New Features in SAP Web Application Server 6.40 for ABAP Developers

Regular Feature

Under Development

Karl Kessler, SAP AG

While much of the development news from SAP in 2003 was focused on Java, the year also brought lots of new benefits for ABAP developers. SAP Web Application Server (SAP Web AS) 6.40, to be released to customers in 2004, touches on and enhances almost every aspect of ABAP development — from the language itself, to tools, to application structure and design, to monitoring. This article introduces just some of the new features targeted for SAP Web AS 6.40, including:

- ✔ New possibilities for structuring and accessing data with shared objects. Shared objects allow for copy-free access to complex ABAP variables in shared memory.
- ✔ A tool to analyze memory consumption in long-running transactions such as a customer interaction centers.
- ✔ A redesigned and enhanced Debugger tool.
- ✔ The new ASSERT statement.
- ✔ Capabilities for activating multiple assertions and breakpoints with a single command.

ABAP Language Enhancements: Shared Objects

Shared objects, as the name suggests, are a new kind of ABAP object that allow objects to be kept and managed in shared memory. They provide a dramatic change for structuring and accessing data in ABAP applications.

With the SAP R/3 application server (the classic ABAP runtime environment), shared memory was primarily used to manage the ABAP program cache and user session information (roll area[1]), as well as a variety of buffers for the central data — such as customizing tables, factory calendar, etc. — that is shared among applications. For developers, programmatic access to shared memory was quite limited: ABAP statements EXPORT TO SHARED BUFFER or EXPORT TO SHARED MEMORY only allow for value-oriented data types (fields, flat structures, internal tables) and make for some tedious tasks during application development (e.g., displacement of shared data in the buffer).

Consider, for example, how you would set up a complex catalog for cross-application use. In typical ABAP development, this would require the export (i.e., copy) of data from the roll area of a "writer" transaction to shared memory. Since only flat data types are supported, the catalog object needs to be split up into tables, structures, and fields. Later "reader" transactions must import (i.e., copy) the data back to their roll area. In other words, a reader transaction can only access information by copying the data from the shared buffer to its own roll area.

[1] Memory area of a fixed (configurable) size that stores the session context.

To overcome these limitations, SAP Web AS 6.40 introduces *shared objects* for ABAP. Now, rather than managing a collection of flat data structures and copying information between transactions, you can use objects that include references to several other objects, stemming from a *root object* (see **Figure 1**). These objects allow you to store ABAP variables of any kind in shared memory for copy-free read access.

With shared objects, you can:

- Reduce copy operations, since objects are directly accessed in shared memory and can be shared among different transactions. You'll find a good example — in fact, one of the early pilots of shared objects — in the 6.40 Workbench itself, which was built using shared objects. Navigation of the Workbench is now 100 times faster at first access and saves 3 MB session memory per user.[2]

- Avoid data inconsistencies, especially in data structures spread across multiple internal tables where managing and maintaining foreign key relationships between tables can be cumbersome.

Defining Shared Memory Areas

Shared objects are managed in *shared memory areas* that you define at design time with transaction SHMA. Each shared memory area manages one or several area instances.

Each shared memory area instance is self-contained — no references to other area instances or variables of the roll area are allowed. When you create a shared memory area (CL_SANA_SHM_AREA in **Figure 2**), you specify a class name to be generated that inherits from the system class CL_SHM_AREA. When you define a shared memory area, you must specify an associated root class. A shared object is always accessed by a root object that can contain references to subordinate objects.

