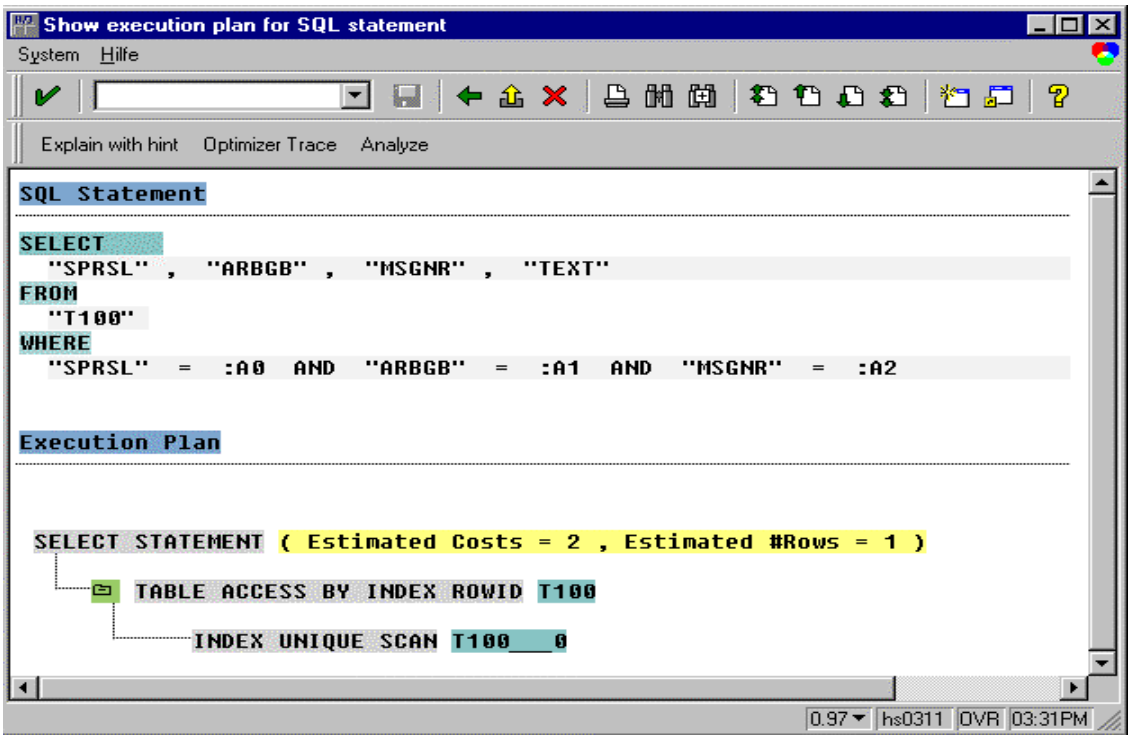


If the log entry is an RFC entry, the column contains a shortened version of the name of the instance on which the function module was executed. In this case, you cannot display a ABAP Dictionary definition.

You can display further ABAP Dictionary information by opening the actual definition of the object within the ABAP Dictionary. For further information, see Finding Dictionary Information [Page 493].

Execution Plan for SQL Statements

To display the execution plan of a selected SQL statement, place the cursor on the statement and choose *Explain SQL*. The SQL statements for which an execution plan can be displayed depends on the database system that you are using. The execution plan for a SELECT statement under Oracle looks like this:



Displaying the Source Code

To switch to the source code containing the current statement in the log, position the cursor on the short form display of the statement and choose *ABAP Display*.

Note that the source code cannot always be displayed. For example, if the call comes from the R/3 kernel, you cannot branch to the program code.

Identical Selects

When you are analyzing a trace log, it can be particularly useful to find out if there are any identical select statements. You can do this by choosing *Identical selects*. The system compiles a list of any SQL statements that are executed more than once. You can then eliminate any SQL statements that are not required.

Summarizing the Trace List

You can summarize the select statements by choosing *Summary*. This leaves you with an overview of the total runtime and the total number of records retrieved.

SQL Trace Analysis

The SQL Trace part of the Performance Trace tool allows you to see how the OPEN SQL statements that you use in ABAP programs are converted to standard SQL statements (see Embedded SQL [Page 470]) and the parameters with which the embedded SQL statements are passed to the database system.

Overview

While the trace is switched on, the SQL Trace function records all database activity by a particular user or group of users. The R/3 System takes OPEN SQL statements and converts them in to embedded SQL statements that it passes to the database. It is the embedded SQL statements, their parameters, return codes, and the number of entries retrieved, inserted, or deleted that are recorded in the SQL Trace file. The log file also contains the runtime of the statement and the place in the application program from which it was called.

The SQL trace tells you:

- The SQL statements executed by your program.
- The values that the system uses for particular database access and changes.
- How the system converts ABAP Open SQL statements (such as SELECT) into Standard SQL statements.
- Where your application executes COMMITs.
- Where your application repeats the same database access.
- The database accesses and changes that occur in the update part of your application.

Embedded SQL

One of the difficulties of connecting a programming language with an SQL interface is the transfer of retrieved data records. When the system processes an SQL statement, it does not know how big the result will be until it has made the selection. The result consists of table entries, which all have the same structure. The system has to transfer these records to the calling program in the form of a data structure, for example an array, that is known to the calling program.

The disadvantage of an array is its static definition. You have to specify the size of an array before runtime. However, because you cannot know the size of the dataset the system will return, you must define a very large array to avoid an overflow.

To circumvent this problem, the R/3 Basis System translates ABAP Open SQL statements into Embedded SQL. In Embedded SQL, the system defines a **cursor** that is used to regulate the data transfer between ABAP programs and a database. See also Database Operations [Page 471].

During every FETCH operation, the database passes one or more data records to the R/3 database interface.

Measured Database Operations

Each SQL statement is broken down into database operations by the R/3 System. The SQL Trace allows you to measure the runtime of each of these operations:

DECLARE	Defines a new cursor within an SAP work process and assigns the SQL statement to it. The short form of the statement is displayed in the list of trace records. The cursor has a unique *cursor ID*, which is used in communication between the R/3 System and the database system.
PREPARE	Converts the SQL statement and determines the execution plan.
OPEN	Opens a cursor for a prepared (converted) SELECT statement. OPEN passes the parameters for the database access. It is only used with SELECT statements.
FETCH	Passes one or more records selected in the SELECT statement to the database interface of the R/3 System. The data records are identified by the cursor ID.
REOPEN	Reopens a cursor prepared by the system for a previous SELECT statement and passes a new set of parameters to the database.
EXEC	Passes the parameters for the database statement, and executes the statements that change data in the database (such as UPDATE, DELETE, or INSERT).
REEXEC	Reopens a cursor prepared by the system for a previous EXEC statement.

For information about the sequence in which these operations occur, refer to Logical Sequence of Database Operations [Page 472].

Logical Sequence of Database Operations

The database operations are all related, and always occur in the same logical order:

The **DECLARE** function defines a cursor and assigns a number to it. DECLARE is followed by PREPARE.

PREPARE takes an SQL statement, for example:

```
select * from sflight where carrid eq 'LH'.
```

It determines the access method, and prepares the statement to be passed to the database. At this stage, the system is only concerned with the structure of the SQL statement, and not the values that it contains.

The **OPEN** function takes the prepared SQL statement and adds the relevant values to it. In the above example, OPEN would give the value LH for the field carrid.

FETCH passes the entries from the database to the database interface of the R/3 System. All of the database operations required to execute an SQL statement are linked by the same cursor ID.

If the SQL statement makes changes in the database (INSERT, UPDATE, DELETE), **PREPARE** is followed by **EXEC**, which executes the statement.

If the system can refer back to an SQL statement that has already been prepared, there is no PREPARE operation, and the statement is executed using **REOPEN** or **REEXEC** as appropriate.

Buffering

The system ensures that data transfer between the R/3 System and the database system is as efficient as possible. To do this, it uses the following techniques:

- Table buffering: The program accesses data from the buffer of the application server.
- Database request buffering: Individual database entries are not read or passed to the database until required by an OPEN SQL statement.

When you analyze trace records, you should also examine the system's buffering mechanisms.

Table Buffering

Tables can be either partially or fully buffered (refer to Buffering Database Tables [Ext.]). This means that an OPEN SQL statement only accesses the database if the results of the statement are not already in the buffer. Consequently, the SQL Trace does not contain a command or command sequence for every OPEN SQL statement. On the other hand, every SQL statement in the trace file has been sent to the database and executed.

Database Request Buffering

To minimize the number of time-consuming PREPARE operations, each work process on the application server has a buffer of SQL statements that it has already prepared. The default buffer size is 250 statements.

Whenever an OPEN SQL statement appears in a program, the work process checks whether it already exists in the "**statement cache**". If it does, the statement is executed immediately; that is, there is no further PREPARE operation, and the statement is executed using a REOPEN (for SELECT) or a REEXEC (for INSERT, UPDATE, or DELETE).

If the statement does not exist in the buffer, it must be prepared for the subsequent OPEN or EXEC operation. The buffer administration uses a LRU (least recently used) algorithm to delete those statements, whenever necessary, that are only seldom used. Frequently-used statements normally only need to be prepared once.

Application servers buffer DECLARE, PREPARE, OPEN, and EXEC statements in the cursor cache of their work processes. Once the system has opened a cursor for a DECLARE statement, it can carry on reusing it in the same work process.

