LOGALI

TEORÍA
Modulatizar programas

SAP ABAP Programación

Contenido

1. Introducción

Una **unidad de modularización** es una parte de un programa en la que se encapsula una función concreta. Una parte del código fuente se almacena en un módulo para mejorar la transparencia del programa, para poder usar la función correspondiente del programa varias veces sin tener que volver a implementar el código fuente entero en cada ocasión (consulte el gráfico anterior).

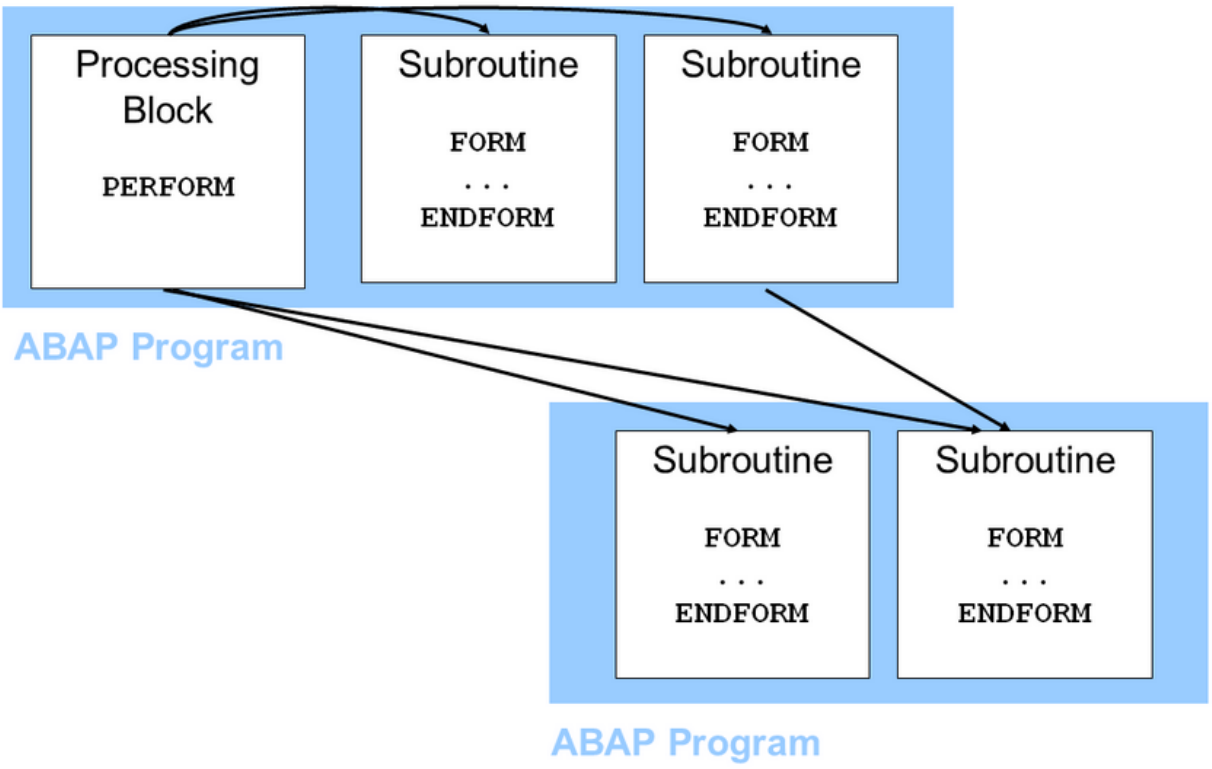
El resultado obtenido cuando un programa pasa a estar más orientado a las funciones es la mejora de la transparencia: el programa divide la tarea global en subfunciones, que son responsabilidad de las unidades de modularización correspondientes.

La modularización facilita la actualización de programas, ya que permite que las modificaciones se realicen únicamente en la función, o las correcciones en las unidades de modularización respectivas, y no en varios puntos del programa principal. Asimismo, también le permite procesar una llamada “como unidad” en el debugger durante la ejecución del programa y visualizar posteriormente el resultado. De este modo, resulta mucho más fácil encontrar el origen del error.

