Write Smarter ABAP Programs with Less Effort:

Manage Persistent Objects and Transactions with Object Services

Stefan Bresch, Christian Fecht, and Christian Stork

Stefan Bresch, Business Programming Languages Group, SAP AG

Christian Fecht, Business Programming Languages Group, SAP AG

Christian Stork, Business Programming Languages Group, SAP AG

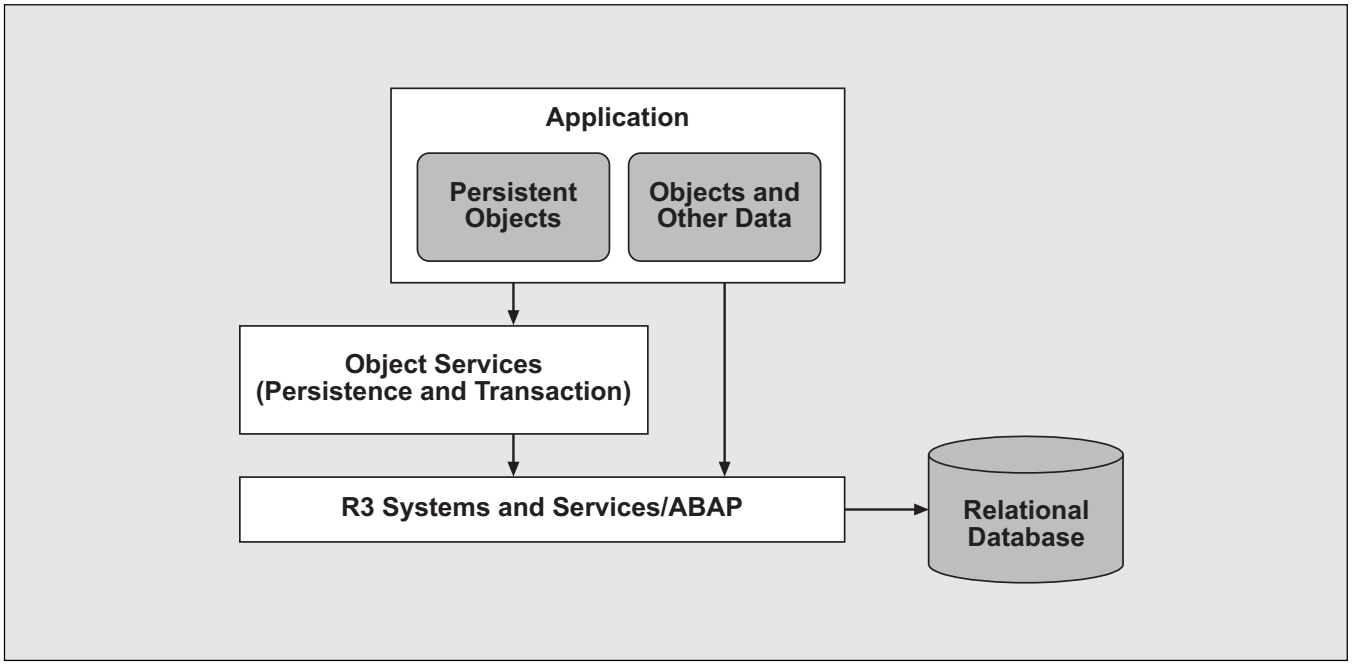
(complete bios appear on page 76)

ABAP developers, does this dilemma sound familiar? You're building an application with ABAP Objects. Parts of your application are modeled by objects and references between them. As with most business applications, you also need to store and manipulate persistent data. But since R/3 uses a relational database as its datastore, achieving object persistence has been difficult at best due to the gap between the object and relational models. In your application, you think in terms of objects that are related by references. However, the underlying relational database forces you to think in terms of tables and rows that may be related by foreign keys.

In the past, this gap between the object and relational worlds has seriously hampered the use of ABAP Objects in application development. Either you couldn't fully leverage proper object-oriented design principles, or you provided the infrastructure to close this gap yourself. In other words, you had to map classes to relational tables and write SQL code to load objects from and store them in the database. Instead of focusing on your application, you spent a considerable amount of time and effort designing and implementing a persistence framework.

With the emergence of Object Services in Release 6.10, the situation has changed substantially. The Object Services layer now provides a persistence framework that closes the object-relational gap. You no longer need to write SQL code, objects are transparently loaded from the database when needed, and changes to persistent objects are automatically tracked. Object Services offers a framework to handle most persistence functions automatically and transparently, freeing you to focus on the business logic in your application.

Figure 1 The Role of Object Services in the R/3 System Architecture



What Are Object Services?

Object-oriented *programming* is now well supported in ABAP. However, ABAP Objects alone don't provide everything you need for object-oriented *application development*. For example, ABAP Objects don't support a variety of system-level services that are common to most applications:

- Storing objects persistently in a database
- Binding an object to a UI representation
- Transaction and lock management, logging, archiving, and security

Clearly, repeatedly designing and implementing these services separately in different applications isn't an efficient approach. To address this drawback, the Object Services software layer was added. The Object Services runtime environment, which is itself implemented in ABAP Objects, sits between the application and the ABAP processor to provide these services. **Figure 1** shows how Object Services — of which there are two, Persistence and Transaction — fits into the overall SAP system architecture. In this article, we'll show you how to use the Persistence Service for storing objects in the database, and the Transaction Service for controlling transactions. Note that the Persistence and Transaction Services make their debut in Release 6.10.

