Extending and Modifying the SAP Standard with Business Add-Ins and the New Modification Assistant

Karl Kessler

Karl Kessler studied Computer Science at the Technical University of Munich, Germany. He joined SAP AG in 1992 as a member of the basis modeling group. In 1994, he joined the product management group of the ABAP/4 Development Workbench. Since 1997, Karl has been product manager for SAP's application engineering tools.

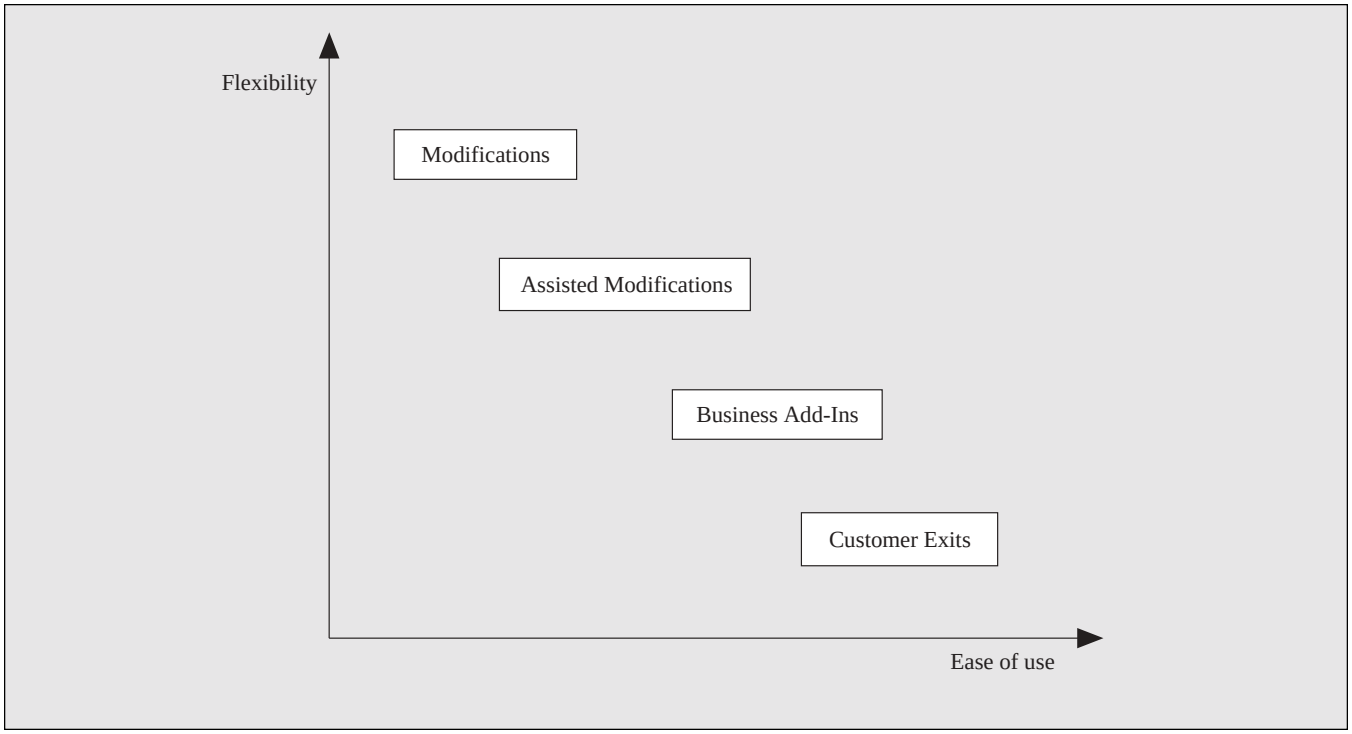
(complete bio appears on page 16)

BETWEEN CUSTOMER EXITS AND MODIFICATIONS lies a great development divide. Customer Exits provide predefined exit points from SAP source code, enabling you to insert your own code and locally *enhance* a standard application without having to dive into SAP application logic. To actually *extend* or *modify* a standard application requires the more powerful adaptation techniques found in the ABAP Workbench. Here, advanced programming knowledge is a must, as you are actually diving into the application's logic in order to make changes. As a rule, most developers only employ modifications when absolutely necessary, because the modified objects cannot be automatically reimported into your system after an upgrade. All modifications must be compared and adjusted manually.

Up until now, Customer Exits and Modifications were the only development techniques available to developers for extending and modifying standard SAP applications. There was nothing in between these two extremes. But that all changes with Release 4, where R/3 basis technology now offers two new techniques — Business Add-Ins (as of Release 4.6) and the Modification Assistant (Release 4.5):

- **Business Add-Ins** are predefined exit points in a source that allow developers to either insert their own logic during implementation or simply adopt one of the standard supplemental solutions provided by SAP. In contrast to Customer Exits, Business Add-Ins are not bound by a two-system infrastructure (SAP and customers), but instead allow for *multiple* levels of software development (by SAP, partners and customers, as country versions, industry solutions, and the like). In removing some of the restrictions inherent to Customer

Figure 1 Where Business Add-Ins and the Modification Assistant Fit In



Exits, Business Add-Ins allow different software producers to work independently in a veritable "software supply chain." Business Add-In logic can be created at each level within such a system infrastructure. Implementation of the add-ins themselves can also be undertaken at each level of development and their delivery to subsequent levels is now allowed.