2. Modularización de programa local

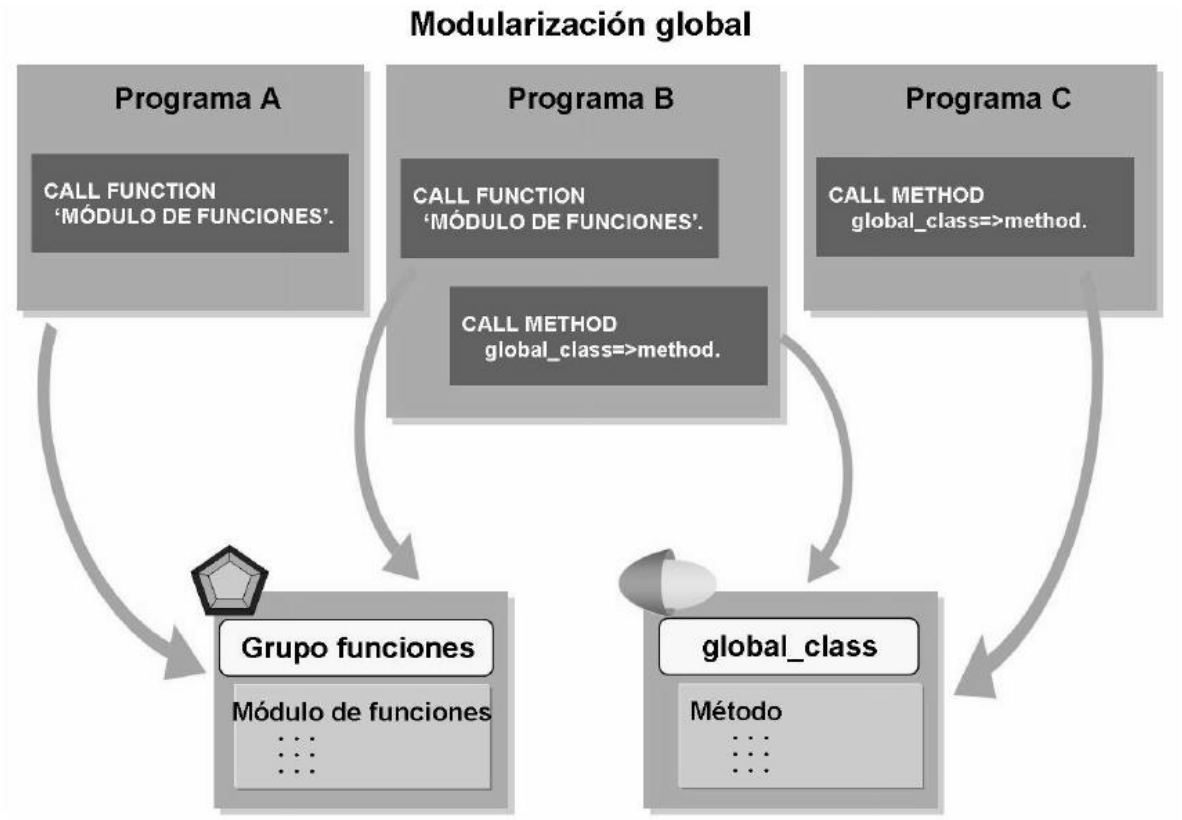
Existen dos técnicas para la modularización de programa local en el lenguaje de programación ABAP:

- **Subrutinas**, también conocidas como rutinas **FORM**
- **Métodos** en clases locales



Con ambas técnicas de modularización local, las unidades de modularización sólo están disponibles en el programa en el que se implementan. Para llamar al módulo local, no es necesario cargar ningún otro programa en el contexto de usuario en tiempo de ejecución. Las clases locales, los métodos y las subrutinas pueden tener el mismo nombre en diferentes programas sin causar conflictos. Esto se debe a que el código fuente de los programas se trata por separado en la memoria principal del servidor de aplicación.

3. Modularización global



También existen dos técnicas para la modularización global en el lenguaje de programación ABAP:

- Módulos de funciones que están organizados en grupos de funciones
- Métodos en clases globales

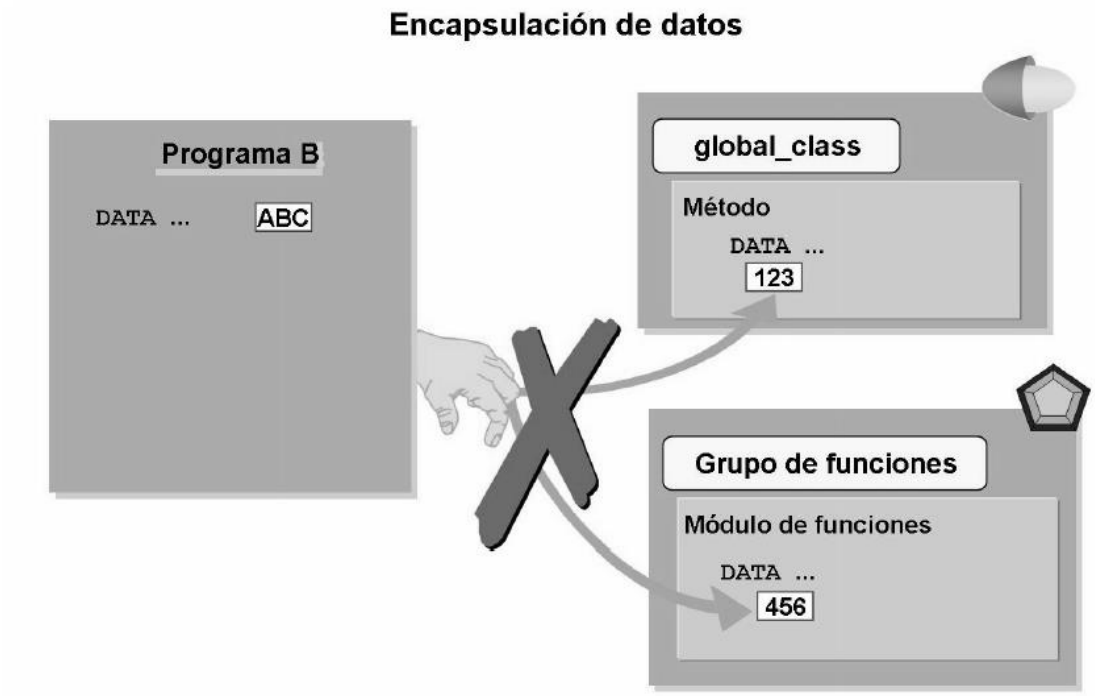
Cualquier número indefinido de programas puede utilizar simultáneamente las unidades de modularización definidas globalmente. Las unidades de modularización definidas globalmente se almacenan centralmente en el Repository y se cargan cuando son necesarias (es decir, cuando se llaman) el contexto del programa de llamada.