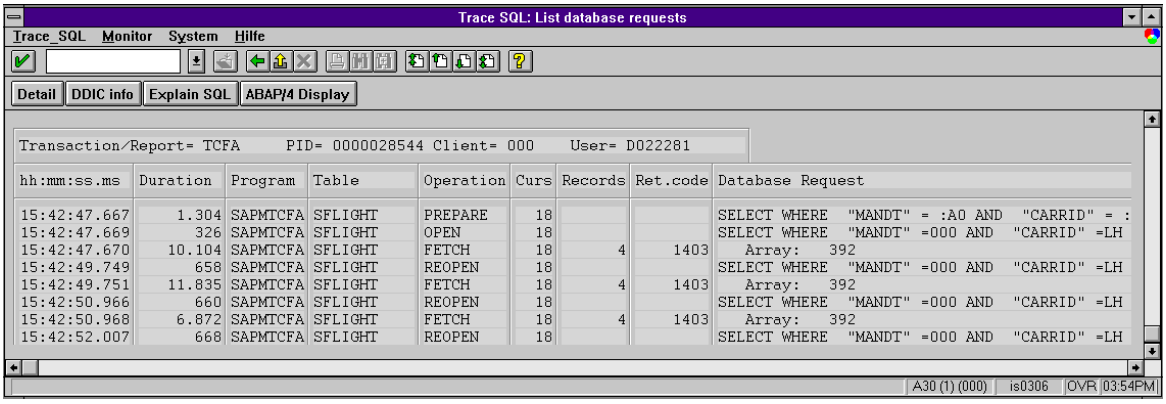
Analyzing a Sample SQL Data File

When you create an SQL trace file for an application, you can see exactly how the system handles database requests. In a sample application, a report reads and later changes records on the ABAP Dictionary table SFLIGHT using ABAP Open SQL statements. Since the table SBOOK is not buffered, the system first needs to access the database to retrieve the records. In the sections below, the data file from the sample application is examined.

Read Access

The first screen of the SQL Trace data file displays each measured database request the application made. The trace file records when the request occurred and its duration. The ABAP Dictionary table involved in the request is also listed.

A trace file for a read access of the table SFLIGHT might look like this:



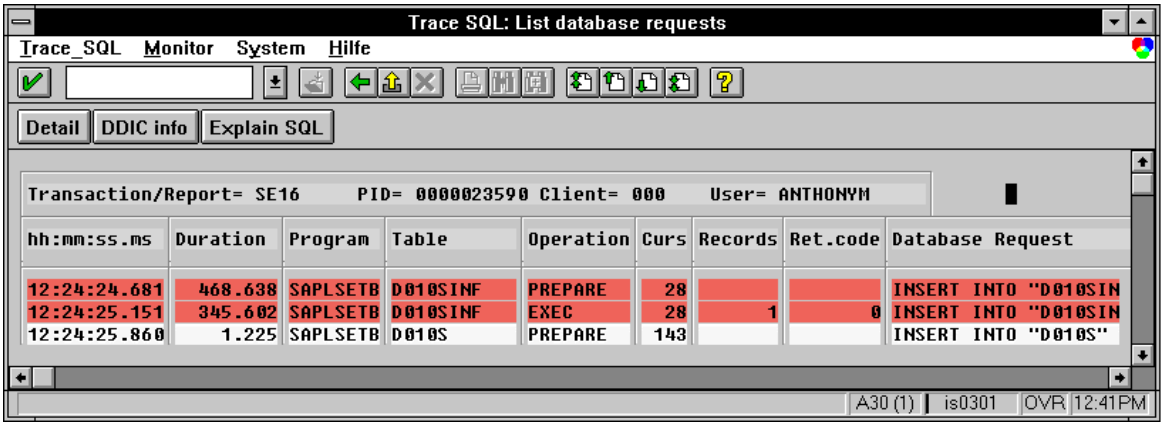
The system measured several database operations involved in retrieving records from SFLIGHT:

Operation	Function

PREPARE	Prepares the OPEN statement for use and determines the data access method. Since an active cursor with the number 18 is available in the work process's cursor cache, the system does not perform a DECLARE operation. However, the system must prepare the SELECT statement that is used to read the table SFLIGHT. The system does not issue the field 'MANDT' and 'CARRID' in the SELECT statement a value at this point but instead gives it a database-specific marker.
OPEN	Opens the cursor and specifies the selection result by filling the selection fields with concrete values. In the case of this example, the field 'MANDT' receives the value '000' and the field 'CARRID' receives the value 'LH'. The OPEN operation then creates a set of retrieved records in the database.
FETCH	Moves the cursor through the dataset created by the OPEN operation. The array size displayed beside the fetch data means that the system can transfer a maximum package size of 392 records at one time into the buffered area. The system allocates this space on the application server for the SFLIGHT table. In the above example, the first FETCH retrieves the maximum number of records from the dataset. Then, the FETCH transfers these records to the program interface.

Write Access

An example SQL data file analyzing a request that changes data in the Table D010SINF might look like this:



The example shows the system inserting new records into the table (INSERT). As in the first example, where the system carried out a read-only access, the system needs to prepare the database operations (PREPARE) that change the database records. The PREPARE precedes the other operations.

Example Explanation of an Oracle Statement

You can use the SQL Trace facility to view explanations of specific Oracle statements. From within a trace file display, you use the *Explain SQL* function to display more information about a specific database request. The *Explain* function is available only for PREPARE and REOPEN operations. To explain a request:

1. Place the cursor on a line containing the database request you want explained.
2. Choose *Explain.*.

   The *Explain* screen shows you the database's strategy for carrying out the selected operation.

For example, if you are working with an ORACLE database, you can show the explanation for the following statement:

```
select * from fllog where flcode = '00000123'.
```

The system provides the following explanation:

Operation	Options:	Object Name	ID	PAID	POS
SELECT STATEMENT			0		
TABLE ACCESS BY ROWID		FLLOG	1	0	1
INDEX UNIQUE SCAN		FLLOG___0	2	1	1

QUERY PLAN

SELECT STATEMENT		
	TABLE ACCESS BY ROWID	FLLOG
	INDEX UNIQUE SCAN	FLLOG___0(UNIQUE)

The fields in the explanation have the following meanings:

OPERATION	Identifies the operation name.
OPTIONS	Operation attributes.
OBJECT NAME	Identifies the object involved in the operation.
ID	Specifies the operation's ID number.
PAID	Specifies the ID number that the current operation transfers its results to. This is important if nested accesses on various hierarchy levels are involved.
Position	Identifies the next number for operations working on the same hierarchy level.

In the example above, the key is fully qualified. The database can use the primary key index FLLOG__0 to access the table records. Every transparent table in the ABAP Dictionary has a primary key. The system automatically creates an index for this key. The primary key index is

also unique, meaning that there is only one index entry for every line in the table. As a result, the system uses the UNIQUE SCAN operation.

The UNIQUE SCAN has the ID 2 and parent ID 1. This means that the operation passes its results to the operation with ID 1. ID 1 belongs to the TABLE ACCESS operation. TABLE ACCESS can directly access one record because of the uniqueness of the BY ROWID index. Once the system chooses an access strategy, it sends the SELECT statement with ID 0 to the database.

If the SELECT statement does not specify a fully qualified key, the database could be forced to read the records using a FULL TABLE SCAN. In this case, no index is available and the database reads the entire table in packages.

If the index is ambiguous, the database uses a RANGE SCAN. The RANGE SCAN scans over an index area that might contain several sets of retrieved data.

The NESTED LOOP operation exists for nested reads where several indexes are joined together within one database access.

Example Explanation of an Informix Statement

You can use the SQL Trace facility to view explanations of specific Informix statements. From within a trace file display, you use the *Explain SQL* function to display more information about a specific database request. The *Explain* function is available only for PREPARE and REOPEN operations. To analyze a statement:

1. Place the cursor on a line containing the database request you want explained.
2. Choose *Explain*..

   The *Explain* screen shows you the database's strategy for carrying out the selected operation.

If you are working with an Informix database and you display the explanation for the following statement:

```
select owner from systables where tabname = 'atab'
```

The system provides the following explanation:

```
Execution plan of a select statement (Online Optimizer)

QUERY:

SELECT OWNER

FROM SYSTABLES

WHERE TABNAME = '  ATAB'

Estimated Cost: 1

Estimated # of Rows Returned: 1

1) informix.systables: INDEX PATH

      (1)  Index Keys: tabname owner  (Key-Only)

            Lower Index Filter: informix.systables.tabname = '  ATAB'
```

The fields in the explanation have the following meanings:

QUERY	Identifies the SQL statement that was traced.
Estimated Cost	Estimates the database expenditure required to execute the statement. The cost-based optimizer estimates this value in terms of the I/O and CPU required by the statement. The larger the **Estimated Cost** the greater the expenditure.
Estimated # of Rows Returned:	Estimates the number of table rows that the SQL statement will return.

Immediately below the number of rows returned is the selected execution plan. In the above example, the execution plan is as follows:

```
1) informix.systables: INDEX PATH
```

The **1)** indicates that the system processes the **systables** table as the first step of the execution plan. For queries that span several tables (views and joins), the numbering sequence indicates the order the system processes the tables. In this example, only a single step was needed.

Example Explanation of an Informix Statement

The execution plan specifies the type of table access. In the above example, the access was the **`INDEX PATH`**. Access to the required data row is made using the index of the **`systables`** table. Normally, the execution plan uses the primary key as an index. Every transparent table in the ABAP Dictionary has a primary key and the system automatically creates an index for this key.

When the system must read a large proportion of a table, the system does not use the primary key as an index.

For this example, the system did not need to read the row that corresponds to the index key. The information that was required was present in the key itself. The explanation indicates this using the phrase **`Key-Only`** as follows:

```
(1) Index Keys: tabname owner (Key-Only)
```

If a SELECT statement is specified without a fully-qualified key, the database may need to read the relevant rows with a FULL TABLE SCAN. In this case, you will not see an index in the SQL-Explain output but instead you will see something like the following:

```
1) informix.systables: SEQUENTIAL  SCAN
```

This indicates that a read of the entire table is necessary (FULL TABLE SCAN).

With more complex operations, where the combination of results from several SELECTS on different tables is required, you will see further strategies mentioned (such as MERGE JOIN, DYNAMIC HASH JOIN). These refer to the join strategy chosen by the optimizer.