- The **Modification Assistant**, as its name implies, makes it easier to modify the R/3 System. It works behind the scenes to register all the modifications you make to objects in the standard system in a separate layer in the ABAP Workbench. At the same time, Workbench tools have been redesigned to operate in "modification mode," which prevents you from inadvertently overwriting code, making an unwanted deletion, etc. You will find that when modifying the standard, the process is less error-prone and far easier to handle when you go to upgrade your system because modifications made using the Modification Assistant can generally be reimported during a release upgrade without manual intervention — i.e., reimporting of those modifications takes place automatically, in many cases.

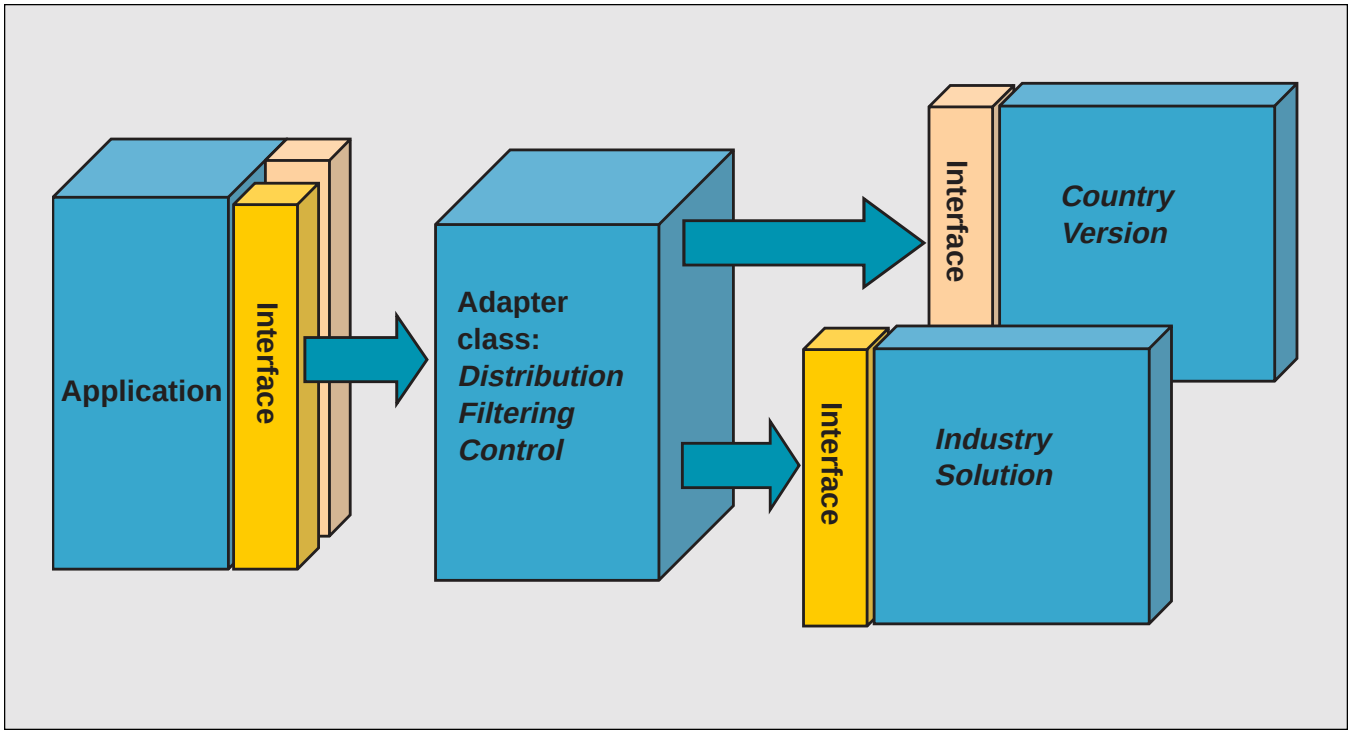
Figure 1 gives you a feel for where Business Add-Ins and the Modification Assistant fit within the spectrum of techniques that allow you to adapt the standard SAP R/3 applications.

In this article, I will introduce you to both Business Add-Ins and the Modification Assistant, with descriptions of how they will impact your current R/3 environment and do's and don'ts for using these powerful new modification techniques.

Business Add-Ins

Specific country versions and industries often require user requirements too specific to be included in the standard SAP delivery. Business Add-Ins accommodate the need to attach additional software to standard SAP source code.

Figure 2 Enhancing Programs with Business Add-Ins



Take, for example, the benefit calculation of the payroll in Human Resources. If employees participate in benefit plans, calculation procedures are performed that require tax calculations that differ from country to country. Consequently, country-specific processing needs to be factored out. Here, neither Customer Exits nor Modifications are the ideal development choice — Business Add-Ins are. 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). Customer Exits are not aware of the specific environment for a country version (nor can they support multiple implementations that operate in parallel in one installation), which means that for this HR scenario, you would have to rely on a modification of the SAP standard. When you do that, you compromise the idea of an upward compatible extension.