4. Encapsular datos

Lo ideal es que las unidades de modularización a las que se llama no usen los objetos de datos del programa de llamada directamente. Y viceversa: el programa de llamada no debería modificar directamente los datos de las unidades de modularización. Este principio se conoce como encapsulación de datos.

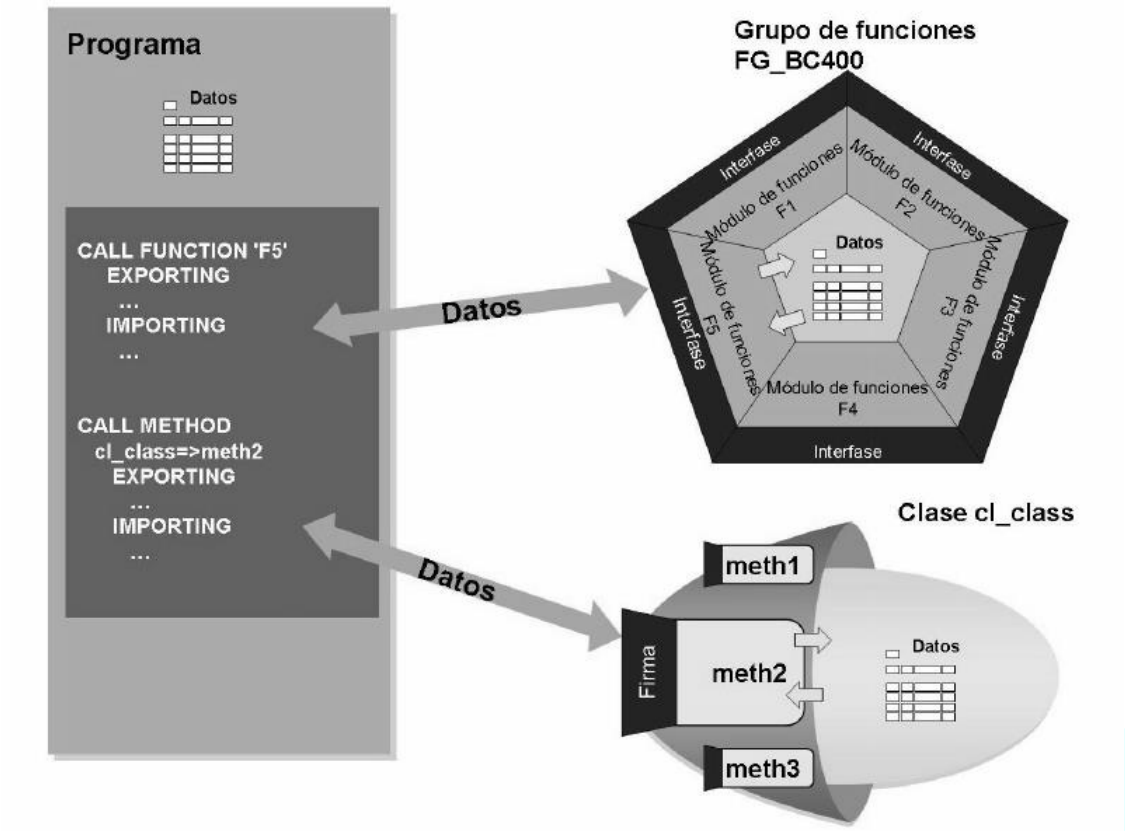
La encapsulación de datos supone una ayuda importante a la hora de desarrollar un código fuente transparente y actualizable. Hace que sea más fácil comprender en qué partes del programa se ha modificado el contenido de los objetos de datos. Además, garantiza que los datos de las unidades de modularización se modifiquen de modo coherente si, por ejemplo, el contenido de varios objetos de datos dentro de una unidad de modularización son dependientes mutuamente.

Tomaremos como ejemplo una unidad de modularización en la que se procesa una serie de documentos factura. El hecho de introducir números de factura diferentes para posiciones de factura que van juntas podría tener consecuencias graves. Si fuera posible acceder desde el exterior a las posiciones de factura individuales, esto es lo que sucedería. Entonces sería difícil para los responsables de la unidad de modularización determinar la causa de la inconsistencia de datos.