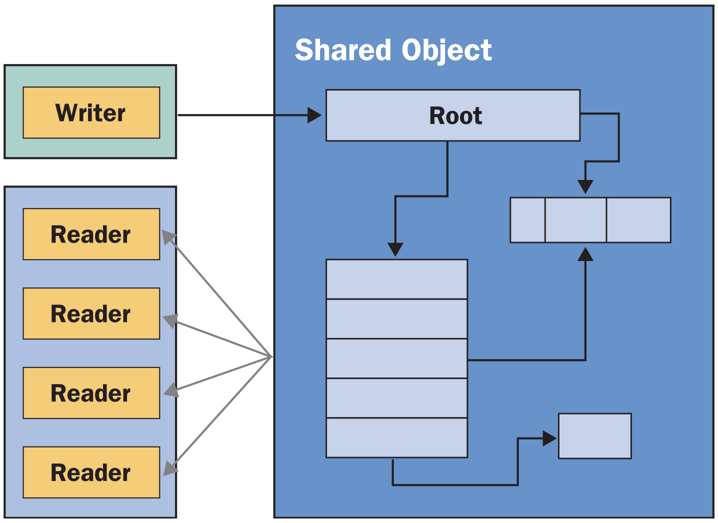
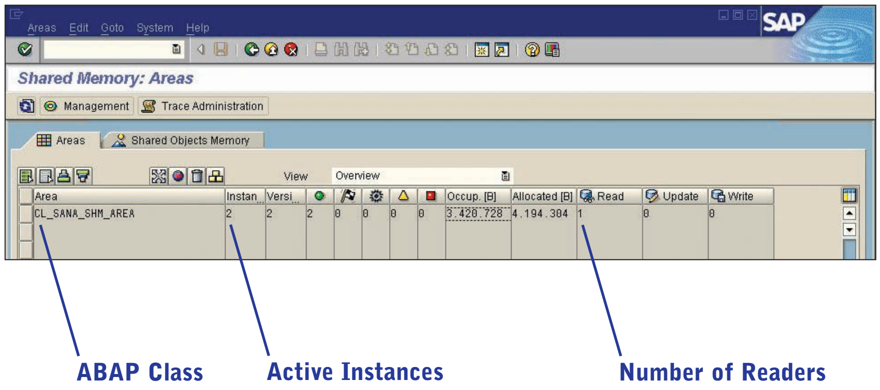


Figure 1 Use Shared Objects to Provide Stable, Copy-Free Access to Central Data



Shared Memory: Areas

View: Overview

Area	Instan...	Versi...	●			△	🟥	Occup. [B]	Allocated [B]	Read	Update	Write
CL_SANA_SHM_AREA	2	2	2	0	0	0	0	3.420.728	4.194.304	1	0	0

Figure 2 A View into a Shared Memory Area with the Shared Memory Monitor

Creating Shared Objects

To create shared objects, simply use the CREATE OBJECT statement with an AREA HANDLE addition, and make sure the underlying class is "shared-enabled." The area handle allows you to access the object graph starting at the root object (for more on handles, see the sidebar on the following page).

In a typical code sequence, for example, you first attach the running transaction to a shared memory area in order to gain a handle. Then, you create an object in shared memory associated with that handle and make it the root object for the shared memory area instance. From there, you would simply use ABAP Objects code to manipulate the shared object as you would any other object.

There is also a new shared memory monitor (transaction SHMM, shown in Figure 2), which allows you to view all shared memory areas and instances created therein (e.g., see any locks or navigate the content).

New ABAP Tools

SAP Web AS 6.40 offers new tools and features to enhance developers'

[2] Other examples are the metadata repositories in SAP BW and the Workflow Engine, to name a few.

Additional Information on Creating and Updating Shared Objects

Attach and Detach with Handles

If a transaction wants to access a shared object, it creates a *handle* in the roll area for identification and *attaches* to the shared memory area instance containing the shared object. When you detach from the shared memory area instance, the shared object becomes unavailable, and you must indicate whether you want to commit or roll back your changes.

Locking Prevents Read/Write Conflicts

Simultaneous access of multiple readers is managed through read locks. Shared objects can be written or updated with locking. When a writer wants to update or insert a shared object, the complete area instance is locked.

✔ Note!
The coarse-grained locking concept for shared objects works when the frequency of updates is low compared to read access (for example, a complex catalog of objects where updates are done on a daily basis). For this reason, OLTP access would not be a good candidate for a shared-objects approach.

Advanced Features with Shared Objects

Shared objects include advanced features, such as versioning, which can be used to better indicate the status of shared objects. Typical version states are "UNDER CONSTRUCTION", "ACTIVE", "OUT-OF_DATE", or "EXPIRED". Thereby transactions accessing the shared object can better interpret the shared memory contents and prepare themselves for corresponding clean-up procedures.

productivity and adapt to the new ways ABAP applications are being used.