Let's take a look now at how you would actually enhance a program using a Business Add-In.

Enhancing Programs with Business Add-Ins

A single Business Add-In contains all of the interfaces necessary to implement a specific task implementation. 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.

An overview of the enhancement process is shown in **Figure 2**.

A single Business Add-In contains all of the interfaces necessary to implement a specific task implementation. 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.

In order to enhance a program, a Business Add-In must first be defined. Application developers create an interface for the add-in. Enhancement management takes this interface and generates an adapter class for implementing it, thus opening a path for implementations created by partners or customers. Your developer then creates an instance of the adapter class in the application program and calls the corresponding method at the appropriate time. The application interface definition ensures that consistent data is passed to the different add-in implementations. The adapter class, which is automatically generated by the system, takes care of calling and filtering out the proper components. (I will be providing details on all these activities.)

Customers can find the enhancements present in their system in the IMG and in the component hierarchy. To actually use a Business Add-In, customers must create their own implementation of that add-in, implement methods and user interface enhancements, and then activate their implementations of the enhancement. The enhancement's active components are then called at runtime.

Defining Business Add-Ins

In order for application developers to include Business Add-Ins in their programs, they must define an interface for the enhancement in transaction SE18 and call this interface at the appropriate point in their application program. Customers can then choose the add-in and implement it according to their needs.

In order for application developers to include Business Add-Ins in their programs, they must define an interface for the enhancement in transaction SE18 and call this interface at the appropriate point in their application program. Customers can then choose the add-in and implement it according to their needs.

Say, for example, you want to be able to convert strings in your application program. You also want users to determine how their strings are converted themselves. As the application developer, you define an enhancement consisting of an interface with a method. A changing parameter is used to transfer strings.

In order to create an add-in like this, you would proceed as follows:

1. Start the Add-In Manager (transaction SE18).

2. Name your Business Add-In.

3. Associate an ABAP Objects Interface with your add-in. You can change the name of the interface that is generated, if you like.

4. Add method definitions to the interface. The system will take you to the Class Builder.

5. Add parameters to your methods.

6. Create documentation for your add-in.

Whenever you assign a method to an interface, the corresponding executing class is generated. The code generated cannot be altered in the initial expansion phase.

Calling Add-Ins from Application Programs

During Business Add-In definition, enhancement management generates an adapter class that implements the interface. Application developers use factory methods to create instances of adapter classes during initialization. The instance methods are then called at the appropriate time.

The adapter class methods generated by add-in management decide if multiple active implementations should be called. If necessary, these implementations are subsequently executed. The application program itself simply calls the adapter class methods; it does not need to know which implementations are actually being called.

Here is an example of how to program a Business Add-In call into your ABAP source code. It builds upon the string conversion example:

```
Report businessaddin.
class cl_exithandler definition load.
   "declaration
data exit type ref to if_ex_businessaddin.
   "interface reference
data word(15) type c value 'Business Add-In'.
   "string you want to change

start-of-selection.
call method cl_exithandler=>get_instance
   "factory method call
changing instance = exit.
write:/'Please click here'.

at line-selection.
write:/ 'Original word: ',word.

call method exit->method "add-in call
changing parameter = word.

write:/ 'Changed word: ',word.
```

In order to be able to call static methods, you must declare the corresponding class in ABAP Objects. This is why the "class … definition load" (in this case, "class cl_exithandler definition load") statement is necessary for the factory class.

A variable for object reference is also necessary for the method call. Use the "data" statement to create it and type it to the interface.

During Business Add-In definition, enhancement management generates an adapter class that implements the interface. Application developers use factory methods to create instances of adapter classes during initialization. The instance methods are then called at the appropriate time. The adapter class methods generated by add-in management decide if multiple active implementations should be called. If necessary, these implementations are subsequently executed.