Ensuring Up-to-Date Information

The optimizer can compute an accurate value for each explanation field only if the statistical information for each table is up to date. To enable the optimizer to compute accurate values for the above fields, you must ensure that up-to-date statistical information about the contents of relevant tables is available.

To update your information, use the *Update Statistics* function. Since the execution plan selected by the optimizer (for example, the use of a table scan versus an index) depends crucially on this information, you should always ensure that it is kept as up to date as possible by regularly running *Update Statistics*.

Enqueue Trace Analysis

Use

The Enqueue Trace allows you to track the locking and unlocking statements that your application or the R/3 System uses, and the locking objects and parameters to which they apply. You can display the records logged in the trace file for further analysis.

Features

While the Enqueue Trace is switched on, the system records all of the locking and unlocking statements that occur for a user or group of users.

The trace recording contains the following information:

- The locking statements executed
- The table names in the lock object
- The name of the program that set the lock
- The lock type
- The lock owner
- The time required to set the lock
- The time required by the enqueue server to release the lock

See also:

Enqueue Trace Records [Page 482]

Detailed Display of Enqueue Trace Records [Page 483]

Enqueue Trace Records

The following list columns are particularly relevant to enqueue trace records:

In the Basic List

- **Duration** Runtime for the lock operation in the form milliseconds.microseconds.
- **Object** The name of the lock object.
- **Oper** The lock operation. For further information, refer to Lock Objects [Ext.].
- **RC** Return code.
  If the value in this column is zero, the enqueue operation was successful. If it is "1", the operation was unsuccessful because the lock object or parts of it were already locked.
- **Rec** Number of granules in the lock object.
- **Statement** This column lists the granules for the lock request. If there is more than one, they are separated by the "|" character. The lock mode, lock table, and lock argument of each granule are listed.
  See also: Lock objects [Ext.].

In the Extended List

- **hh:mm:ss.ms** The time at which the lock operation was performed, in the form hours : minutes : seconds. milliseconds.
- **Program** Name of the ABAP program that requested the lock operation.
- **Curs** not used.

For a more detailed analysis, use the Detailed Display of Enqueue Trace Records [Page 483].

Detailed Display of Enqueue Trace Records

To display an enqueue trace record in more detail, choose the magnifying glass icon on the basic list or extended list.

The detailed display contains the following information:

- **First line:**
  - Operation of the enqueue statement
  - Name of the lock object
- **Owner** Lock owner.
- **Owner_UPT** Owner of the lock in the logical processing unit.
- **Scope-Parameter** Indicates who owns the lock.
- **Collision Owner** If this field is filled, the user whose name appears is the owner of the lock that was requested.
- **Collision Object** The lock object that already possesses the lock requested (or parts of it).
- **Collision Username** The user name of the lock owner.
- **Remote Time** Total time in milliseconds required for the lock (in client). This is made up of the request time and the time required for the communication. If the request is local, the local execution time is entered.
- **Request Time** Time required by the enqueue server to execute the lock (server time, in milliseconds). If the). In the case of an asynchronous call, or no server being requested, the time is zero.
- **ABAP-Programname** Name of the ABAP program that requested the lock.
- **Number** Number of granules.
- **Lock-Mode** Type of lock.
- **Tablename** Table name of the lock object.
- **Granularity Argument** Lock argument.

RFC Trace Analysis

Use

You can use RFC Trace to monitor the remote calls that your application or the R/3 System makes, and the instances on which they are executed. The RFC Trace generates a trace file of logged trace records that you can display and analyze further.

Features

While the RFC Trace is switched on, the system records all Remote Function Calls made by a particular user or group of users.

The trace recording tells you:

- Which function modules have been called "remotely" from the program that you analyzed.
- Whether the RFC was successfully executed.
- The total runtime required for the remote call.
- Whether the RFC communication was as a RFC client or an RFC server.
- The instance on which the remote call was executed.
- The technical parameters of the remote instance.
- The number of bytes sent and received in the Remote Function Call

See also:

RFC Trace Records [Page 485]

Detailed Display of RFC Trace Records [Page 486]

RFC Trace Records

The following columns of an RFC trace record are particularly significant:

In the Basic List

- **Duration** Execution time for the Remote Function Call in the form milliseconds.microseconds.
- **Object** Shortened name of the instance (system) in which the remotely-called function module was executed.
- **Oper** ID of the cross-instance RFC communication. The RFC Client is the instance that calls the function module remotely. The server is the instance that makes the function available and executes it.
- **Rec** Not used.
- **RC** Return code of the logged remote call. If the return code is zero, the remote call was successful. If the value is not equal to zero, an error occurred.
- **Statement** This column contains the name of the local instance, the name of the remote instance, the name of the function module called, and the number of bytes sent and received.

In the Extended List

- **hh:mm:ss.ms** The time at which the remote call was executed, in the form hours: minutes: seconds: milliseconds.
- **Program** Name of the ABAP program that triggered the Remote Function Call.
- **Curs** Not used.

For further details, refer to Detailed Display of RFC Trace Records [Page 486].

Detailed Display for RFC Trace Records

To display detailed information about an RFC trace record, choose the magnifying glass icon on the basic or extended list.

The detailed display contains the following information:

- **Name** of the function module that was called.
- **Local IP-Address** IP address of the local host on which the RFC trace was generated.
- **Local computername** Name of the local host.
- **Remote IP-Address** IP address of the remote host on which the function module was executed.
- **Remote computername** Name of the remote host.
- **Client/Server** Classification of the RFC communication as RFC client or RFC server.
- **ABAP-Program** Name of the ABAP program that triggered the remote call. If there is no name explicitly specified ("space"), the call came from the kernel of the R/3 System.
- **RFC-Time** Total runtime of the Remote Function Call.
- **RFC-Time** Total time required to execute the remote call.
- **Bytes send** Number of bytes sent to the remote system.
- **Bytes receive** Number of bytes received from the remote system.

Other Funtions

Configuring the Trace File

Profile Parameters

The system requires the two following profile parameters to be set in order for the performance trace to work:

- **rstr/file**: The pattern for the path name of the trace file.
- **rstr/max_diskspace:** The maximum size (in bytes) that can be assigned to a trace file.

  The recommended default size is 16 384 000 (16 000 KB). However, you can change the name of the trace file at any time using Transaction **ST01**. This allows you to work with several trace files at once and prevent performance data from being overwritten.
- **rstr/fileset/allowed:** This parameter determines whether users should be allowed to change the name of the trace file. The default value is "1", that is, it is possible to change the name of the file. If you change the value to "0", users cannot change the name of the file.

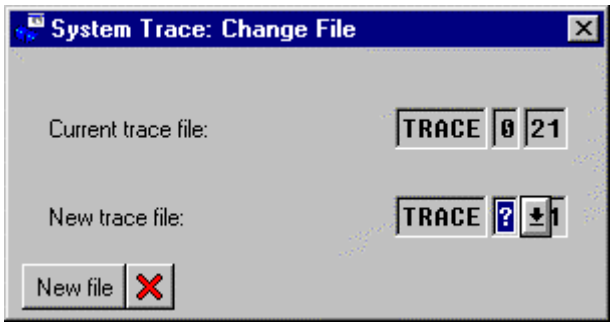
These parameters are set when the system is installed. You can also change them using Transaction **RZ11**.

Changing the Name of the Trace File

The name of the trace file into which the performance trace writes entries is predetermined by the system administrator. If the profile parameter **rstr/fileset/allowed** is set to its default value, you can change the name of the trace file in the profile parameter **rstr/file** using Transaction **ST01**.

Procedure

1. Start Transaction **ST01**, either using its transaction code, or by choosing *Administration* → *System administration* → *Monitor* → *Trace* → *SAP System trace*.
2. Choose *Edit* → *New file.*
   A dialog box appears in which you can choose any character for the third-last character (as long as the profile parameter **rstr/fileset/allowed** is set to "1").



3. Enter a character.
   You can enter any character from **0** through **9** and **A** through **Z**.

Saving Lists Locally

The results of the trace recording are written in a trace file, which you can display in the form of a basic or an extended list. You can save these lists in local files on your frontend. This allows you to keep performance data that would otherwise eventually be overwritten.

Procedure

1. Start the initial screen [Page 453] of the performance trace.
2. Choose *List Trace.*
   The display filter [Page 458] appears.
3. In the display filter, set the range of trace records that you want to display.
4. Choose *Show.*
   The basic list or extended list appears, according to your choice.
5. Choose *Trace* → *Save to PC file…*
   The *Save List in File* dialog box appears.
6. Select the format in which you want to save the data and choose *Continue*.
7. In the following dialog box, enter the name of the local file.
8. Choose *Transfer*.

Result

The trace records from the list have now been saved in a file on your frontend that you can display in the format you specified during the above procedure.

If you choose *Unconverted* in step 6, you can also download the list to the clipboard.

The *Explain SQL* Function

The *Explain SQL* function allows you to analyze the database strategy used to access any given table or view defined in the ABAP Dictionary or in the database itself. With it, you can see the indexes that the system uses.

There are two ways of analyzing an SQL statement:

- From the initial screen of the Performance Trace, you can check a statement without generating a trace file.
- From the list display of a trace file, you can select an ABAP Dictionary table and display details of the access methods that it uses.
  See also Analyzing Trace Records [Page 466].

Note that the result of the *Explain SQL* function (both the SQL statement and the operation) is **database-specific**. There are certain database systems under which you can only use the *Explain SQL* function for the SELECT statement.

To be able to interpret the results of the *Explain SQL* function, you will need a sound knowledge of the relevant database system. For further information, refer to the documentation of the database manufacturer.