How the Persistence Service Works

The role of the Persistence Service is to provide transparent object persistence. Previously, you had to do all the persistence work yourself: mapping objects to rows in the database, writing code to load and store objects, and keeping track of object changes. The Persistence Service now automates most of these tasks:

- You don't need to write any code for database manipulation. The Persistence Service knows how your objects are mapped to the database, and how to load and store them.

- The Persistence Manager automatically tracks all object changes. At commit-time, it synchronizes your objects with the database. It knows which entries must be inserted, updated, and deleted.

Note that the Persistence Service can only manage instances of persistent classes. Thus you need to define persistent classes in the Class Builder, which we'll describe later in this article. Persistent classes can have transient and persistent instances. Transient instances only exist in application server memory and are not saved to the database. Their lifetime is limited by the lifetime of the internal session in which they were created. A persistent instance is associated with a persistent object in the database. Both transient and persistent instances are in-memory objects that exist for the duration of an internal session, while the actual persistent object is stored in the database. Think of a persistent instance and its associated persistent object as two representations of the same logical entity. If the state of the persistent instance is updated, the persistent object in the database must also be updated. For efficiency reasons, this update is deferred to commit-time. Thus the persistent object and the persistent instance can be inconsistent at certain times.

> ✓ ***Exclusive Update Rights***
>
> *In order to guarantee exclusive update rights, you must use the R/3 enqueue mechanism to lock an object.*

All persistent instances are cached in memory during a session. The cache is then synchronized with the database at commit-time. You always operate on persistent instances in the cache. Before you can access an attribute of a persistent object or invoke one of its methods, the persistent object must be loaded from the database into the cache. The Persistence Service automatically performs this load. Objects are always loaded on demand, one by one. Suppose persistent Object A contains a reference to persistent Object B. If you address Object A by a query, only Object A is loaded into the cache. Object B is not loaded until it is accessed. The Persistence Service also supports transparent navigation. For example, if you navigate through an object graph, following references from one object to another, the objects are transparently loaded as needed.

Every persistent object has a unique identity that distinguishes it from all other persistent objects. The Persistence Service supports two types of object identifiers:

- **Business key**, which you assign in your application based on a subset of its persistent object attributes (business keys, key attributes, etc.).

- **GUID** (global unique identifier), which the Persistence Service automatically generates when the persistent object is created. A GUID is not part of the state of the persistent object, nor is it stored in any persistent attribute. You can't control GUID generation in your application.

In addition, a persistent object is uniquely represented by a persistent instance in the internal session. Thus if you address a specific persistent object several times within the same session, you always get the reference to the same persistent instance. In order to implement this uniqueness of representation, the Persistence Service maintains a mapping from identities to persistent instances. There is only one mapping per internal session.

Using Object Relational Mapping with Classes and Tables

In the world of object-oriented programming, there is no meaning to persistence in the relational sense. To bridge this gap, you need to establish a connection between classes in your ABAP code and tables in the R/3 database. Object instances of a class correspond one-to-one to table entries. Therefore, you map the persistent attributes of a class to a subset of fields in the table. This action is referred to as "object relational mapping." Classes have two types of

Figure 2 Mapping Classes to Tables

Method	When to Use It
Map one class to one table	Wherever possible. This scenario is the most common, as well as the most efficient.
Map one class to several tables (multi-table mapping)	When data belonging to a class is spread across more than one table. Key fields must be the same for all affected tables.
Map two or more classes to one table	For implementing inheritance. See the section "Advanced Considerations for Object Relational Mapping" for more information.

attributes: persistent and transient. You only need to define an object relational mapping for persistent attributes. Follow these additional guidelines to ensure proper support for object identifiers:

- If you are using business keys to identify objects, you must map the table key fields to persistent attributes. The object identity is derived from these key fields. These attributes, which are called business keys, are read-only because the object identity must not change.
- If you are using GUIDs to identify objects, the table must contain a key field of data type `OS_GUID` and you must map it to a special attribute named `OS_GUID`. This `OS_GUID` attribute is *not* part of the class. It is only used internally for object identity in the cache and the database.

You can perform object relational mapping in several ways, as shown in **Figure 2**.

Creating a Persistent Class

Now that you understand how persistent classes work, let's examine how to create them. You can create persistent classes in two ways:

- Start with the object model — Implement the classes first, then generate the tables needed.
- Start with the tables — Use the tables to create the object model.

The first method, generating tables from an object model, is not yet supported by Object Services. Object Services supports only the second method with the Persistence Representation tool in the Class Builder (transaction SE24).

In our experience, real-world examples provide the best context for learning a new technique, so let's walk through an example application that handles airline seat reservations. The example starts in this section by showing you how to create two persistent classes, one to represent the reservation, and another to represent the flight, with the necessary attributes and object relational mapping to link the classes to a table. In the sections that follow, you'll learn how to write the code for these classes to check space availability and reserve the seat. In all cases, we'll assume that your database and the required tables already exist.