Implementing Business Add-Ins

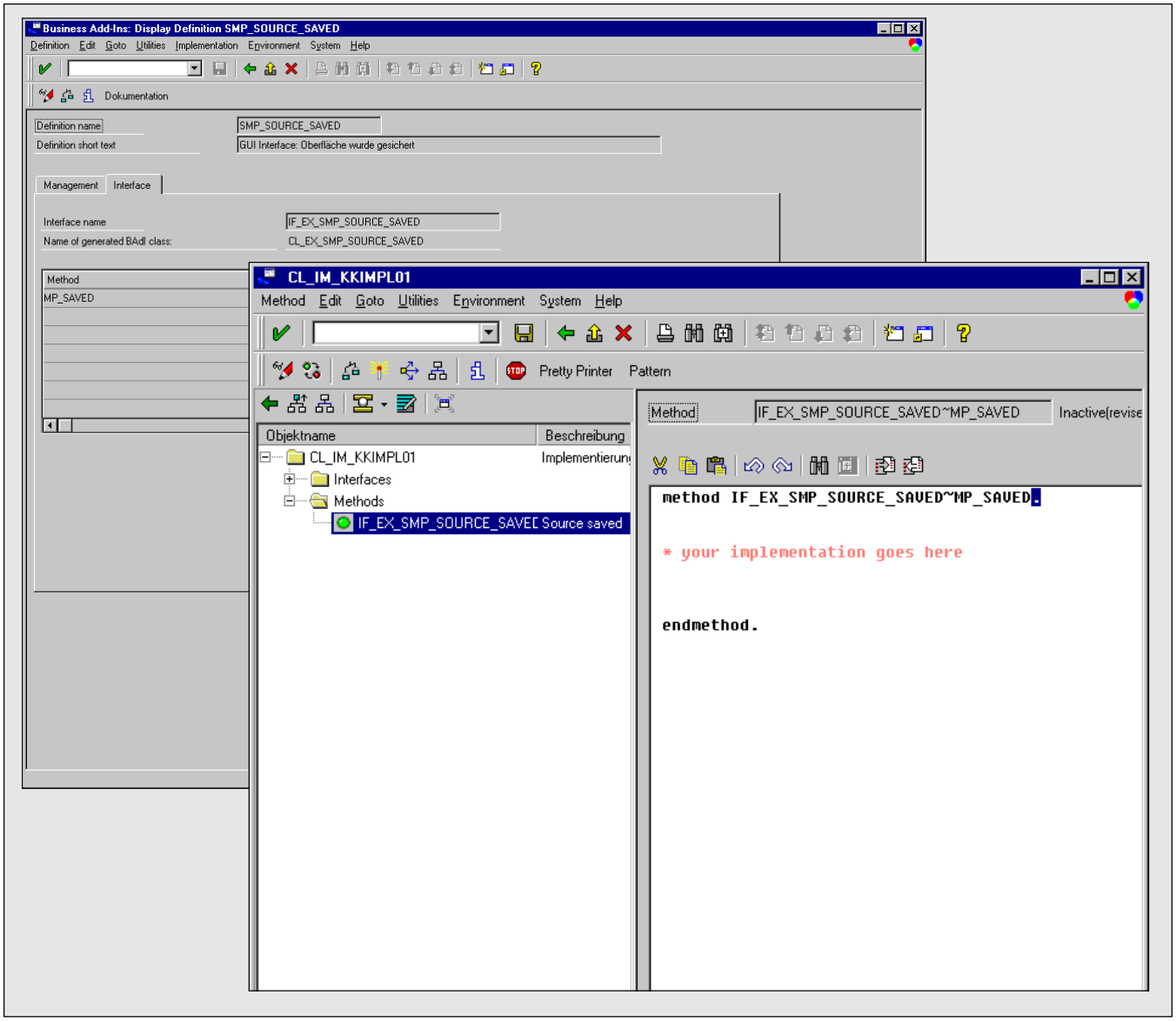
A list of the Business Add-Ins present in your system can be found either in the IMG or in the component hierarchy. The enhancements' names and corresponding documentation should help you decide the add-in for which you want to create an implementation. During implementation creation, a class for implementing the enhancement's interface is also created. Implementations are discrete transport objects and lie within the namespace of the person or organization implementing them.

In order to create an implementation for the string conversion example, the add-in (in this case, the interface's method) needs to be filled with logic that converts the string. This logic will be run through every time the add-in is called from the application program.

To create an implementation, proceed as follows:

1. Start the Add-In Manager or select the corresponding activity in the Implementation Guide.
2. Enter a name for the implementation.
3. Associate an add-in with your implementation.

Figure 3 Implementing Add-In Methods



4. Add code to the interface's methods (see **Figure 3**).

In order to implement a method for your add-in, insert the desired source code for the implementation between the following method statements, which are automatically provided to you by the system:

```
if_ex_businessaddin~methode.
...
endmethod.
```

Numerous implementations may exist for a Business Add-In that only supports a single implementation. However, only one implementation can be active for these kinds of Business Add-Ins at any one time.

Filter-Dependent Business Add-Ins

Business Add-Ins may be implemented depending on a specific filter value. If the standard allows for an

enhancement for different country versions, for example, it is likely that various partners will want to implement this enhancement. Distinct country-specific implementations can then be created and activated according to country.

Enter a filter type when defining your enhancement (a country or industry sector, for example). All methods created in the enhancement's interface have filter value FLT_VAL as their import parameter. The application program provides the filter value to the enhancement method. The method then selects the active implementation for that value.

Defining a Filter-Dependent Business Add-In

To define a filter-dependent Business Add-In, you must associate a filter type to your add-in.

Filter types are data elements and must fulfill the following criteria:

- The data element's domain may contain a maximum of 30 characters and must be of type Character.
- The data element must either have a search help with a search help parameter of the same type as the data element, and this parameter must serve as both the import and export parameter, or the element's domain must have fixed domain values or a value table containing a column with the same type as the data element.

If need be, you can create such data elements yourself.