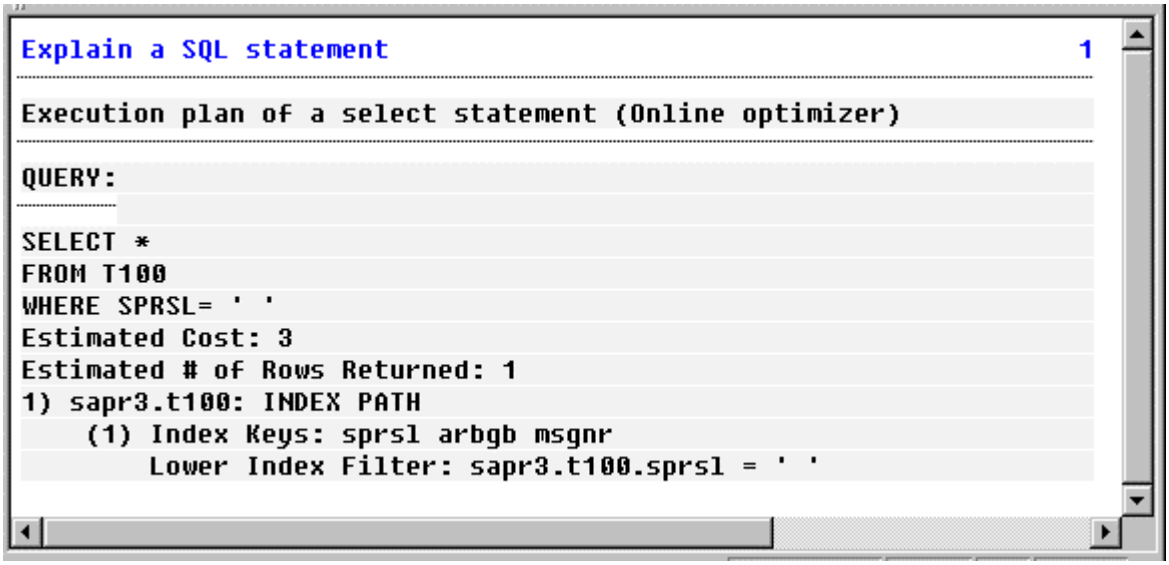
Analyzing an SQL Statement Without a Trace File

1. On the initial screen of the Performance Trace, choose *Explain one SQL statement* (for Oracle), or *Explain one SQL request* (for Informix).
   The system displays a screen containing an input field. Enter the name of the table (or view) that you want to use.
2. Enter an SQL statement.
   Make sure that you enter the name in uppercase.

```
000010 SELECT * FROM T100 WHERE SPRSL=:*T100-SPRSL
```

3. Choose *Save*.

   The system analyzes the statement and displays information about the access strategy of the database.


Explain a SQL statement 1
Execution plan of a select statement (Online optimizer)
QUERY:
SELECT *
FROM T100
WHERE SPRSL= ' '
Estimated Cost: 3
Estimated # of Rows Returned: 1
1) sapr3.t100: INDEX PATH
(1) Index Keys: sprsl arbgb msgnr
Lower Index Filter: sapr3.t100.sprsl = ' '

Finding Dictionary Information

From the list of trace records, you can display further information about ABAP Dictionary objects, and also branch to the definition of the object in the ABAP Dictionary.

Prerequisites

The trace records of the trace file must not have been overwritten.

Procedure

1. Start the initial screen [Page 453] of the Performance Trace.
2. Choose *List Trace.*
   The display filter [Page 458] appears.
3. Define the range of trace records that you want to display.
4. Choose *Show.*
   The trace file appears in a basic list or an enhanced list, according to your selection.
5. Position the cursor on the line containing the ABAP Dictionary table that you want to analyze.
6. Choose *DDIC-Info.*
   The first part of the ABAP Dictionary information screen displays the administrative data for the chosen tables, such as its class, amount of memory required, and so on.
   The second part of the screen contains information about the indexes that exist for the table. ABAP Dictionary tables always have at least one index, which is drawn from the primary key of the table. You can also create further indexes in the ABAP Dictionary, which are also listed here.
7. Choose *Table/View fields.*
   This function branches from the trace list to the table definition in the ABAP Dictionary.

Information About Development Objects

The ABAP Workbench contains a comprehensive information system. This documentation contains the following sections:

Navigation and Information System: Overview

The ABAP Workbench provides the following tools for navigating among development objects: The Repository Browser, the R/3 Repository Information System, and the application hierarchy. All three navigation tools use a "file manager" type interface for displaying development objects. The Workbench also contains the Data Browser for displaying the contents of database tables.

The Repository Browser is the central tool for organizing and managing your personal development objects. The Repository Browser is the most commonly-used tool in day-to-day development.

The R/3 Repository contains all of the development objects in the system. From the Repository Information System you can search for Dictionary objects, program objects, function groups, and so on. You can use the Repository to:

- generate lists of programs, tables, fields, data elements, and domains.
- find where tables and fields are used in screens and ABAP programs.
- display foreign key relationships and so on

The Application Hierarchy depicts the organization of all the applications in your R/3 system. The application hierarchy is an organizational tool. Each company defines its own hierarchy explicitly. You can use this tool to view the development objects used by each application in your company. You can also use the application hierarchy to plan an application before you develop it.

The Data Browser is a tool for retrieving information about a table without using an ABAP program. You can browse the contents of a table and branch from a specific entry to its related check table entries. If the table attributes allow it, you can also create or update table records with the Data Browser.

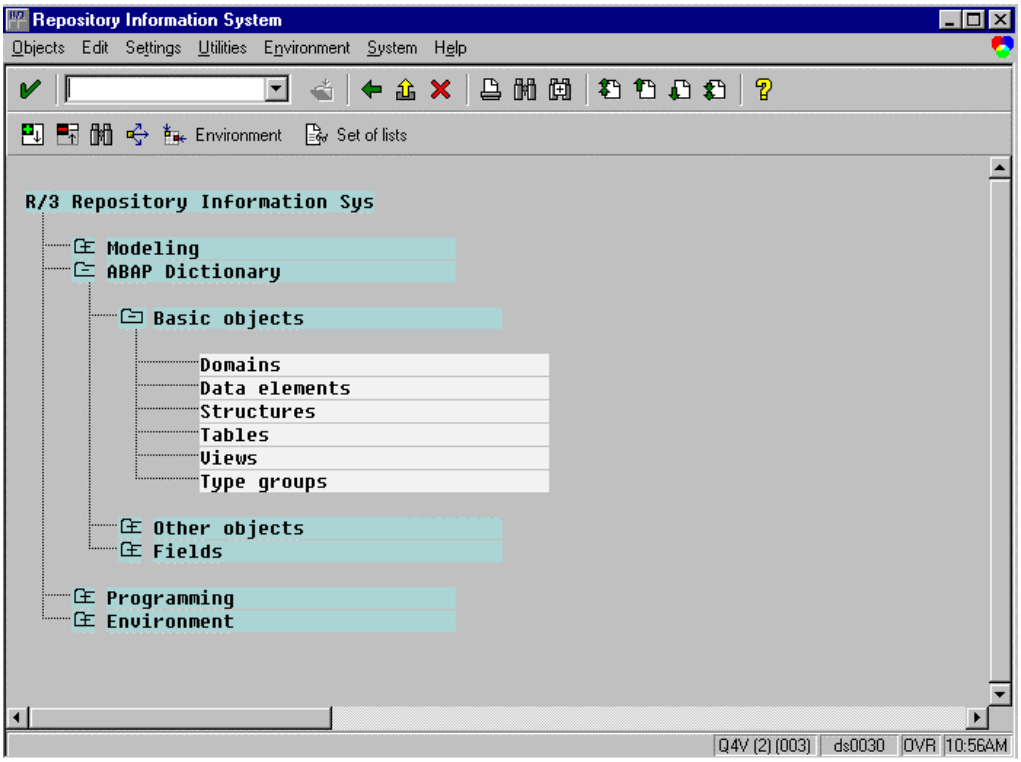
The Right Tool at the Right Time

From each of the Workbench information and navigation tools, you can reach not only any of the objects in your system but any of the Workbench tools. For example, if you navigate to a table object, the system starts the Dictionary tool and displays the table within it. If you branch to a program object, the system starts the ABAP Editor and displays the program within the Editor.

The Repository Information System

You use the Repository Information System to search for objects in the R/3 System. To access the R/3 Repository Information System from the ABAP Workbench, choose *Overview → Repository Infosys.*

The initial screen of the Repository Information System displays a hierarchical list of all the different types of objects in your R/3 System.



The object categories in the R/3 System are modeling objects, ABAP Dictionary objects, programming objects, and environment objects.

The Selecting Objects in Lists [Page 23] section describes in more detail how to work with the lists in the Repository Information System.

For example, a domain that matches specific requirements for an application you are writing. To search for an object:

1. Open the Repository Information System.
2. Select an object category.
3. Choose *Find*.

   The contents of the search screen depend on the object category. By default, the system displays the standard selection fields for the object.
4. Enter search values and choose *Execute*.

   The system displays the matches. By default, the system displays the basic list. However, you can display an overall view by choosing *Edit → Overall view*.

In the *Settings* group box in the bottom part of the search screen, you can enter the maximum number of hits in the *Max. Hits* field. By default the system searches both customer and SAP objects.

Search Results

When a search is finished, a result list appears. The utilities available on the results display depend on the object you are searching for.

Saving Search Criteria as a Variant

You can create variants for the objects you search for frequently. A variant is a set of search criteria. See the Getting Started [Ext.] documentation for information about creating variants. To list the variants available for an object, choose *Get variant*. To save your current search criteria as a variant, enter the criteria and choose *Goto* → *Variant* → *Save as variant*.

Customizing Selection Values

The *Maximum no. of hits* value specifies the number of matches to display. To do this, choose *Settings* → *User parameters* from the initial screen of the Repository Information System. The Maximum number of hits shows the number of found locations that the system will display. You can set the *Entry variant* as follows:

Standard selection criteria	Identifies the SAP_STANDARD variant. This is the default SAP variant.
All selection criteria	Specifies that all available selection criteria are available.
User-specific variant	Identifies a variant created by the user. You must enter the name of the variant in the field provided.