Memory Inspector

Traditionally, a typical ABAP program was executed as a short transaction, so a detailed analysis of memory consumption wasn't necessary. But now, with long-running transactions in applications like customer interaction centers, the Memory Inspector helps to analyze memory consumption in your application over time. Simply switch on memory-consumption tracing by entering the OK-code `/hmusa`, then use the same code to switch off the tracing later.

The S_MEMORY_INSPECTOR transaction lets you examine the trace results providing detailed information on allocated internal tables, strings, objects, and anonymous data objects (see **Figure 3**). For instance, suppose you want to access parts of your application to make changes to a development object. Using Memory Inspector you take a look and find 263 classes that have been loaded, and inside those programs and classes are 939 table bodies. There are also 3,166 strings — which is rather high — so that's memory you'll want to inspect in more detail. Perhaps you're using too many tables inside your program, or wasting space inside the tables. You can even inspect each individual table, or look at a class, class pool, names of strings, and so on.

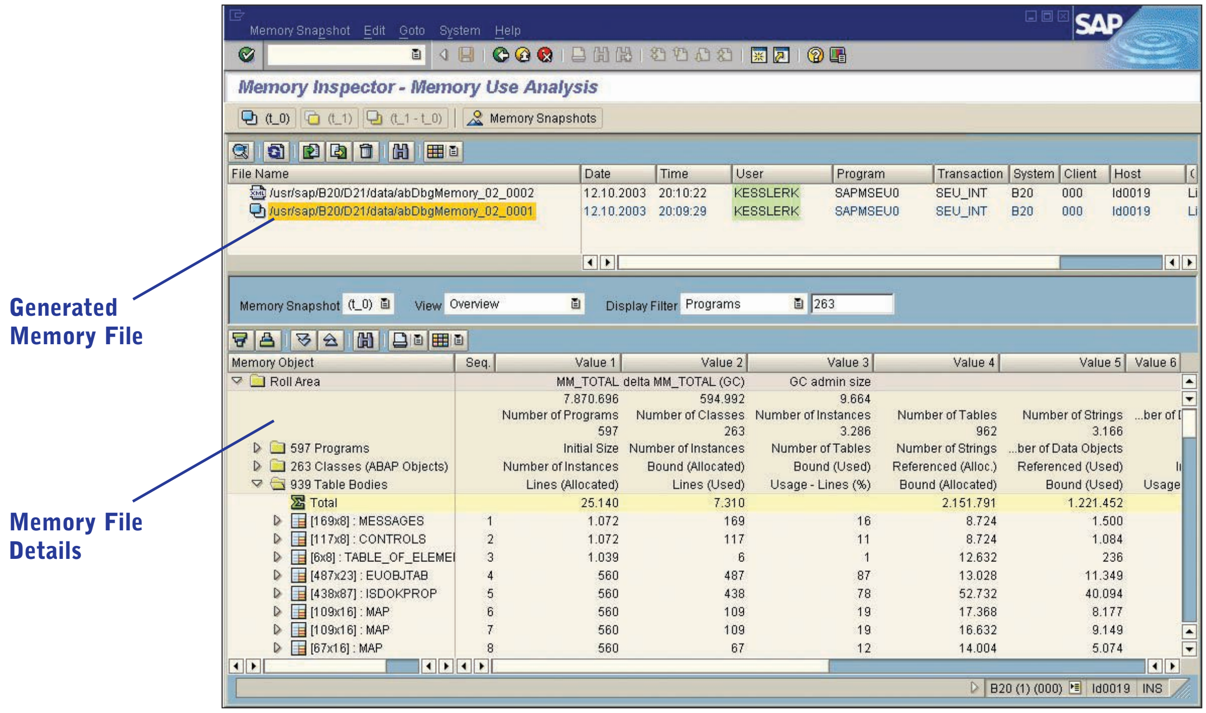
New Design for the Debugger

In SAP Web AS 6.40, the new design of ABAP Debugger cleanly separates it from any program you are debugging. By separating the control of the Debugger from your other applications, you can ensure that the results of the Debugger program do not interact with the program you're checking. However, note that this separation leads to a slightly lower performance, since the new Debugger runs in its own main mode.