5. Transportes de datos, parámetros e interfaz

Los parámetros se utilizan para intercambiar datos entre el programa y el módulo. El número total de parámetros de una unidad de modularización se llama interfaz o firma. Los parámetros disponibles se determinan al definir la unidad de modularización. Los parámetros se diferencian en función de si se utilizan para pasar datos a la unidad de modularización (importar parámetros), para devolver datos de la unidad de modularización al programa de llamada (exportar parámetros) o para pasar datos a la unidad de modularización y devolverlos después de su modificación (modificar parámetros).



6. Modularización con subrutinas

Estructura de una subrutina

- Una subrutina se introduce con FORM.
- El nombre y la interfaz de la subrutina se especifican después de FORM.
- A continuación, figuran las sentencias de la subrutina.
- La sentencia ENDFORM concluye la subrutina.

En la definición de interfaz se crea una lista de los parámetros formales de la subrutina (f1,f2,f3,...) y se escriben si es necesario. Se debe escribir el tipo de transferencia requerido para cada parámetro:

```
REPORT ...
DATA: a TYPE ... ,
      b TYPE ... ,
      c TYPE ... .
  :
PERFORM xyz

    USING
      a
    CHANGING
      b
  :   c.
```

Llamada

```
FORM xyz
  USING
    value(f1) ...     " Llamar por valor
  CHANGING
    value(f2) ...     " Llamar por Valor y Resultado
          f3  ... .   " Llamar por Referencia
  ...
ENDFORM.
```

Definición

6.1. Llamar por valor

Se crea una lista de cada uno de los parámetros formales que deba tener el tipo de transferencia “Llamar por valor” con el prefijo VALUE en USING.

6.2. Llamar por valor y resultado

Se crea una lista de cada uno de los parámetros formales que deba tener el tipo de transferencia “Llamar por valor y resultado” con el prefijo VALUE en CHANGING.

6.3. Llamar por referencia

Se crea una lista de cada uno de los parámetros formales que deba tener el tipo de transferencia “Llamar por referencia” sin el VALUE prefijo debajo de CHANGING.

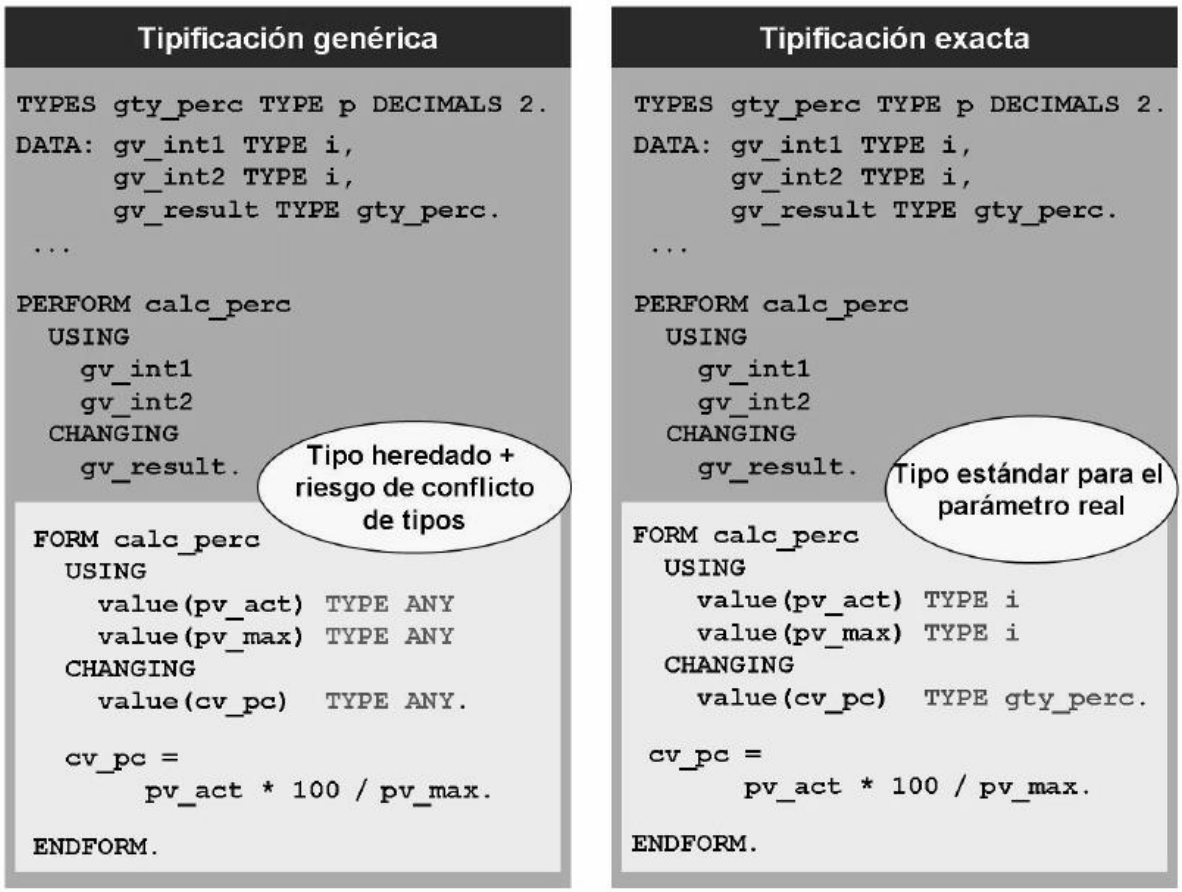
Al llamar la subrutina, los parámetros reales que se transferirán sin el prefijo VALUE están especificados en USING o CHANGING. El orden de especificación determina su asignación a los parámetros formales. En el ejemplo del gráfico anterior, ‘**a’** se transfiere a **f1**, **b** a **f2** y **c** a **f3**.