To begin our example, we're going to create a persistent class named `CL_FLIGHT` to represent the airline flight:

Figure 3 Creating a Persistent Class in the Class Builder

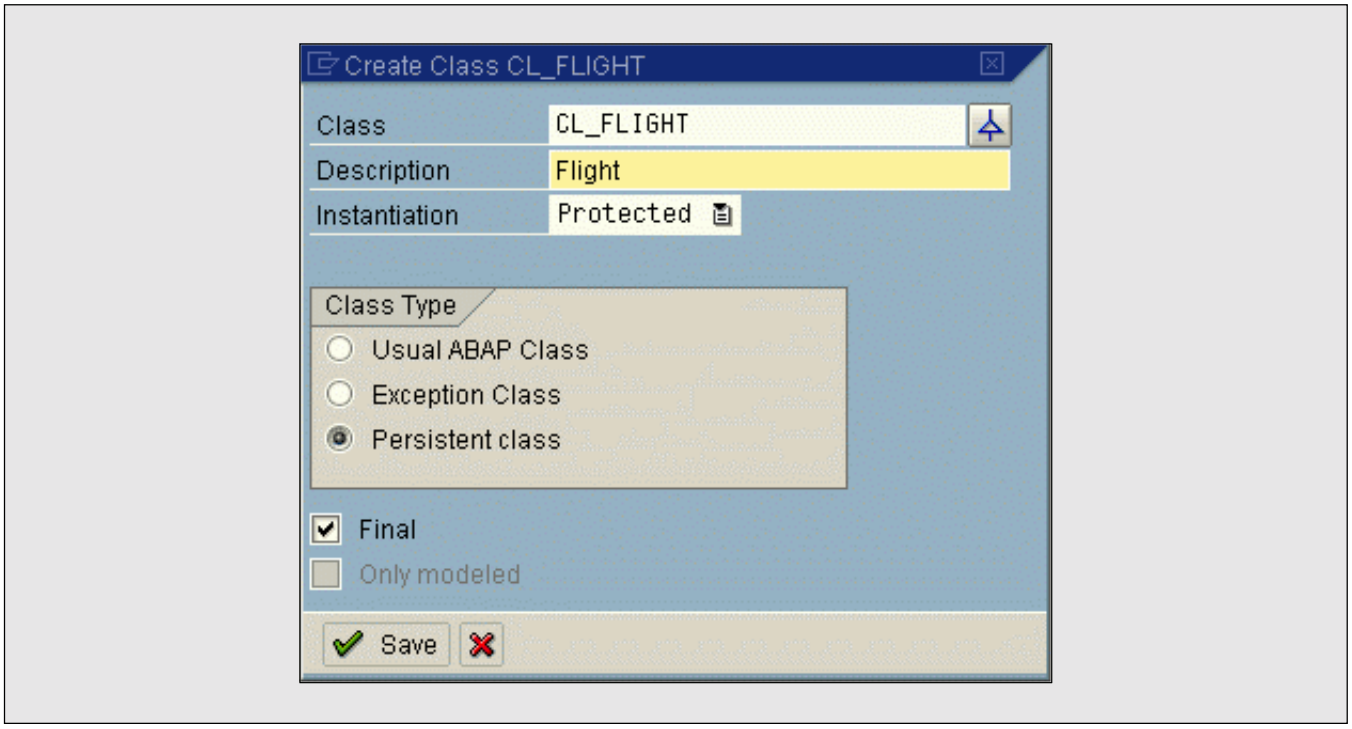
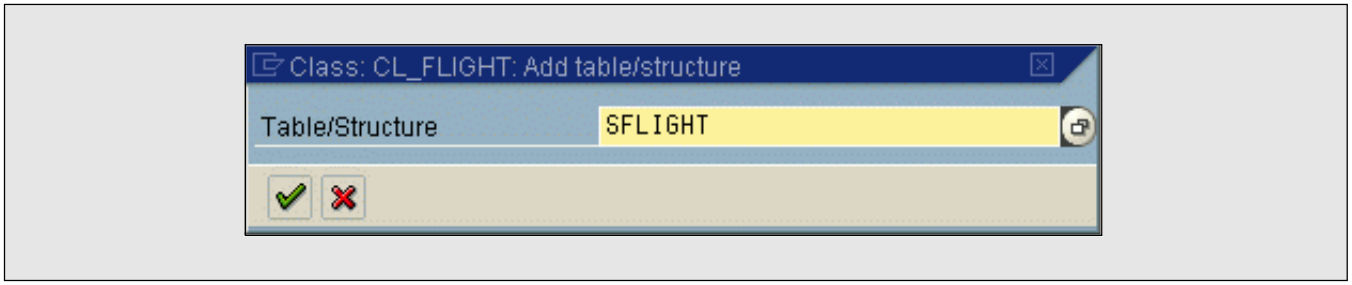


Figure 4 Mapping the Persistent Class to a Table



1. To create a class, open the Class Builder (transaction SE24) and click on "Create." This launches the screen shown in **Figure 3**.

2. Select "Persistent class," as shown in Figure 3. (Beware: you can't change this property later!) Note also that "Instantiation" switches to "Protected" after you select "Persistent class." Instantiation of a persistent class can only be "Protected" or "Abstract"; other values are not allowed.