Where-used List

The *Where-used list* function is a search utility of the Repository Information System. It is available from most screens in the ABAP Workbench. To use this function from the initial screen of the Repository Information System, select an object type, such as *Data models*, and choose *Where-used list*. For details on using this function, see Where-used Lists [Page 500].

Environment Analysis

You can use the environment analysis function to determine the external references of an object (that is, the referenced objects that do not belong to the object itself). For details on using this function, see Environment Analysis [Page 498].

Environment Analysis

Use

You use this function to determine the external references for an object. External references are those objects to which your object refers but which are not defined within the object itself.

The external references for development class **XYZ** are all development objects from other development classes that are referenced by at least one object from development class **XYZ**.

Environment analysis is a useful function if you want to see how well an object is encapsulated. A fully encapsulated object has no external references.

It is particulary important to determine an object's environment before transporting objects into other systems. You must ensure that the required environment will be present in the target system after the transport (in other words, your object must not reference local objects).

Prerequisites

Before you can use this function, you need to specify the type of object for which you want to determine the environment. To do this, enter the Repository Information System, place the cursor on an object type and choose *Environment*.

Features

The system lists all external references to the specified object. The list is displayed sorted by object type.

Determining the Environment

1. Open the Repository Information System.
2. Position the cursor on the object type for which you want to determine the environment.
3. Choose *Environment*.
4. In the *Environment Analysis* dialog box, enter an object name.
5. If there is a list of object types for the environment analysis, you can restrict the analysis by selecting fields.
6. If you only want to search in certain development classes, you can enter these using the *Search area* function.
   When you have finished making your entries, choose *Copy* to continue.
7. Start the environment analysis in one of the following ways:
   - Start immediately: The system runs the environment analysis immediately, and outputs a list of results.
   - Start in the background: Starts the environment analysis as a background job. Once the analysis is finished, the system sends a dialog message to tell you that the list has been created. You can access the list by choosing *Set of lists* on the Repository Information System initial screen.

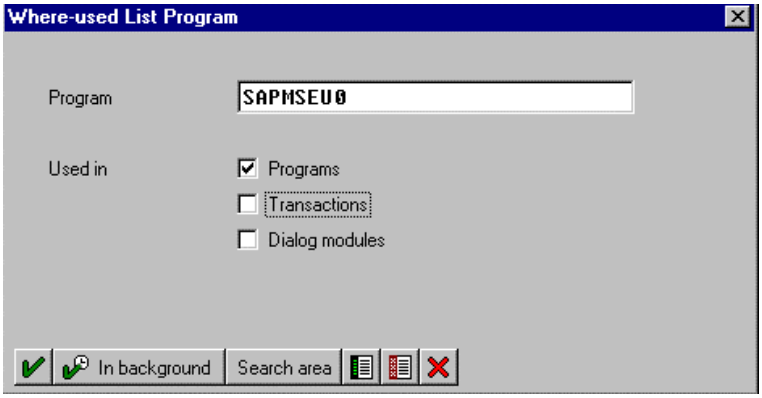
Suppose you want to determine the environment for development class FBK (vendors) You want to exclude development classes beginning with ‘S’ from your search.

1. Open the Repository Information System.
2. Select object type development classes under *Environment* → *Development coordination*
3. Choose *Environment*
4. Enter development class **`FBK`**.
5. Choose *Search area* and enter **`S*`** in the development class field.
6. Choose *Selection options*. In the dialog box, choose the appropriate option, then go back.
7. Choose *Execute*.
   The system displays a list of all external references to development class FBK that are not in development classes beginning with ‘S’.

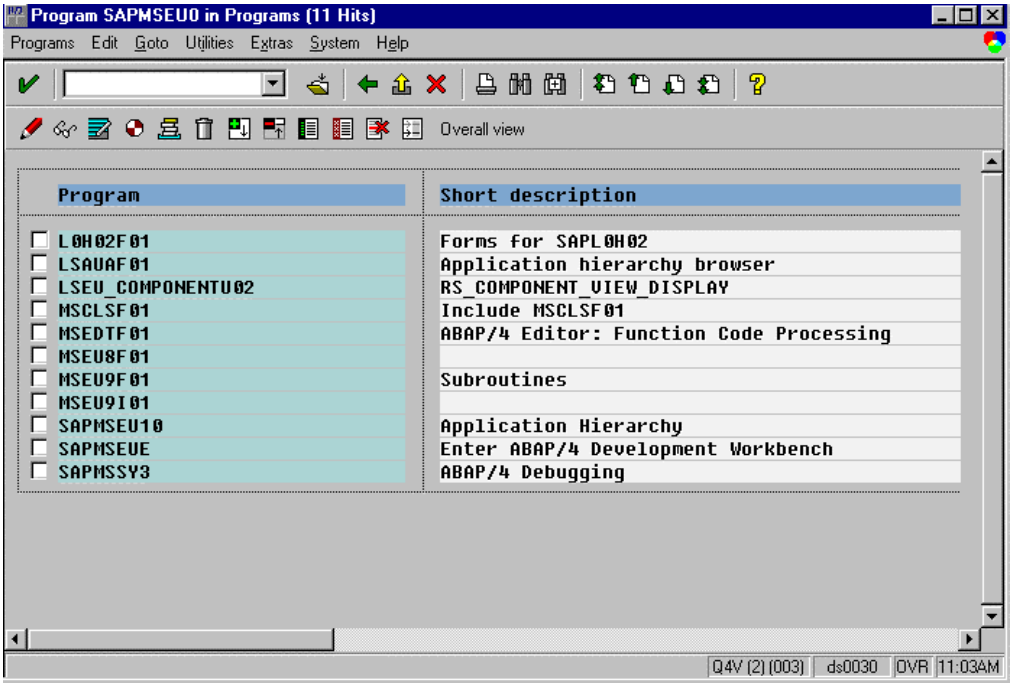
Where-used Lists

The Repository Information System tracks where an object is defined and where it is used. You can create a where-used list for an object in any of the Workbench tools by choosing *Utilities* → *Where-used list*.

When you choose the *Where-used list* function, the system asks you for search criteria. For example, if you are searching for a program, the system displays the following dialog:



The output from the *Where-used list* function is a hit list of each location where the object is used:



From the hit list, you can select a location and then go there by choosing to *Display* or *Change* the Object. If you choose *Found locations*, the system lists the line numbers where the object was found.

From Where-Used to Where Defined

From any object use in any Workbench tool, you can navigate from where an object is used to where it is defined. To do this, you simply double-click on the object name where it is used. The system takes you to where the object is defined.

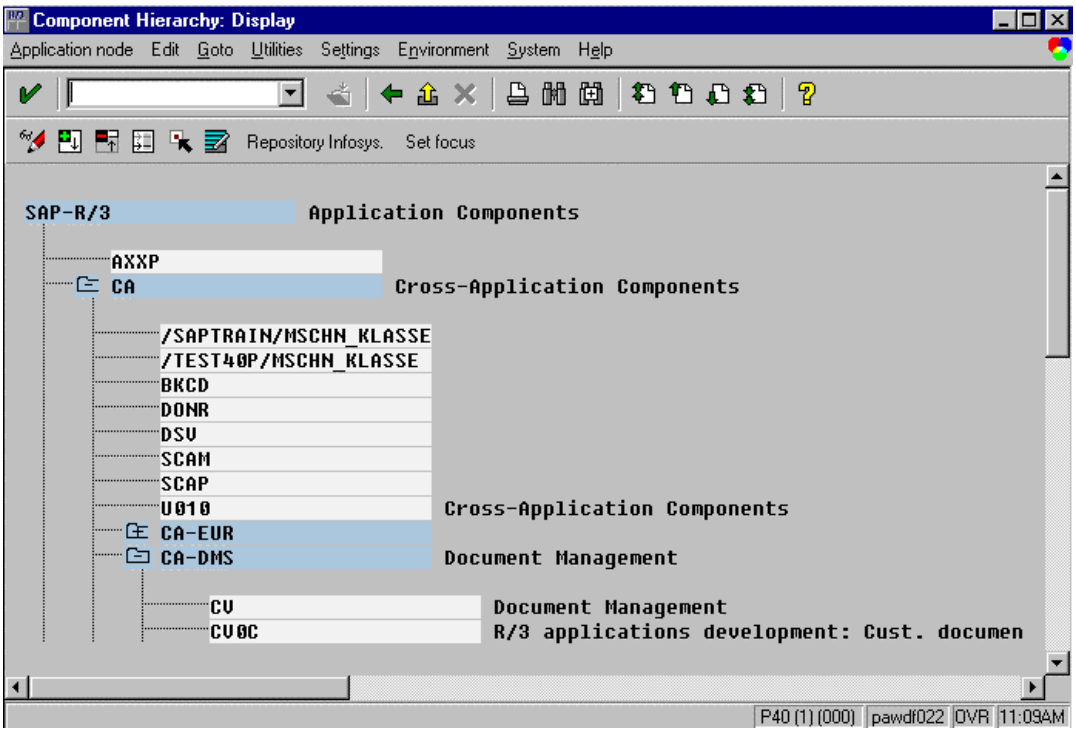
If you click on an object definition, the system takes you to where it is used. If the object is used in more than one place, the system prompts you to create a where-used list.

The Application Hierarchy

The Application Hierarchy displays the organization of all the applications in your R/3 system. You can use the Application Hierarchy to look at the hierarchy of SAP applications delivered with your system. Your company's application hierarchy is either created for you or, if you have the authorization, by you.

To start the application hierarchy from the Workbench initial screen, choose *Overview* → *Applic. hierarchy* → *SAP* or *Customer*.