Un parámetro formal se tipifica generalmente si se hace mediante TYPE ANY, o no se tipifica en absoluto. Los parámetros reales de cualquier tipo se pueden transferir a un parámetro de este tipo. En tiempo de ejecución, se determina el tipo de parámetro real y se asigna al parámetro formal (herencia de tipo) cuando se llama la subrutina. Sin embargo, si las sentencias de la subrutina no son adecuadas para el tipo heredado, puede producirse un error de tiempo de ejecución (conflicto de tipo). Por lo tanto, sólo se debe utilizar la tipificación genérica si el tipo del parámetro real aún no está determinado al crear el programa o si puede variar durante el tiempo de ejecución (programación dinámica).

Para implementar la tipificación concreta de un parámetro formal se especifica un tipo global o integrado en el suplemento TYPE. Al hacerlo, se determina que sólo se puedan transferir a la subrutina los parámetros reales del tipo especificado. La verificación de sintaxis detecta cualquier violación de la consistencia de tipo entre los parámetros formales y reales. Esto incrementa la estabilidad del programa, ya que previene los conflictos de tipo en sentencias dentro de la subrutina.

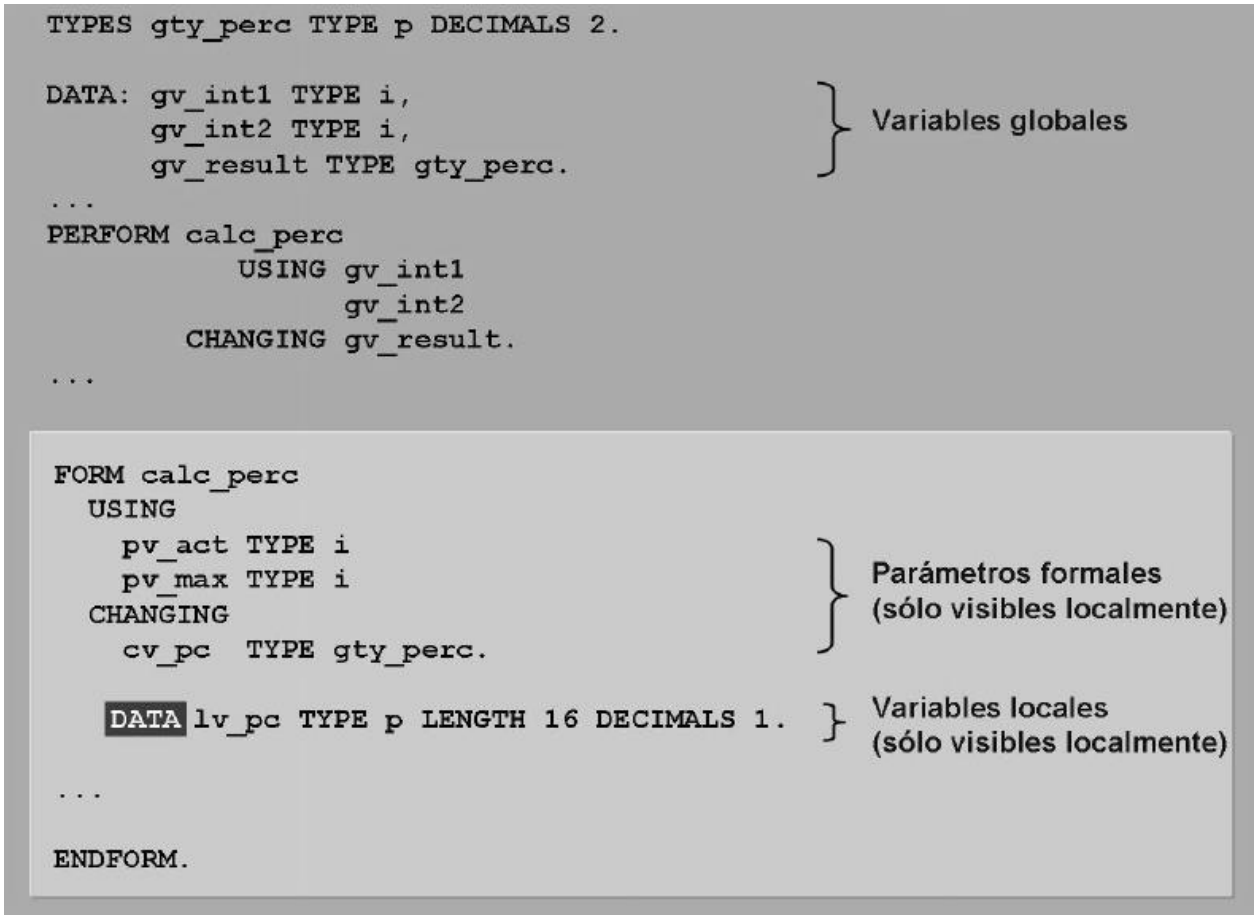
7. Tipificación de los parámetros de interfaz