3. Click "Save," which returns you to the Class Builder main screen.

4. In the Class Builder main screen, click Persistence, which launches the screen shown in **Figure 4**. Select the database table to which you want to map the persistent class — here, we select "SFLIGHT."

5. Selecting "SFLIGHT" takes you back to the Persistence Representation screen shown in

Figure 5 Using the Persistence Representation Tool

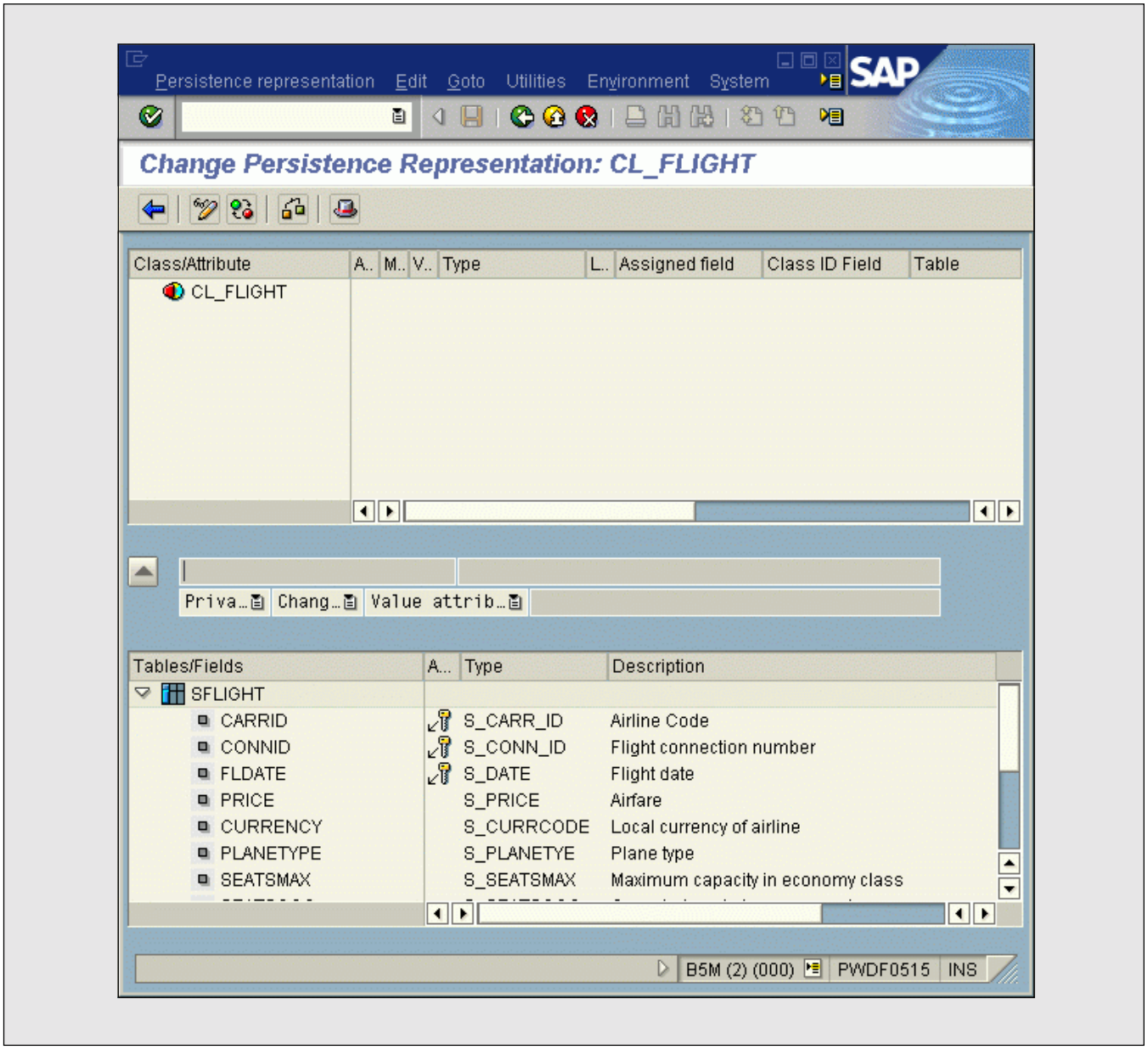


Figure 5, where you define persistent attributes for the class `CL_FLIGHT` and their table field mapping.

6. Double-click on a table field to fill the attribute editing area with values derived from the table field. We will create persistent attributes for the table fields `CARRID`, `CONNID`, `FLDATE`, `SEATSMAX`, and `SEATSOCC` with the same name. By default, the attribute's name is identical to the field name, but you can change the defaults as you wish.