The nodes in the application hierarchy are either title nodes or development nodes. Development nodes have an accompanying development class indicating there are actual objects associated with the node. Title nodes have no development class and are used to help visually organize the hierarchy.



Whenever you assign a development class to a title node, a new development node is created.

You can use the hierarchy to locate a development class when you know the application but not its class. This is useful, for example, when you are working in a team. If a programmer needs to update an old application that is new to him, he can find the application's development class by using the hierarchy.

Click on a development class to display a list of its objects.

Creating a Hierarchy

You can also use the Application Hierarchy to depict an existing application hierarchy or plan a new application hierarchy. For example, if you wanted to add a node, you would do the following:

1. Select a node.

The node you select will provide the top of your new hierarchy.

2. Choose *Application node →Create*.

   The system displays the *Create Node* dialog box.

3. Enter a node name or a name for a development class.

   The system adds a new node. If you entered a development class name, the system uses the class short text for the node title and adds the class name to the right of the node.

You can use this option to assign an existing class to a node or create a new class for a node.

Searching within an Application

You can branch directly from the application hierarchy to the Repository Information System. You use this function whenever you want to restrict your search to the objects in one or more specific applications. Example: Suppose you are looking for a particular domain used in a financial accounting application.

To branch from the Application Hierarchy to the Repository Information System:

1. Select one or more nodes by placing the cursor on each one and choosing *Select*.

   If you select a node, all of its subordinate nodes are also selected.
   To deselect a node, double-click it.

2. Choose *Repository Infosys.*

   The system opens the Repository Information System.

3. Use the Repository Information System [Page 496] as you normally would.

The Data Browser

You use the Data Browser to access table entries without using an ABAP program. With the Data Browser, you can:

- Display table records.
- Display all table field values and related text field values.
- Branch from table entries to their related check-table entries.

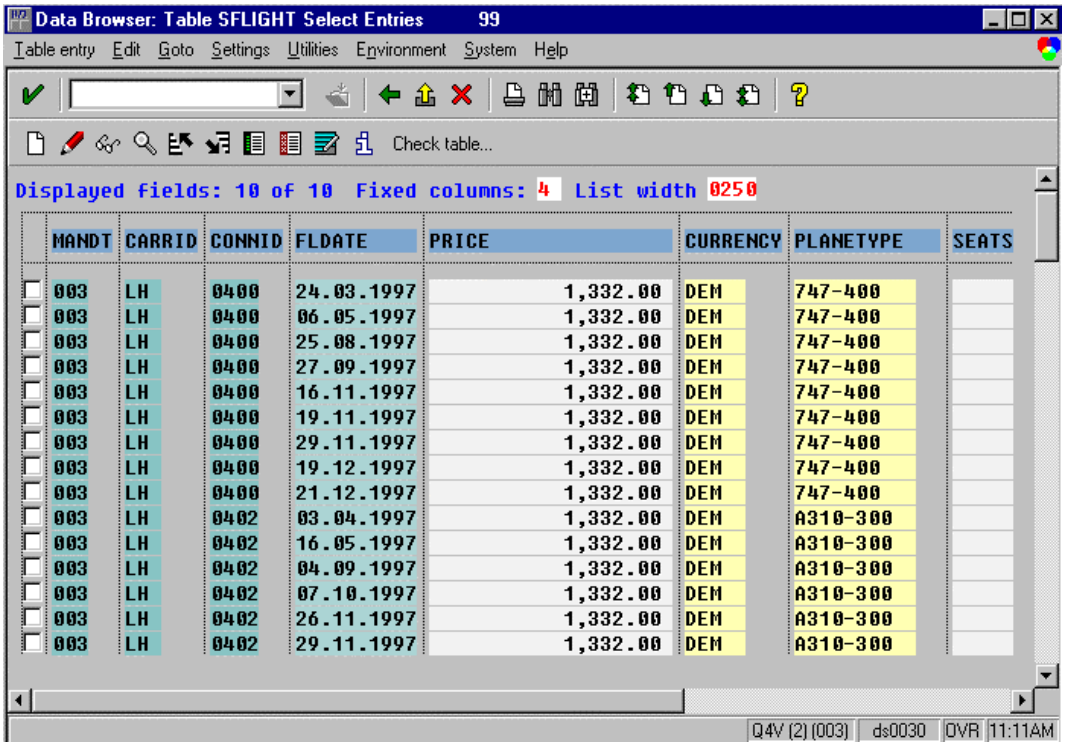
If a table has *Table maintenance allowed* set you can also create or update table records with the Data Browser.

Displaying a Table

To start the Data Browser, choose *Overview* → *Data Browser* from the Workbench tools initial screen. You can also reach the Data Browser from the *Environment* menu in the Repository Browser or the *Utilities* menu in the ABAP Editor. The Data Browser prompts you for a table name.

The selection screen lists the table's key fields, the remaining fields, and then a description of the table size. Enter selection criteria on the selection screen to restrict the number of entries that the system selects. To view the results of your selection, choose *Execute* from the selection screen. The system displays the results.

You can see in the results display that the number of records you selected appears in the title bar:



A status bar appears above the table display, containing the following information:

- Fields displayed: Shows the number of fields displayed and the actual number of fields in the table.
  To display more fields, enter a new value for the *List width*.
- Fixed columns: Shows the number of fixed columns. These columns do not move when you scroll horizontally through the table.
  You can enter a new value in the *Fixed columns* field to change the number of fixed fields.
- List width: Number of characters displayed in the list. The standard width is 250 characters. However, you can set any width you like between 50 and 250 characters.

Both the list and detail displays show fields in different colors, depending on the field's status. Choose *Utilities* → *Color key* to display the meanings of the different colors.

Customizing the Data Browser Display

The Data Browser contains various options allowing you to customize both the selection screen and the result screen.

The Selection Screen

To customize the selection screen and the Data Browser output, choose *Settings* → *User parameters.* The system displays the *Maintain Settings* dialog box.

Setting	Function
Width of output list	Changes the width of the Data Browser output.
Maximal no. of hits	Changes the number of entries displayed in the output. For example, setting this value to 100 means that Data Browser will display only the first 100 entries that match the search criteria.
Check conversion exits	Applies a table's conversion routines to the Data Browser output.
Display maximum number of hits	Displays the total number of records available that match your search.
Keyword group box	Changes how fields are labeled on the selection screen and in the output list. If you choose *Field Name* the Data Browser labels each field by its Dictionary field name. If you choose *Field text*, the field is labeled by its Dictionary description.

The system maintains your settings between SAP sessions.

Changing How Data is Sorted

You can change the way the Data Browser sorts output. To do this, choose either *Sort ascending* or *Sort descending* on the result screen. You can also change the primary sort field.

By default, the system sorts data using the key fields as input. To specify different fields to sort on, choose *Settings* → *List Format* → *Sort..* The system displays the *Enter table... sort field* dialog box. Enter a `1` by the field you want as a primary sort field, a **2** by the next sort field, and so forth. You can sort by up to `9` fields.

Limiting the Fields Displayed

You can limit the number of fields displayed in the selection screen. To do this, choose *Settings* → *Selection criteria* in the selection screen. The system displays the *Select Fields* dialog. Select the table fields you want to appear on the selection screen. By default, all the fields are selected.

You can also limit the fields displayed in the Data Browser output. To do this, choose *Settings* → *List format* → *Select columns* from the output display..

Other Data Browser Functions

To provide you with Data Browser capability, the system generates a table-handling program for each new table. This program is saved in the system and is used as long as the table definition remains the same. When the table changes, the system automatically regenerates the table-handling program.

Sometimes you may need to regenerate the table-handling program manually. Choose *Environment* → *Program generation* to regenerate the program.

Foreign-Key Relationships

Some fields have foreign key relationships with other tables. For these fields, you can also display the element from the other table that corresponds to the current table field. To do this, you first display a foreign-key table, then do the following:

1. Place the cursor on the field for which a foreign key is defined.
2. Choose *Environment* → *Check table*.

   The system displays the element of the check table that corresponds to the foreign key field you selected.

Other Concepts

Inactive sources is a concept affecting most of the development objects in the ABAP Workbench.
Business Add-Ins provide an additional means of enhancing the SAP Standard system.

See also:

Inactive Sources [Page 509]

Business Add-Ins [Page 527]

Inactive Sources

This documentation contains the following topics:

Concept

Terminology

There are three essential new terms used in the context of inactive sources:

Term	Meaning
Active version	The database version of a development object used to generate the runtime object
Inactive version	A saved database version of a database object that does not affect the runtime object (even after regeneration)
Inactive object list	The set of all inactive versions of development objects belonging to a particular user. When a user edits a new object, it is added to his or her inactive object list. When a user activates an object, it is removed from the inactive object list. Leaving aside local private objects ($TMP), a user's inactive object list is a genuine subset of all of the objects he or she is working on and that are included in open tasks administered by the Workbench Organizer.

Reasons

The introduction of inactive sources provides developers with a separate local view of the R/3 Repository, and is the basis for a "**local runtime system**".
Changes to development objects can be tested within this local system without disturbing the wider development environment.

The main advantage of this is that **the development process becomes seamless**. For example, it makes it possible for you to change the interface of a function module without the changes immediately becoming visible in programs that call it. The changes are only visible systemwide once the object has been activated

Furthermore, the concept **avoids redundant program generation**. Previously, the system always generated a new load version whenever you saved a program in the ABAP Editor. The introduction of inactive sources means that the program is not generated until you decide that it is appropriate to activate it.

The introduction of inactive sources is accompanied by a standardization of the working methods of the different ABAP Workbench tools. Consistency is also assured by the main program check that is performed whenever you activate an object.

Concept

- Objects are always **saved as inactive versions**.
  When you create or change a development object and then save it, an inactive version is written to the database.
- Inactive objects are **included in the user's inactive object list**.
  Development objects that have been edited and saved are placed in the inactive object list of the developer responsible. Each user has their own inactive object list, which other users cannot access directly. Users always work with their own personal inactive object list.