Un parámetro formal se tipifica generalmente si se hace mediante TYPE ANY, o no se tipifica en absoluto. Los parámetros reales de cualquier tipo se pueden transferir a un parámetro de este tipo. En tiempo de ejecución, se determina el tipo de parámetro real y se asigna al parámetro formal (herencia de tipo) cuando se llama la subrutina. Sin embargo, si las sentencias de la subrutina no son adecuadas para el tipo heredado, puede producirse un error de tiempo de ejecución (conflicto de tipo). Por lo tanto, sólo se debe utilizar la tipificación genérica si el tipo del parámetro real aún no está determinado al crear el programa o si puede variar durante el tiempo de ejecución (programación dinámica).



Para implementar la tipificación concreta de un parámetro formal se especifica un tipo global o integrado en el suplemento TYPE. Al hacerlo, se determina que sólo se puedan transferir a la subrutina los parámetros reales del tipo especificado. La verificación de sintaxis detecta cualquier violación de la consistencia de tipo entre los parámetros formales y reales. Esto incrementa la estabilidad del programa, ya que previene los conflictos de tipo en sentencias dentro de la subrutina.

8. Objetos de locales y globales



Las variables definidas en el programa principal son objetos de datos globales. Son visibles (se puede acceder a ellas) en todo el programa principal y en cualquier subrutina llamada.

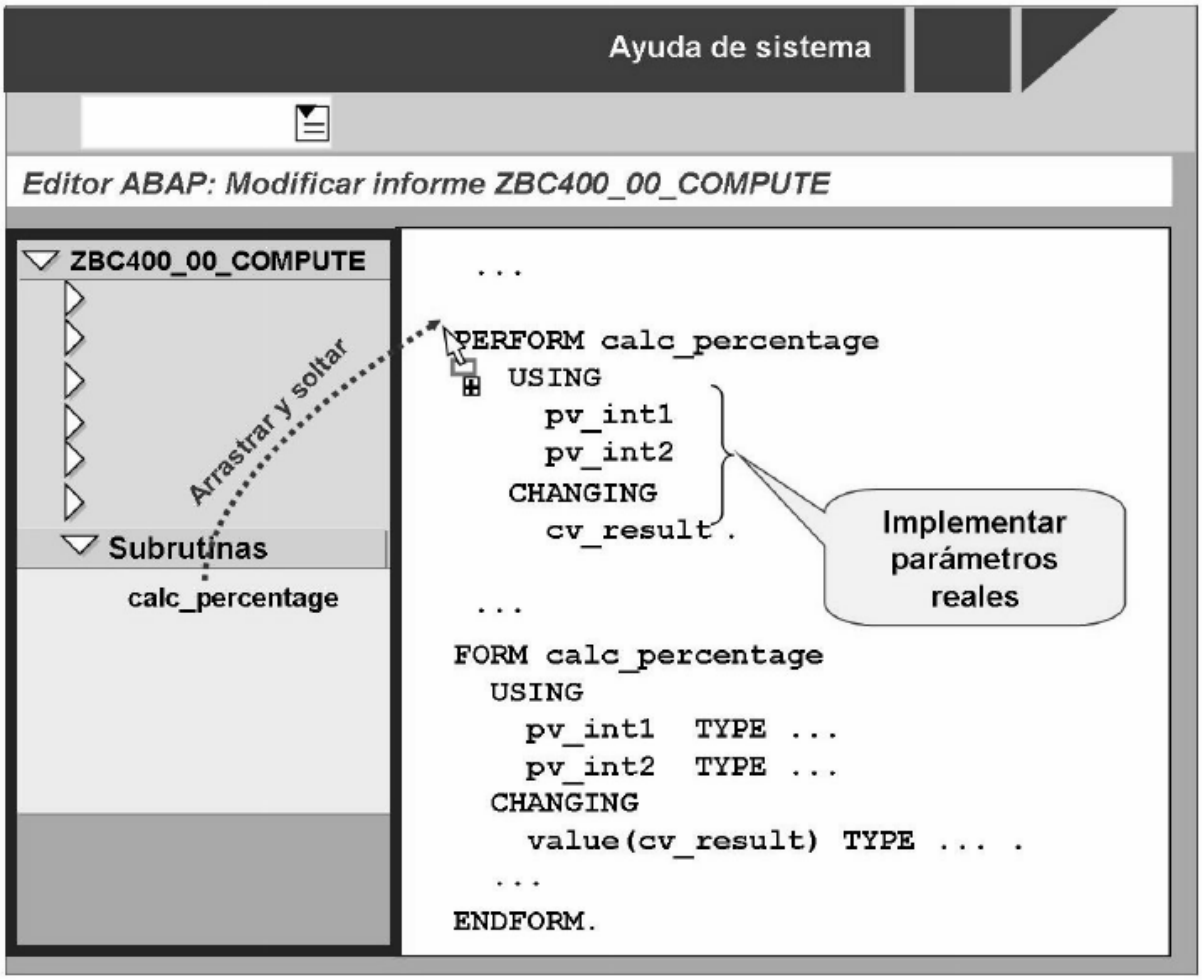
Las variables definidas dentro de una subrutina son locales porque sólo existen en la subrutina relevante, igual que los parámetros formales. El espacio de memoria de los parámetros formales y los objetos de datos locales sólo se asigna al llamar la subrutina y se libera de nuevo después de la ejecución.