Now create an interface with a method. Be aware that for each method you create in the interface of a filter-dependent enhancement, the appropriate filter value must be defined as the import parameter so that the application program can provide the filter value to the enhancement method. The method then selects the active implementation for that value. The filter value is declared using parameter FLT_VAL and is preset in the list of parameters.

Calling a Filter-Dependent Business Add-In from an Application Program

As previously discussed, application developers create an instance of the generated class in their application programs and call the corresponding method at the appropriate time. The filter value is passed to the method as an export parameter, as shown below:

```
Report businessaddin.
class cl_exithandler definition load.
data flt type usa_land.
data exit type ref to if_ex_businessaddin.
data word(15) type c value 'Business Add-In'.

start-of-selection.
perform formatlist.
call method cl_exithandler=>get_instance
changing instance = exit.
write:/ 'Please click here'.

at line-selection.
new-page.
write:/ 'Original word: ',word.

call method exit->method
exporting
flt_val = flt.
Changing
parameter = word.

write:/ 'Changed word: ',word.
```

The subroutine *formatlist* looks like this:

```
form formatlist.
write:/'USA ->Conversion to upper
case'.
flt = 'USA'.
hide flt.
write :/'Ireland ->Conversion to upper
case'.
flt = 'Ireland'
hide flt.
write :/'Italy ->Conversion to...'
flt = 'Italy'.
hide flt.
endform.
```

Implementing a Filter-Dependent Business Add-In

If you want to use a filter-dependent Business Add-In, you will need implementations for each filter value you intend to use. Multiple filter values may use the same implementation, however.

When implementing a filter-dependent Business Add-In, proceed as follows:

1. Create an implementation by referring to the corresponding Business Add-In definition.

2. Select a characteristic filter value for the implementation. In principle, it is possible to define multiple characteristic filter values for each implementation.

3. Use the Class Editor to fill the interface method.

4. In the string conversion example, you would make the following entries for each country:

   USA:
   translate parameter to upper case.
   Ireland:
   translate parameter to lower case.
   Italy:
   translate parameter using 'iItTaAlLyY'.

5. Activate your implementations.

Now, whenever you execute the report program described above, different country-specific implementations are executed.

Multiple-Use Business Add-Ins

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. This feature allows you to couple the SAP standard with additional solutions based on the publish/subscribe pattern for communication.

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. This feature allows you to couple the SAP standard with additional solutions based on the publish/subscribe pattern for communication.

You can differentiate between single- and multiple-use Business Add-Ins. Single-use add-ins are based on procedures, whereas multiple-use add-ins have characteristics similar to those of events. In the first case, the program waits for the enhancement to return something, usually a return code. Benefit calculation in HR is a good example of this type of enhancement. Here, different calculations can be performed according to whichever implementation is active. With multiple-use add-ins, an event is processed in program flow that may be of interest for other components. These components can then use this event as a hook to hang their own additional actions on; nothing is returned to the original program.

Say, for example, you want your application to continue processing indexes with a different component after you have saved (in other words, the system should allow you to use an add-in after saving). Since this is a good callup point for numerous different functions, you want to create an enhancement at this juncture that can be used by multiple subscribers.

The number of subscribers that subsequently call the event and hang their own additional actions on it is of no importance to the application program calling the add-in. Active implementations are called in the adapter method.

Coming Soon…

Normally, a Business Add-In contains an interface and other additional components such as function codes for menu enhancements. Starting with the next release, Business Add-Ins will also include enhancements for screens and tables.

Helpful Hints

✔ 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.

✔ Business Add-Ins are not a replacement for Customer Exits and there is no automatic conversion from Customer Exits to Business Add-Ins.

✔ Enhancements, interfaces, and generated classes all lie in the namespace for application development. Business Add-In implementations lie in the respective namespaces of the people who created them. So, if you plan to distribute your Business Add-In (both definition and implementation), you should consider working in your own namespace.

✔ The standard naming conventions for repository objects apply for Business Add-Ins. Start your implementation with Z or Y.

✔ Give careful thought to the kind of Business Add-In you want to design, because changing from one kind to another can have consequences. To change from a standard to a filter add-in, for instance, might invalidate your generated interfaces, thereby invalidating potential implementations.

✔ Be aware that Business Add-In Interfaces are based on ABAP Objects. Parameter passing defaults to reference. Internal tables do not have a header line.

✔ Calling a Business Add-In requires allocating an instance of your add-in class. This can cause performance problems if your add-in is heavily called. The benefit calculation mentioned earlier uses a table of references that are reused whenever a calculation is done.

✔ When working with multiple-use Business Add-Ins, do not use IMPORTING parameters, since you work in publish/subscribe mode with multiple event consumers.

✔ If you want to ask your Business Add-In implementation for something like status information and later inform it about some state change, I recommend you use standard add-ins that use methods with an export-only interface.

✔ Add-ins are a complementary technology and do not contrast with BAPIs. You can think of them as an outbound call, whereas a BAPI represents an inbound call in most cases.

✔ Design your add-in with a local scope in mind. If you add too many dependencies to your program, you will encounter modification-like problems.