- You can **link inactive object lists**.
  If another user changes a development object and saves it, the object is included in his or her inactive object list.
- The ABAP Workbench tools always take into account the user's inactive object list.
  In display mode, the user always sees objects from his or her own inactive object list in their inactive version, but all other objects in their active version (even if an inactive version exists). This particularly applies to navigation within the ABAP Workbench.
  In change mode, the latest version is always displayed, regardless of whether it is included in the user's inactive object list or not.
  All tools display the current status of the object that you are currently working on. For further information, refer to Status Display [Page 516].
- You can display an overview of your work list and of all inactive objects in the system. For further information, refer to Overview of Inactive Objects [Page 515].
- You **activate your inactive object list** when you choose.
  Furthermore, you can decide whether to activate the whole inactive object list or just a part of it. When you activate an object, it is removed from the inactive object list. For further information, refer to Activating Objects [Page 514].
- Only the active version of an object is used to **generate** runtime objects.
- Support in the **Workbench Organizer**.
  All members of a project team can work on a single object that belongs to the change request in which they all have tasks. Consequently, all team members will see the inactive version of an object in display mode, if one exists.
  Users without a task in the change request cannot change the object, and therefore see the active version in display mode.
  You cannot release a transport request until all of its objects have been activated.

Support in the ABAP Workbench Tools

Tools that Support Inactive Sources

- ABAP Editor
- Class Builder
- Function Builder
- Screen Painter
- Menu Painter

In the ABAP Dictionary, the existing activation concept is still in place. ABAP Dictionary objects are saved in an inactive version, but do not appear in your inactive object list. For further information, refer to activation [Ext.] in the ABAP Dictionary.

Further Reading

Activating Objects [Page 514]

Overview of Inactive Objects [Page 515]

Status Display [Page 516]

Activating Classes and Interfaces [Page 519]

Inactive Sources and the Modification Assistant [Page 526]

Incompatible Changes

Tool	Changes
ABAP Editor	The 'Save' function always saves the inactive version of the object (was previously always the active version). The 'Check' function uses the inactive object list. New 'Activate' functions. New status display.
ABAP Dictionary	Changed descriptions in the status display
Function Builder	Previous 'Activate' function replaced. The new 'Activate' function allows you to activate the top include and form routine include as well as the function module. The 'Deactivate' function has been removed. New status display.
Screen Painter	The previous 'Generate' function has been replaced by 'Activate'. New status display.
Menu Painter	The previous 'Generate' function has been replaced by 'Activate'. New status display.

Activating Objects

Use

You can activate either your entire worklist, selected objects, or just components of one object (classes in ABAP Objects).

Prerequisites

Before activating an object, the system checks the syntax of the entire object (main program, function group, or class). Any syntax errors are displayed in a list. However, it is still possible to activate objects even if they contain errors. This can be useful if you want to create templates for coding generators.

Procedure

4. Select the relevant object in the object list.
5. Choose *Activate* from the context menu or the icon.
   Your worklist appears. The selected object is highlighted.
6. Choose  to confirm your selection.
   If you are activating an include that cannot be assigned to a single main program, the system asks you for a main program. Choose one main program from the list of programs that use the include.
   A message in the status bar informs you that the object has been successfully activated.

Result

When you activate an object, its syntax is checked. The check uses the inactive versions of the components selected for activation, but the active versions of all other components.
The inactive versions are used to create active versions of the objects. A new runtime version is then generated. Finally, the inactive version is deleted and removed from the list of inactive objects.

Special Features

When you activate an entire object from the object list, only the inactive objects belonging to that object are displayed in the worklist. However, you can display all of your inactive objects by choosing *All inactive objects*.

Overview of Inactive Objects

Overview

To display the overview, choose *Environment* → *Inactive objects* anywhere in the ABAP Workbench.

Use

You can choose to display various sets of inactive development objects within the system:

Choose	To
	Activate the selected objects
 *Request/task*	Display a list of all of the inactive objects that are assigned to a task (request) in the Workbench Organizer
 *User*	Display the inactive object list of the specified user
 *Object*	Display all inactive components of a main program or function group
	To display all inactive objects in the system

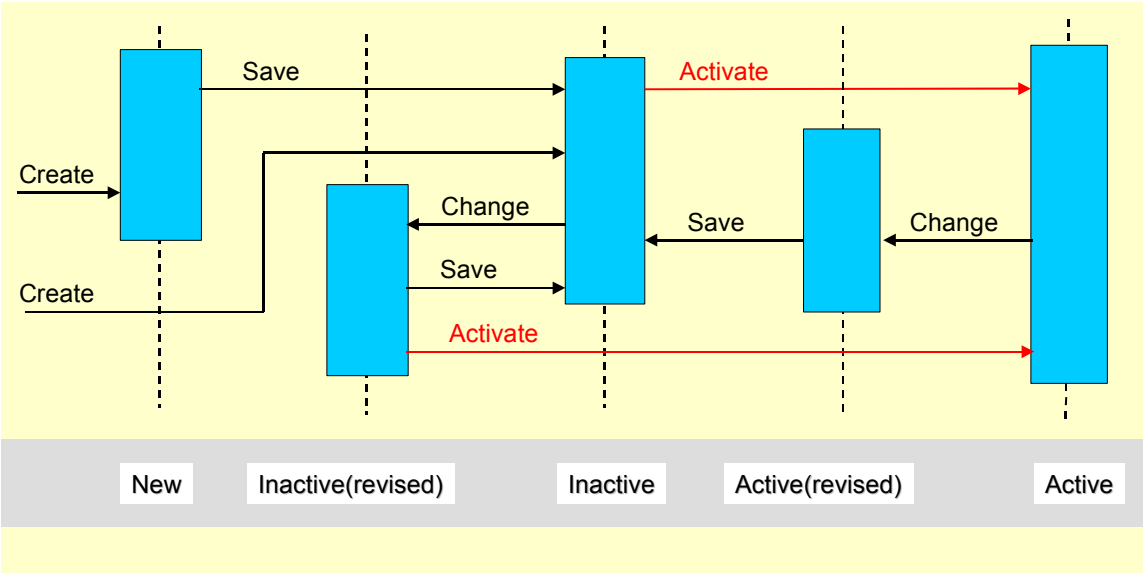
Once you have selected an object, you can open it using the relevant tool and display or change it.
You can also place objects from another user in your own inactive object list ("networked inactive object lists") and activate them.

Status Display

The current status of a development object is always displayed in the tool with which you are editing it.
The status indicates:

- Whether a database version exists for an existing development object (new ↔ active/inactive).
- Whether the current state of the object corresponds to the state of the object in the database (*revised* ↔ *saved*).
  The status *saved* is not explicitly displayed. Objects are always saved as *inactive*.
- Whether the database version is inactive or active (*inactive* ↔*active*).

The following versions of development objects may appear in the status display:



When you create a new object, there are two possible status displays. If, for example, you create a function module, it is already saved, and therefore has a corresponding inactive version in the database. The status display in the Function Builder therefore says *Inactive*.
On the other hand, a new GUI status does not exist in the database until you save it, and consequently has the status *New* in the Menu Painter.

Another source code version occurs immediately after you have loaded a program from temporary storage; in this case, the status *is Temp. version (changed)*.

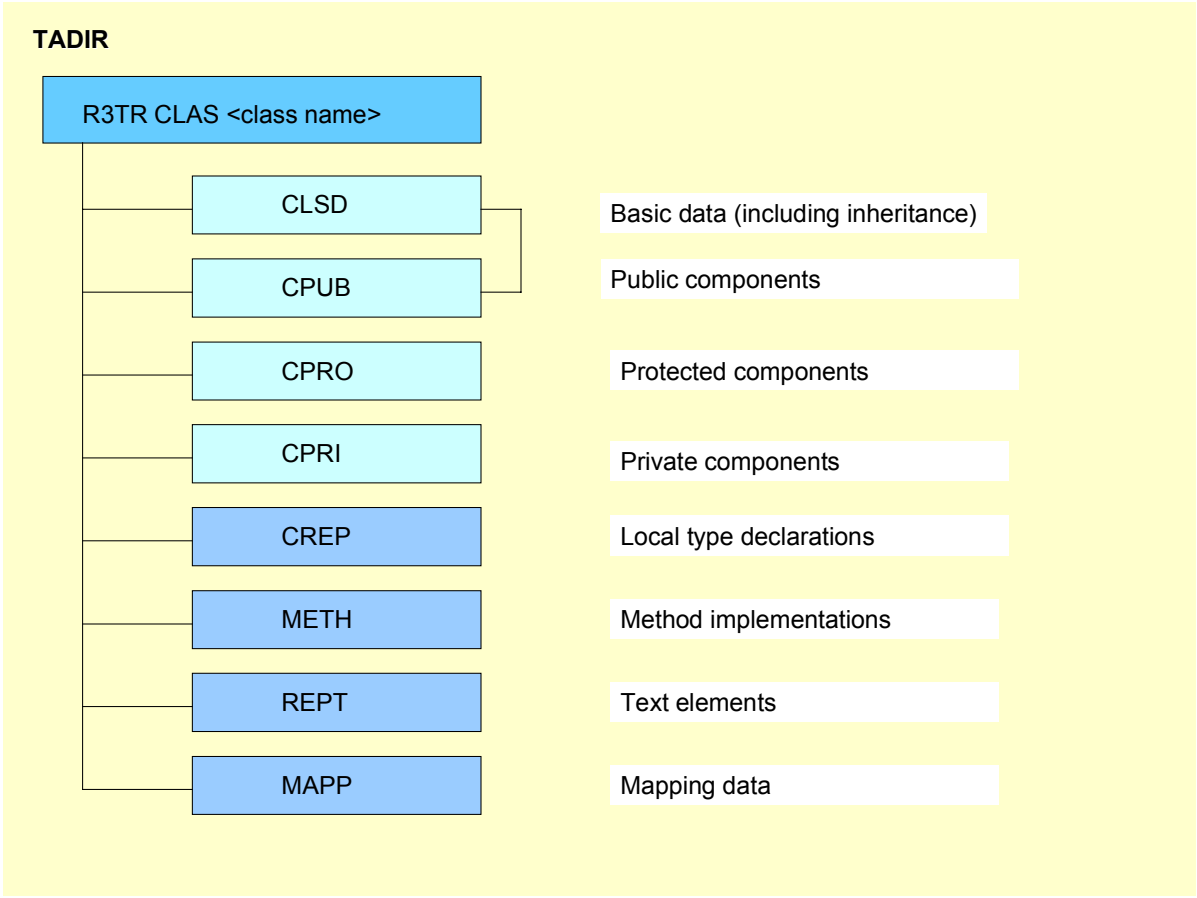
Activating Classes and Interfaces

Significance of Activation

When you create runtime instances, the system always uses active sources. You should remember this when instantiating classes (CREATE OBJECT statement), since this always refers to the activated class. All components of the corresponding global class that you want to access in the calling program must be activated explicitly.