   Depending on the field properties, for each attribute you can choose visibility (Public, Protected, Private), access (Read only, Changeable), and mapping type (Value attribute, Business key, Object reference, Class identifier, Type identifier,

Figure 6 *Setting Attribute Properties*

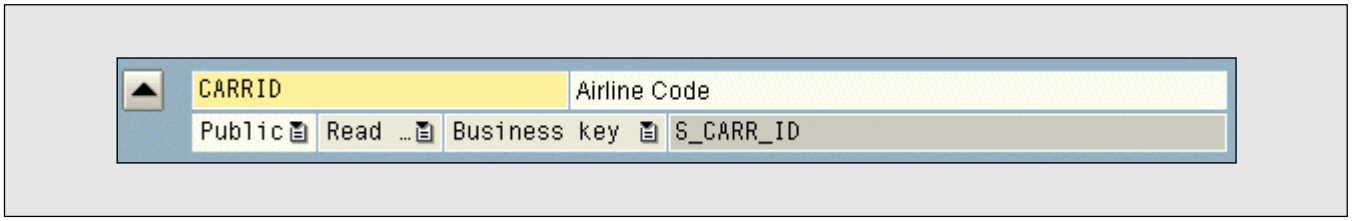
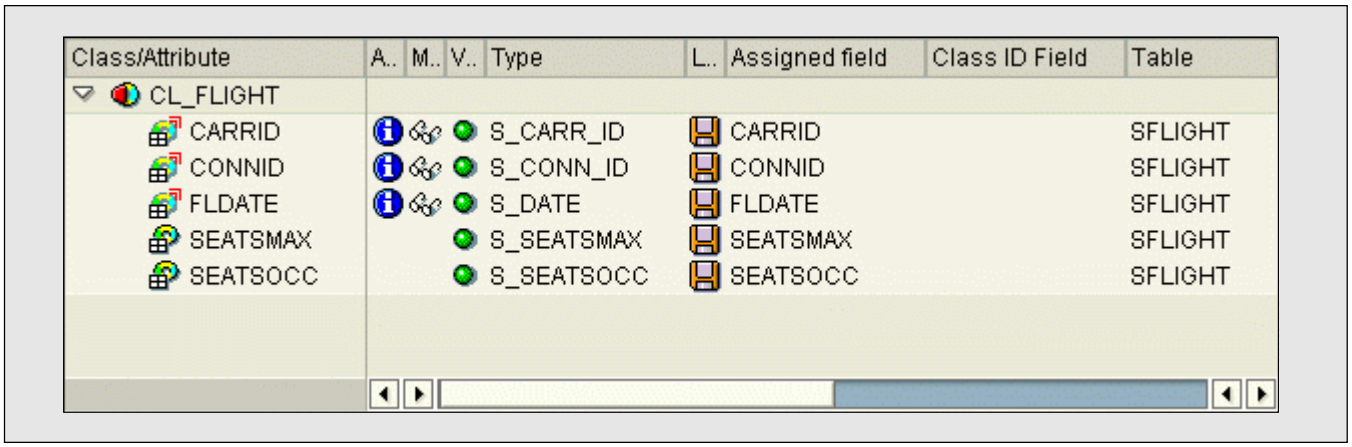


Figure 7 *Creating the CL_FLIGHT Persistent Class*



GUID). In our example, we used a mapping type of "Business key" for key table fields and "Value attribute" for non-key table fields. **Figure 6** shows what the screen looks like when you're setting attributes.

Accessor Methods

When you create a persistent or transient attribute, the Class Builder also generates accessor methods automatically. You use these methods to access the attributes of persistent classes. For the attribute SEATSOCC of CL_FLIGHT, the methods SET_SEATSOCC and GET_SEATSOCC are generated. Note that the visibility value you see in the Persistence Representation screen (see step 6) applies to the accessor method. The attribute itself is protected or private.

7. Press the "Return" key or click on to accept the values in the editing area. The upper half of the Persistence Representation screen now shows the new attribute with its mapping and properties.

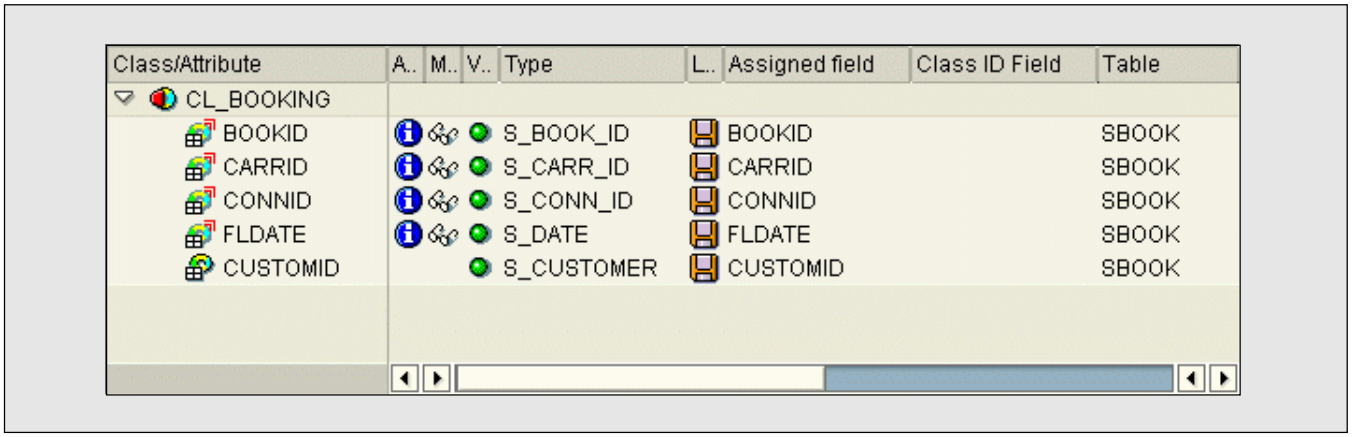
   Figure 7 shows how the upper half of the Persistence Representation screen looks after this step.