Enhancements, interfaces, and generated classes all lie in the namespace for application development. Business Add-In implementations lie in the respective namespaces of the people who created them. So, if you plan to distribute your Business Add-In (both definition and implementation), you should consider working in your own namespace.

Figure 4 Integrating the Modification Assistant into the ABAP Workbench Tools

Tool	New, Integrated Functions	Description/Example
ABAP Editor	Overlay source code in any module Append and delete source code	Extend an SQL statement Add functionality, remove functionality
Screen Painter	Add elements (extra fields and controls) Modify field attributes Change the layout Modify the flow logic	Add custom fields or subscreen Change an input field into an output field Customize the screens Provide logic to handle additional fields
Menu Painter	Insert, replace, and delete menu entries, menus, pushbuttons	Add functionality
Function Builder	Add a customer function module to an SAP function group Add new parameters to a function module Modify the parameter types	Requested by many IS solutions Upward compatible extension Necessary for structural changes
Text Elements	Add new text and modify existing text	Add specific terminology
ABAP Dictionary	Change data element attributes	Change keywords and headings
Documentation	Expand or replace long text	Add customer documentation

The Modification Assistant

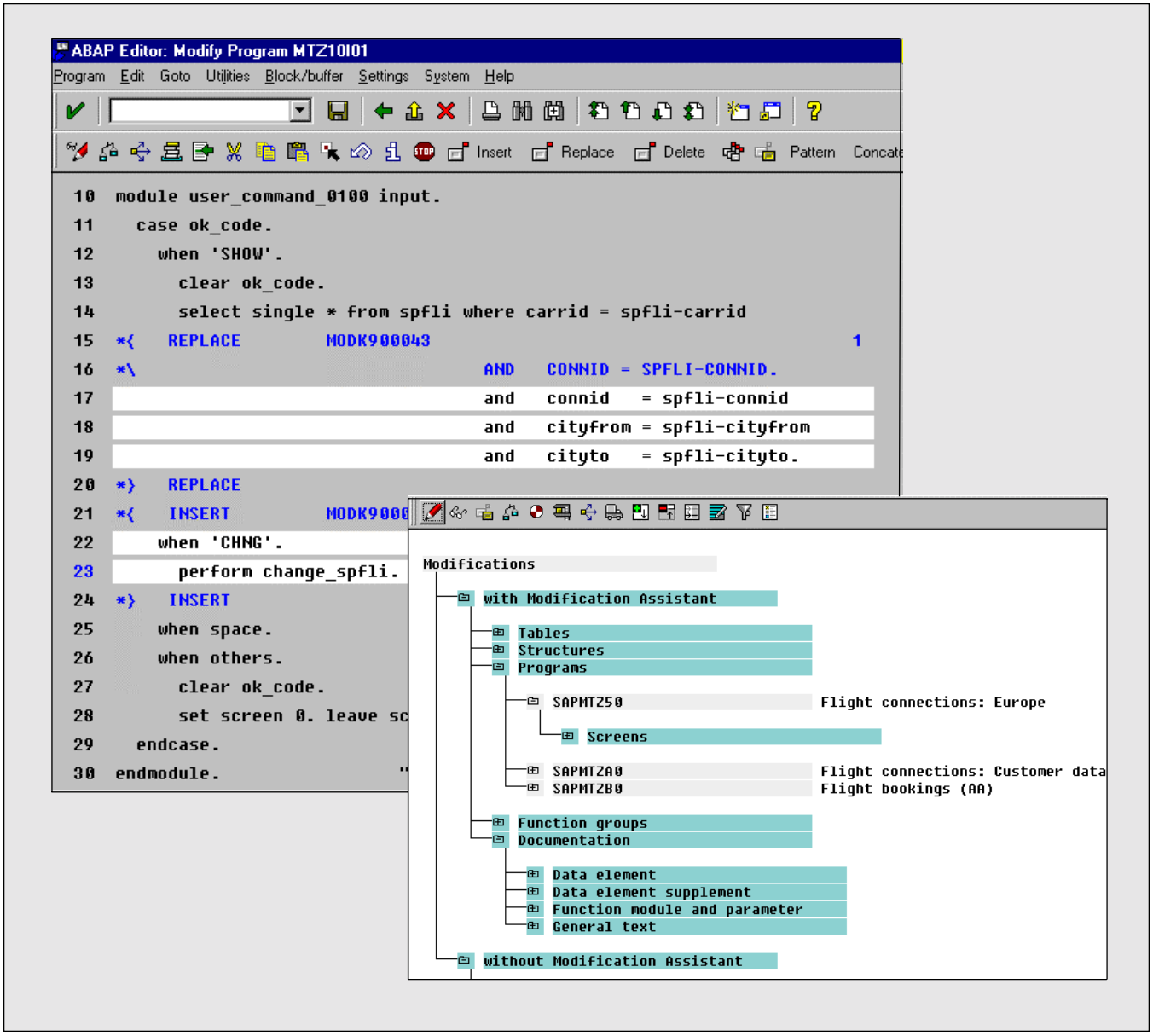
From Release 4.5, the Modification Assistant will make it *considerably* easier to modify the R/3 System. Its greatest benefits are the time and money saved during release upgrades. I predict that with the new Modification Assistant, traditional modifications will become nearly obsolete.