Components of Global Classes

All global classes have an entry in table **TADIR**. The corresponding transport object for a class has the name **R3TR CLAS <class name>** and contains a range of components, each of which is a separate transport unit. Inactive class components appear in your worklist.



- The basic data and public components of a class cannot be activated separately.
- Only the basic data and public, protected, and private sections of a class affect the status display in the class editor. If you activate the entire transport object and then change a method implementation, the status remains *active*.

Components of Global Interfaces

The transport object for an interface has the name **R3TR INTF <interface name>**. It contains a single object with the name **INTF.**

When you activate a class that implements an interface, you must ensure that the interface has already been activated. Otherwise, the public section of the class contains a syntax error.

Status Display in the Class Builder

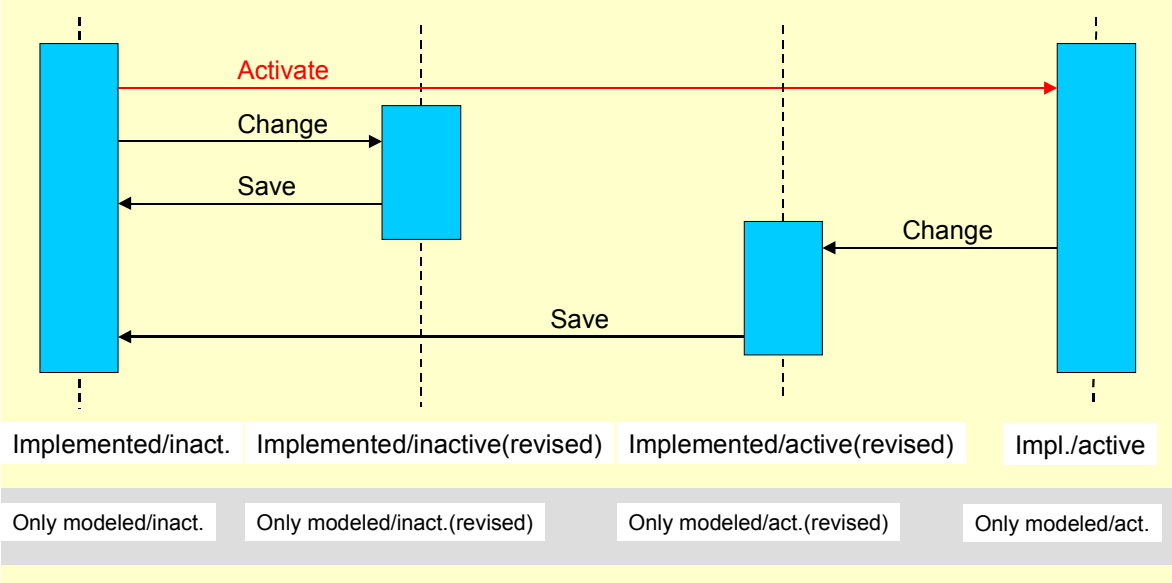
The current status of a class or interface is always displayed in the class editor. It is determined by the:

- Runtime relevance (*Implemented* ↔ *only modeled*)

Activating Classes and Interfaces

- Database state (*Revised* ↔ *Saved*)
- Activation (*inactive*↔*active*)

There are eight possible statuses of classes or interfaces that can appear in the class editor:



Effect of Inactive Sources on Operations

The inactive sources concept affects the following operations within the ABAP Workbench:

Save	Saves the object in an inactive version without a syntax check. Enters the object in your inactive object list.
Activate	Creates an active version from the existing inactive version. If an object contains components (like classes in ABAP Objects), you can activate individual components. Before activating the object, the system checks the syntax of the entire object, then creates an active version, generates a runtime version, and deletes the corresponding entry from your inactive object list.
Generate	Creates a new runtime version from the existing active version. Unlike the 'Activate' function, this function only generates a new load version.
Display Active/Inactive Sources	If an object exists in both active and inactive versions, you can switch between the versions in the ABAP Workbench tools.
Delete	Both active and inactive versions are deleted. Note that when you delete components of a global class in the Class Builder, they reappear in your inactive object list with the "Delete" icon. They are not finally deleted or removed from your inactive object list until you activate the relevant objects.
Copy	The system uses the active version of the source object, except in the Function Builder, where it asks you which version you want to copy if both an active and an inactive version exist). The new copy is always inactive.
Rename	Applies to both the active and inactive versions of an object.
Syntax check	Uses your inactive object list.
Execute	A runtime object can only be generated from a syntactically-correct active version. The inactive version of an executable program can be run from the ABAP Editor. If you execute a program from the object list, there must be an active version.
Transport	Only active objects can be transported. You cannot release a transport request until all objects have been activated.

Further Effects

The list below contains other functions that are affected by inactive sources:

Object list (SE80)	Displays all active and inactive objects. If an inactive version exists, the object is highlighted.
Status display	See Status display [Page 516] in the tools.
Where-used list	Like the object list, the where-used list is global, not user-specific. It is based on all objects.
Navigation	Navigation is user-specific, and takes your inactive object list into account. If an inactive version of an object exists and the object is in your inactive object list, you will see that inactive version. Otherwise, the active version is always displayed. This does not apply only to navigation in the various ABAP Workbench tools, but also from the object list or from a where-used list.
Debugging	The Debugger always displays the active version of an object.
Modifications	See Inactive Sources and Modifications [Page 526].

Inactive Sources and Modifications

This section explains how inactive sources work in conjunction with the Modification Assistant and how you can make modifications to global ABAP classes and interfaces.

Modifications

If you modify an original component of the SAP standard system without using the Modification Assistant, inactive sources are effective in all tools. Consequently, the modified object is saved in its **inactive version**.

If you modify an original component of the SAP standard system using the Modification Assistant, inactive sources are not currently supported. Modified components, with the exception of ABAP Dictionary objects, screens, and GUI statuses, are saved in their **active version**. In the ABAP Dictionary, the conventional 'Activate' function is still relevant. Similarly, in the Screen Painter and Menu Painter, the old 'Generate' function must still be used.

Modifications and ABAP Objects

Global interfaces, classes, and their components **are not currently supported by the Modification Assistant**. However, inactive sources already apply when you change global classes and interfaces in the Class Builder. If you modify a method implementation in a global class supplied by SAP, the changes are stored in an **inactive version**.
Modifications to global classes or interfaces and their components have to be adjusted in an upgrade in the conventional way.

Business Add-Ins

Business Add-Ins are a new SAP enhancement technique based on ABAP Objects. They can be inserted into the SAP System to accommodate user requirements too specific to be included in the standard delivery. Since specific industries often require special functions, SAP allows you to predefine these points in your software.

As with customer exits (SMOD/CMOD [Ext.]), two different views are available:

- In the definition view, an application programmer predefines exit points in a source that allow specific industry sectors, partners, and customers to attach additional software to standard SAP source code without having to modify the original object.
- In the implementation view, the users of Business Add-Ins can customize the logic they need or use a standard logic if one is available.

In contrast to customer exits, Business Add-Ins no longer assume a two-system infrastructure (SAP and customers), but instead allow for multiple levels of software development (by SAP, partners, and customers, and as country versions, industry solutions, and the like). Definitions and implementations of Business Add-Ins can be created at each level within such a system infrastructure.

SAP guarantees the upward compatibility of all Business Add-In interfaces. Release upgrades do not affect enhancement calls from within the standard software nor do they affect the validity of call interfaces. You do not have to register Business Add-Ins in SSCR.

The Business Add-In enhancement technique differentiates between enhancements that can only be implemented once and enhancements that can be used actively by any number of customers at the same time.

In addition, Business Add-Ins can be defined according to filter values. This allows you to control add-in implementation and make it dependent on specific criteria (on a specific *Country* value, for example).
All ABAP sources, screens, GUIs, and table interfaces created using this enhancement technique are defined in a manner that allows customers to include their own enhancements in the standard.

A single Business Add-In contains all of the interfaces necessary to implement a specific task. In Release 4.6A, program and menu enhancements can be made with Business Add-Ins.
The actual program code is enhanced using ABAP Objects. In order to better understand the programming techniques behind the Business Add-In enhancement concept, SAP recommends reading the section on ABAP Objects [Ext.].

More information about Business Add-Ins is contained in the following sections:

Business Add-Ins: Architecture [Ext.]

A Comparison of Different Enhancement Techniques [Ext.]

Defining Business Add-Ins [Ext.]

Calling Add-Ins from Application Programs [Ext.]

Implementing Business Add-Ins [Ext.]

Filter-Dependent Business Add-Ins [Ext.]

Multiple Use Business Add-Ins [Ext.]

Business Add-Ins

Menu Enhancements [Ext.]

Business Add-Ins: Import Procedure [Ext.]