Activatable Assertions and Breakpoints

Closely related to the new Debugger are new assertions and activatable breakpoints. *Assertions* are logical conditions inserted into the code that should, from a design-time perspective, always be true. A classic example: when executing a POP operation, the stack should not be empty. With ASSERT statements, if the condition is not met during execution, the program is automatically stopped to analyze its current state.

A new feature also allows you to group breakpoints or assertions under a particular ID, as in the ASSERT example in **Figure 4**. This grouping is especially helpful in a support situation where you must activate a set of breakpoints that stops program execution at critical points in an application — now they can all be identified by the same breakpoint ID. To define activatable breakpoints or assertions, use transaction SAAB.



Memory Inspector - Memory Use Analysis

File Name	Date	Time	User	Program	Transaction	System	Client	Host
/usr/sap/B20/D21/data/abDbgMemory_02_0002	12.10.2003	20:10:22	KESSLERK	SAPMSEU0	SEU_INT	B20	000	ld0019
/usr/sap/B20/D21/data/abDbgMemory_02_0001	12.10.2003	20:09:29	KESSLERK	SAPMSEU0	SEU_INT	B20	000	ld0019

Memory Snapshot (t_0) View Overview Display Filter Programs 263

Memory Object	Seq.	Value 1	Value 2	Value 3	Value 4	Value 5	Value 6
Roll Area		MM_TOTAL	delta MM_TOTAL (GC)	GC admin size			
		7.870.696	594.992	9.664			
		Number of Programs	Number of Classes	Number of Instances	Number of Tables	Number of Strings	...ber of [
		597	263	3.286	962	3.166	
597 Programs		Initial Size	Number of Instances	Number of Tables	Number of Strings	...ber of Data Objects	
263 Classes (ABAP Objects)		Number of Instances	Bound (Allocated)	Bound (Used)	Referenced (Alloc.)	Referenced (Used)	I
939 Table Bodies		Lines (Allocated)	Lines (Used)	Usage - Lines (%)	Bound (Allocated)	Bound (Used)	Usage
Total		25.140	7.310		2.151.791	1.221.452	
[169x8] : MESSAGES	1	1.072	169	16	8.724	1.500	
[117x8] : CONTROLS	2	1.072	117	11	8.724	1.084	
[6x8] : TABLE_OF_ELEMEI	3	1.039	6	1	12.632	236	
[487x23] : EUOBJTAB	4	560	487	87	13.028	11.349	
[438x87] : ISDOKPROP	5	560	438	78	52.732	40.094	
[109x16] : MAP	6	560	109	19	17.368	8.177	
[109x16] : MAP	7	560	109	19	16.632	9.149	
[67x16] : MAP	8	560	67	12	14.004	5.074	

B20 (1) (000) ld0019 INS

Figure 3 Memory Inspector

```
METHOD sort_by_name.
*       sort algorithm
      ASSERT ID test_sorted
            CONDITION itab->is_sorted_by_name( ) <> ' '.
ENDMETHOD.
```

Figure 4 ASSERT Statement

Conclusion

Over the past few decades, ABAP has proven itself as a programming language, with a powerful set of development tools and a high level of maturity when it comes to enterprise-level applications.

But as with all things, even with an established track record there's always room for enhancements. SAP's commitment to continuing improvements to the ABAP stack is ongoing. And while advances in the Workbench and to the ABAP development platform have always been designed for greater productivity and convenience for developers, these capabilities also now offer unprecedented possibilities for your applications — whether they're designed for the SAP GUI, portals, Web, or third-party applications.

For more information on SAP Web AS 6.40 for ABAP development, see **http://service.sap.com/netweaver.**

Karl Kessler joined SAP in 1992. He is the Product Manager of the SAP NetWeaver foundation including SAP Web Application Server, Web Dynpro, ABAP Workbench and SAP NetWeaver Developer Studio, and is responsible for all rollout activities. You can reach him via email at karl.kessler@sap.com.