The Modification Assistant registers all modifications that you make to objects in the standard system, and administers them in a separate layer in the ABAP Workbench. When you modify an object, the Workbench tools operate in a new modification mode, which, by only offering a selection of appropriate functions, prevents you from making any unwanted changes by, for example, overwriting code. You cannot modify the object directly, and any modification can be undone at the push of a button, allowing you to restore the standard version easily at any time. Modifications made using the Modification Assistant can often be reimported automatically during a release upgrade.

Take a look at **Figure 4**, where you see a list of the new functions that have been added to the individual ABAP Workbench tools. In integrating the Modification Assistant into the ABAP Workbench, SAP placed great emphasis on ease of use and security.

From Release 4.5, the Modification Assistant will make it considerably easier to modify the R/3 System. Its greatest benefits are the time and money saved during release upgrades. I predict that with the new Modification Assistant, traditional modifications will become nearly obsolete.

Figure 5 Modification Overview, ABAP Editor



I hope the emphasis on ease of use becomes immediately apparent to you when you look at **Figure 5**, which presents the ABAP Editor in modification mode. A modification appears cleanly surrounded by comment braces indicating the type of the modification (insertion, replacement, deletion) together with the current date and correction number. As for the emphasis on security, this comment is added automatically and cannot be changed by the end user. The rest of the source is displayed in view-only mode. Note that modifications in the user GUI use different colors and fill patterns to indicate modified sub-objects.

In each of the individual ABAP Workbench tools, there is now a local overview function that allows you to display object-specific modifications as a hierarchical list. Additionally, the Modification Browser

(transaction SE95) provides a system-wide overview that you can call directly from the ABAP Workbench menu. You can use either of these tools as a starting point for modifying an object.

Easier Adoption of Modification Means Easier Upgrades

Because the Modification Assistant logs all of your modifications, the upgrade procedure is simplified. You adjust your modified objects using transaction SPAU. The objects are displayed in a hierarchical list similar to the Modification Browser. There are two possible ways of adopting a modification in the upgraded system:

- **Case 1 — Automatically.** You can reimport your modification with a single mouse-click. All of the modules that you have changed and that have not been re-delivered by SAP in the upgrade can be adopted straightaway. The fine granularity of the comparison — right down to modularization level — ensures that many conflicts can be avoided. The following is a list of circumstances in which modifications can usually be reimported automatically:
  - New modules created by the customer
  - New screen elements, as long as there is enough room on the screen
  - Changes to screen element attributes
  - Changes to the layout, as long as there are no positioning conflicts
  - Additions in the flow logic
  - Changes to menus and menu texts, as long as they do not conflict with the SAP version
- **Case 2 — Semi-automatically.** Here, the system identifies a conflict (for example, an object name used by SAP, a layout conflict in screens or menus, and so on) and advises you on how to solve the problem (consistent renaming). If the conflict arose in a source code modification, the splitscreen editor appears. You can then copy the modifications from your modified program version and paste them into the new standard.

The overview distinguishes between case 1 and case 2 using different color stoplights, as shown in **Figure 6**. Once a modification has been processed successfully, you can remove it from the overview.