Los parámetros formales y los objetos de datos locales de una subrutina no pueden tener el mismo nombre. Si existe un objeto de datos global con el mismo nombre que un parámetro formal o un objeto de datos local, el parámetro formal o el objeto de datos local se trata dentro de la subrutina, y el objeto de datos global se trata fuera de la subrutina. Este proceso se llama regla muestra: Dentro de la subrutina, el objeto de datos local “es la muestra” del global con el mismo nombre.

Para etiquetar claramente sus objetos de programa internos debe, por ejemplo, usar los siguientes prefijos para subrutinas:

- **g**…. para “objetos de datos globales”
- **p**.. . para “parámetros de utilización”
- **c**…. para “parámetros de modificación”
- **l**……para “objetos de datos locales”

9. Llamada de subrutinas

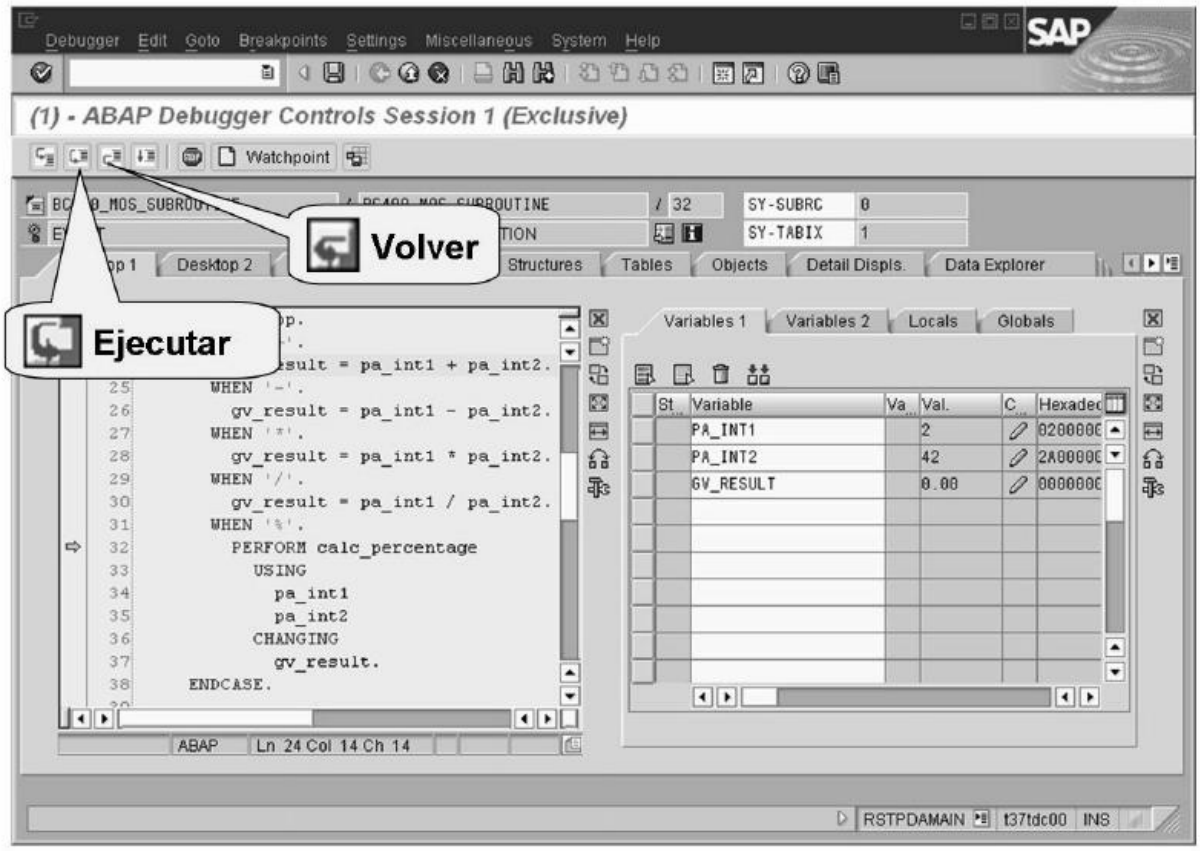


La sentencia **PERFORM** para llamar una subrutina puede generarse en su código fuente. Primero defina la subrutina y, a continuación, guarde el programa principal. La subrutina recién definida aparece en el área de navegación. Arrástrela y muévela hacia el punto de llamada necesario de su programa. Suéltela en este punto. Ahora, todo lo que debe hacer es sustituir los parámetros formales del código fuente generado por los

parámetros reales correspondientes. De forma alternativa, también puede implementar la generación de la llamada mediante el pulsador “ Patrón ” del Editor ABAP.