8. Click on to save your changes and on to return to the Class Builder.

9. Activate the persistent class in the Class Builder with . Also activate the class agent when prompted.

Repeat these steps to create the second persistent class, `CL_BOOKING`, to represent the reservation. Map the class to the table `SBOOK`, with the attributes `BOOKID`, `CARRID`, `CONNID`, `FLDATE`, and

Figure 8 Creating the CL_BOOKING Persistent Class



`CUSTOMID`. **Figure 8** shows how the upper half of the Persistence Representation screen looks after completing these steps for the second class.

> ✓ ***Mapping Multiple Tables to a Class***
>
> *To map more than one table to one class (or if there is no pop-up for selecting a table), use the context menu on the table container header.*

> ✓ ***Creating Transient Attributes***
>
> *To create a transient attribute, use the attribute tabstrip in the Class Builder.*

Before we begin discussing how to write the code to use these persistent classes, let's examine some special scenarios you might encounter with class-to-table mapping.

Advanced Considerations for Object Relational Mapping

As you have seen, object relational mapping is a seemingly simple concept and the Class Builder makes it easy to implement. However, be aware of the following complexities that could cause problems as you begin working with persistent classes:

- Managing object references
- Handling class inheritance

If you are not an experienced object-oriented programmer, you may want to learn more about the related design principles before working with these scenarios.

Managing Object References

Objects normally hold references to other objects. With persistent classes, persistent attributes in persistent instances can hold references to other persistent instances. But storing these references in the database presents a problem. These in-memory references point to objects in an internal session and are only valid for the duration of that session. In order to store references in the database, you need to do two things:

- Use a GUID (rather than a business key) as the identifier for the referenced object.
- The table corresponding to the reference holder's persistent class must contain two fields of type `OS_GUID` for storing the persistence reference: one for the instance GUID and one for the class GUID. The Object Services runtime system transparently handles the transformation between runtime and persistent references.

Figure 9 Mapping Inherited Classes to Tables

Inheritance Type	Mapping Method	Guidelines
Classes that inherit from a non-abstract class	Vertical mapping	Map the new attributes of a subclass, together with the attributes for the object identity, to its own table. Note that the object identity attributes are mapped to more than one table.
A class that inherits from an abstract class	Horizontal mapping	Since you can define some persistent attributes for an abstract class, define the subclass mapping to its own table for each subclass.
Subclasses to be mapped to the same table as their superclass	Mapping to one table with a type field	Each table entry represents a persistent instance of a class, which means you also need to know the persistent class to which the table entry belongs. To accomplish this, the table must contain a field with the data element OS_GUID, which contains the class GUID of the table entry, the "discriminator." Map this table field to a special attribute with the mapping type "Type identifier."

To define a mapping for an attribute that is an object reference, you must map two table fields to the attribute. Choose a name for this attribute and map the two table fields to the same name, one with mapping type "Object reference" and the other with mapping type "Class identifier."

> ✓ ***Storing GUIDs in Tables***
>
> *Use the data element OS_GUID for all table fields that contain a GUID. Otherwise, you can't choose mapping type "GUID," "Object reference," "Class identifier," or "Type identifier" in the Mapping Assistant's mapping type selector.*

Handling Class Inheritance

Mapping classes to tables becomes considerably more complex in the context of persistent class inheritance. Suppose you have a persistent Class A, and you want to define a persistent Class B that inherits from it. How do you handle the mapping? First, understand that using Object Services with mapping and inheritance is based on three principles:

- Mapping is inherited.
- Mapping cannot be redefined.
- A superclass of a persistent class must be persistent, except for the class `OBJECT`.

Consequently, the object identity of the root persistent class must be the same in all inherited classes. You can define a class-to-table mapping for inheritance in three ways, as shown in **Figure 9**.

Working with Persistent Classes

When working with persistent classes, you need to remember that class agents manage instances of persistent classes. These class agents provide lifecycle methods (`create-`, `get-`, and `delete-`) for managing the instance lifecycle in your program, but the agents perform the actual work. Only the

Figure 10 Persistent Class States

State	Description
NEW	Instances that are newly created as persistent with a create-persistent method.
LOADED	Instances that are loaded from the database with a get-persistent method. Note: If a persistent attribute of a loaded persistent instance is actually a reference to another persistent instance, an empty instance is created as NOT LOADED. When an attribute of the referenced instance is accessed, the instance is loaded from the database and its state becomes LOADED.
CHANGED	When an attribute of a loaded persistent instance is changed.
DELETED	If a delete-persistent method is called for this instance.
NOT LOADED	Instances after transaction completion, which means they can be removed by garbage collection.

class agent can create objects. Further, the class agent must be notified when an attribute of the persistent class is accessed. To accommodate these requirements, you can only access persistent and transient attributes with the provided accessor methods. The accessor method signals to the class agent that an attribute has been accessed.