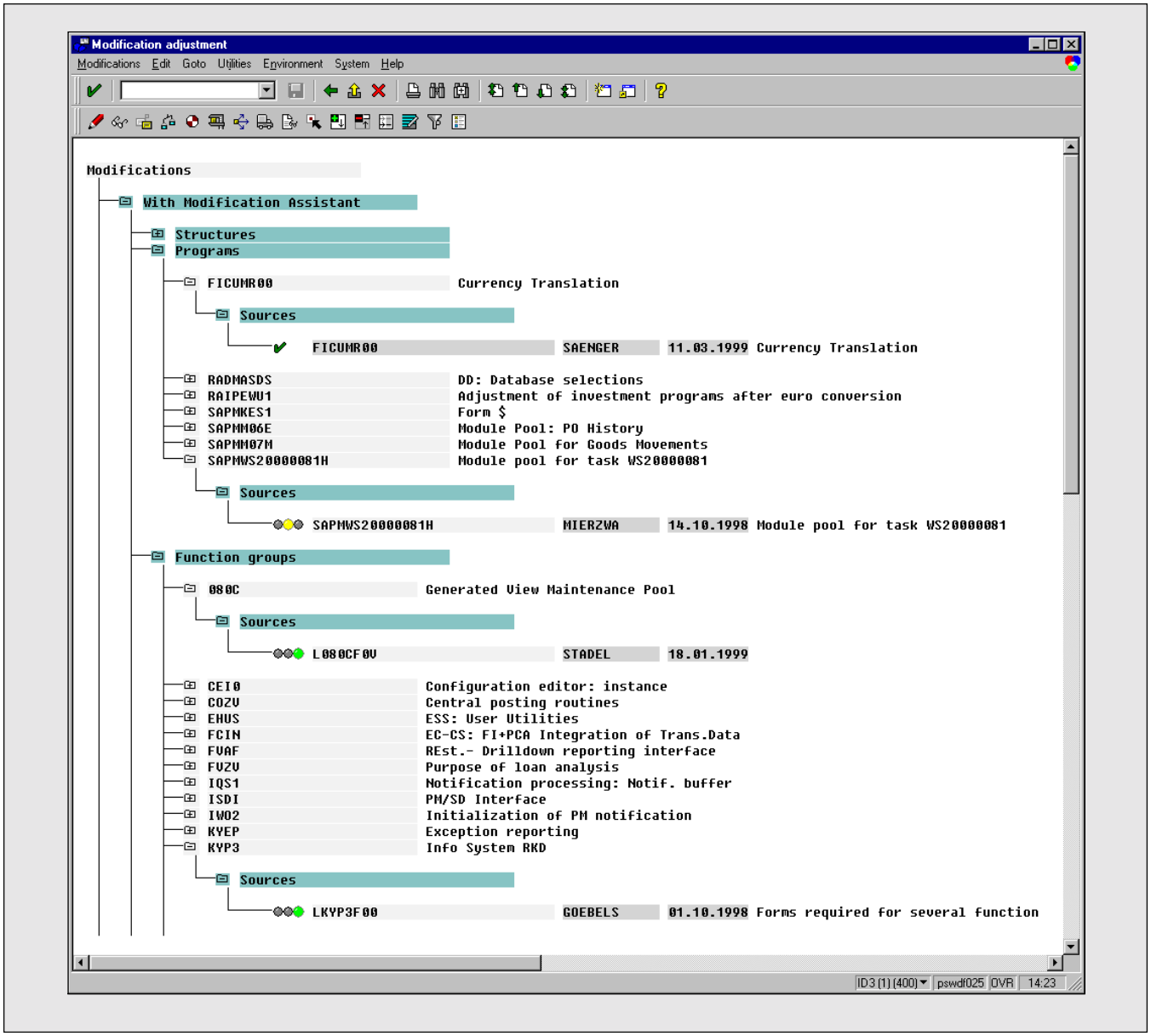
A typical modification example is the handling of custom fields. Initially you would go to the dictionary and create an APPEND structure for some SAP standard table. Next you would add those fields on the maintenance screen for that table. If you are able to manage the fields in a subscreen, you can better encapsulate your source code modifications, which results in less conflicts during upgrade. Once you have added your custom fields, you might add additional menus or pushbuttons to process custom functionality.

Because the Modification Assistant logs all of your modifications, the upgrade procedure is simplified. You adjust your modified objects using transaction SPAU. The objects are displayed in a hierarchical list similar to the Modification Browser. There are two possible ways of adopting a modification in the upgraded system: automatically and semi-automatically.

Helpful Hints

✔ For technical reasons, you can switch off the Modification Assistant for a development object. Thus you are losing any upgrade support other than displaying and comparing versions. The corresponding objects will nevertheless be present in the upgrade overview.

Figure 6 Adjusting Modifications After the Upgrade



✔ When using the Modification Assistant for the very first time, a good place to start is the maintenance of preliminary corrections, especially in the area of user interface elements where many tedious tasks can often be automated. Using the Modification Assistant for maintenance of preliminary corrections will save a lot of time when applying hot packages later on.

✔ The Modification Assistant is activated whenever you try to change a repository object that is not maintained as an original in your system. The corresponding tool will switch automatically into modification mode. As usual, you must open a Repair for Transport Purposes.

✔ Always keep in mind that a modification might affect the functional behavior of your application.

Therefore, never make a modification if you can implement the functional change by means of SAP standard customizing or the various personalization tools that SAP offers.

✔ Keep your modifications in a compact manner. Think of them as add-ins that were not originally defined by SAP. Fewer modifications means fewer conflicts during the upgrade.

✔ I would encourage IT managers to use the modification and upgrade overview functions to examine the modifications inside the development system.

✔ The modification protocol (log) is maintained in system tables of the ABAP Workbench. When doing a modification, the log is applied to the original and saved at the original's location. If you later transport the modified original to your production system, the modification log remains in your development system. That means that modifications are delivered via the source. The upgrade support is based on the modification log. Other system components like the runtime environment and the correction and transport system are not aware of the modification log.

Conclusion

SAP has a long tradition in offering technology to derive customized industry solutions out of the core R/3 System. Only a set of well-defined tools like the ones mentioned above can help with the task to adapt the standard to meet the customer's need. If you follow the extension and modification guidelines of SAP you will benefit from the new features of ABAP Workbench as of Release 4.

Karl Kessler studied Computer Science at the Technical University of Munich, Germany. He joined SAP AG in 1992 as a member of the basis modeling group, where he gained experience with SAP's basis technology. In 1994, he joined the product management group of the ABAP/4 Development Workbench. Since 1997, Karl has been product manager for SAP's application engineering tools. Karl can be reached at karl.kessler@sap-ag.de.