La ventaja de generar la llamada es que de este modo es imposible olvidar o mezclar parámetros.

10. Unidades de modularización en el debugger

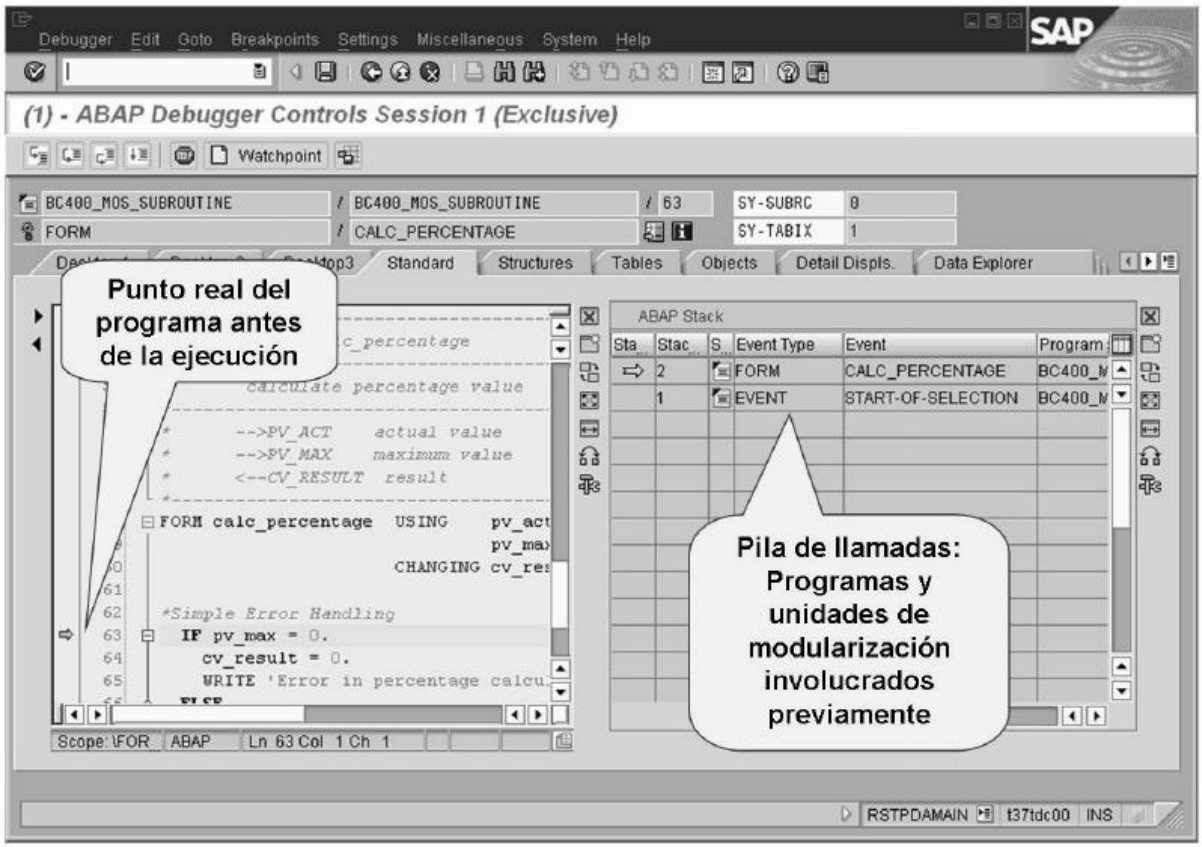


Si la sentencia actual es una llamada de subrutina, puede ejecutar toda la subrutina sin detenerse seleccionando **Ejecutar**. El procesamiento sólo se detiene una vez completada la subrutina.

Por otro lado, puede utilizar el **Paso individual** para detenerse en la primera sentencia de la subrutina y realizar un seguimiento más detallado de las operaciones.

Si la sentencia actual está ubicada en una subrutina, puede ejecutar el resto de la subrutina sin detenerse seleccionando **Volver**. El procesamiento sólo se detiene una vez completada la subrutina.

Todas las funciones de control del debugger (paso individual, ejecutar, volver y continuar) también están disponibles en el ABAP Debugger clásico con el mismo significado.



En la etiqueta **Estándar** del debugger puede visualizar desde qué programas se ha llamado la subrutina. La herramienta para ello es la **Pila de llamadas**.