Accessing Class Agents

For each persistent class, the Class Builder automatically generates two classes:

- Base Class Agent
- Class Agent

In our example application, the Class Agent class for the persistent class `CL_FLIGHT` is named `CA_FLIGHT`; the Base Class Agent class is named `CB_FLIGHT`. The Class Agent class, which is the only subclass of the abstract class Base Class Agent, follows the Singleton pattern. Although the instance is created implicitly, you can access it through the public static attribute `CA_FLIGHT=>AGENT`.

A superclass of all class agents provides methods that are common to all class agents and do not need to be generated. Method generation applies only to the Base Class Agent class. Therefore, you can redefine both the generated and common methods in the Class Agent class if you want to change the class agent's behavior (for example, if want to implement your own database table buffer).

Instance Lifecycle

An instance of a persistent class can be either persistent or transient. It also has an associated state at each point in its lifecycle, from creation to removal by garbage collection. **Figure 10** describes the possible lifecycle states of persistent classes.

If a persistent instance is created or loaded from the database, the callback method `IF_OS_STATE~INIT` of the persistent class is called. If a persistent instance is deleted or changes to `NOT LOADED`, the callback method `IF_OS_STATE~INVALIDATE` is called. Within these methods, your application can handle resources needed or initialize transient attributes.

Note that transient instances always have the state

Figure 11 Lifecycle Management Methods for Persistent Instances with Business Key Identity

Category	Methods Provided
Class-specific methods that pass the business key as a single parameter	CREATE_PERSISTENT(<BKEY1..BKEYN>)
	GET_PERSISTENT(<BKEY1..BKEYN>)
	DELETE_PERSISTENT(<BKEY1..BKEYN>)
Generic methods that pass a business key	IF_OS_FACTORY~CREATE_PERSISTENT_BY_KEY(<BKEY>)
	IF_OS_CA_PERSISTENCY~GET_PERSISTENT_BY_KEY(<BKEY>)
Generic methods that pass an object reference	IF_OS_FACTORY~DELETE_PERSISTENT(<OREF>)
	IF_OS_FACTORY~REFRESH_PERSISTENT(<OREF>)
	IF_OS_FACTORY~RELEASE(<OREF>)

TRANSIENT. You can't change an instance from transient to persistent, or vice versa. You must create an instance as transient with a `create-transient` method. If a transient instance is created, the callback method `IF_OS_STATE~INIT` of the persistent class is called.

Lifecycle Management

Lifecycle management methods for persistent instances are either class-specific or generic, and differ in how they pass the object identity as a parameter. **Figure 11** summarizes these methods for the business key object identity. For class-specific methods, the signature depends on the object identity type. If the object identity type is business key, you pass a parameter for each business key. For generic methods that pass a business key, the signature provides an untyped parameter. The class agent expects to receive a structure that contains all business keys in the defined order. In addition, some generic methods pass the instance as an object reference.

You can use lifecycle management events that are raised by the class agent to learn about changes to the lifecycle state of the instances of persistent classes. These events are raised when instances are created (`CREATED_PERSISTENT` and `CREATED_TRANSIENT`), loaded (`LOADED_WITH_STATE` and `LOADED_WITHOUT_STATE`), about to be deleted (`TO_BE_DELETED`), or have been deleted (`DELETED`). All lifecycle management events belong to the interface `IF_OS_FACTORY`. You can use these events to keep track of all objects that are created or deleted while your application is running.

Creating a Persistent Object

To create a persistent object, you use the class-specific method `CREATE_PERSISTENT`. The required parameters depend on the object identity type:

- For objects identified by business key, you must pass the business keys and (optionally) the value attributes and references. The persistent instance is first created only in the internal session. At commit-time, the persistent object is created in the database. For performance reasons, its existence is checked only in the cache, not in the database. To check the existence of an object, you must use the `GET_PERSISTENT` method manually. If an instance with the same identity already exists in the cache, the exception `CX_OS_OBJECT_EXISTING` is raised.

Listing 1: Creating a Persistent Instance of a Persistent Class

```
1  REPORT OS_EXAMPLE.
2  DATA: BOOKING       TYPE REF TO CL_BOOKING,
3        BOOKING_AGENT TYPE REF TO CA_BOOKING.
4  START-OF-SELECTION.
5  BOOKING_AGENT = CA_BOOKING=>AGENT.
6  TRY.
7    BOOKING = BOOKING_AGENT->CREATE_PERSISTENT(
8      I_CARRID   = 'LH'
9      I_CONNID   = '2104'
10     I_FLDATE   = '20010801'
11     I_BOOKID   = '00000042'
12     I_CUSTOMID = '00000013' ).
13   CATCH CX_OS_OBJECT_EXISTING.
14 ENDTRY.
15 COMMIT WORK.
```

- For objects identified by GUID, you don't need to pass any identity information. Value attributes and references are optional parameters. The GUID is created internally and assigned to the instance.

To continue our example, in the last section you learned how to create the persistent classes you would need for an airline seat reservation application. Now let's examine how to use those classes as you begin writing the code to reserve a seat. The first step is to create a persistent instance of the persistent class `CL_BOOKING`, which represents the reservation (see the sample code in **Listing 1**).

Here's how this sample code works. In line 5, the reference to the class agent (`CA_BOOKING=>AGENT`) is stored in the local variable `BOOKING_AGENT`. In lines 7-12, a persistent instance with the identity "LH 2104 20010801 00000042" is created in the internal session. Note that this example illustrates the use of the business key object identity type. The object is stored to the database when the `COMMIT WORK` statement is executed (line 15). If an object with this identity already exists in the database, there will be an update task failure. If an object with this identity already exists in the internal session, an exception is raised. (For simplicity, this example doesn't show the code for error handling.)

> ***Warning!***
>
> *If the class agent is not activated, your program will produce nasty syntax errors.*

Loading a Persistent Object

You use different methods to load a persistent object, depending on the object identity type:

- To load a persistent object with a business key object identity, you use the class-specific method `GET_PERSISTENT`. It expects the business keys as parameters. If the object is already loaded, the method returns a reference to the loaded object. If the object can't be found, the exception `CX_OS_OBJECT_NOT_FOUND` is raised.

- Persistent objects with a GUID identity are transparently loaded as you navigate from one object to another. Additionally, if you know the GUID, you can load the object manually with the generic method `IF_OS_CA_PERSISTENCY~GET_PERSISTENT_BY_OID`. It expects the GUID as a parameter.

You can load an already-loaded persistent object from the database with the method `IF_OS_FACTORY~REFRESH_PERSISTENT`. It expects the instance reference as a parameter.

Our example is not yet complete. In order to reserve a seat for a particular flight, you first need to determine whether a seat is still available. This information is stored in the persistent object that represents the flight (`CL_FLIGHT`). So, you need to load this object from the database. You also need to increase the number of occupied seats for the flight. And both changes must belong to the same transaction in order to guarantee data consistency. **Listing 2** shows the sample code to accomplish this task.

Let's review how this code works. In lines

Listing 2: Loading and Updating a Persistent Object

```
1  REPORT OS_EXAMPLE.
2  DATA: FLIGHT        TYPE REF TO CL_FLIGHT,
3        FLIGHT_AGENT  TYPE REF TO CA_FLIGHT,
4        SEATSFREE     TYPE I,
5        SEATSOCC      TYPE I.
6  DATA: BOOKING       TYPE REF TO CL_BOOKING,
7        BOOKING_AGENT TYPE REF TO CA_BOOKING.
8  START-OF-SELECTION.
9  FLIGHT_AGENT = CA_FLIGHT=>AGENT.
10 BOOKING_AGENT = CA_BOOKING=>AGENT.
11 TRY.
12   FLIGHT = FLIGHT_AGENT->GET_PERSISTENT(
13     I_CARRID   = 'LH'
14     I_CONNID   = '2104'
15     I_FLDATE   = '20010801' ).
16   SEATSFREE = FLIGHT->GET_SEATSMAX( ) -
17               FLIGHT->GET_SEATSOCC( ).
18   IF SEATSFREE > 0.
19   BOOKING = BOOKING_AGENT->CREATE_PERSISTENT(
20     I_CARRID   = 'LH'
21     I_CONNID   = '2104'
22     I_FLDATE   = '20010801'
23     I_BOOKID   = '00000043'
24     I_CUSTOMID = '00000013' ).
25   SEATSOCC = FLIGHT->GET_SEATSOCC( ) + 1.
26   FLIGHT->SET_SEATSOCC( SEATSOCC ).
27   ENDIF.
28   CATCH CX_OS_ERROR.
29 ENDTRY.
30 COMMIT WORK.
```

12-15, the flight with identity "LH 2104 20010801" is loaded from the database. In lines 16-17, the accessor methods are used to obtain the flight's attributes and compute the number of seats available. In lines 19-24, a booking is created if a seat is still available for this flight. In lines 25-26, the flight's attribute SEATSOCC is increased by 1. Here you can see the use of both accessor methods, GET_SEATSOCC and SET_SEATSOCC. For simplicity, we generically catch a superclass of all Object Services user exceptions. As in the previous code sample, we excluded the error-handling code. If the flight is not found, an exception is raised and caught in line 28, and no booking is created. We previously handled the case where a booking with this identity already exists (see Listing 1, which shows the code for creating a persistent instance). In both cases, transactional integrity is preserved. Note that you need to change the booking ID in line 23 to avoid duplicate key errors. Furthermore, make sure that the data for the flight exists in the database. You could also add code that retrieves the next booking ID from the corresponding number range.

> ✓ ***Handling Error Conditions***
>
> *Object Services offers a powerful, class-based exception framework. It provides built-in exceptions for almost any error situation, such as get-exception, create-exception, and delete-exception. This feature makes it easy for you to handle error conditions properly.*

Deleting a Persistent Object

Last but not least, let's not forget that you need to delete persistent objects. To delete a persistent object with a business key object identity from the database, use the class-specific method DELETE_PERSISTENT. It expects the business keys as parameters. First, the instance is marked as deleted in the cache. Its existence is checked first in the cache and then in the database. If the object does not exist, the exception CX_OS_OBJECT_NOT_EXISTING is raised. Then, at commit-time, the persistent object is deleted from the database. If you access a deleted instance with an accessor method, the exception CX_OS_OBJECT_NOT_FOUND is raised. You can delete an already-loaded persistent object from the database with the method IF_OS_FACTORY~DELETE_PERSISTENT. It expects the instance reference as a parameter.

In summary, you have now learned how to use the Persistence Service to transparently create, load, manage, and delete persistent objects. However, Object Services also provides a valuable Transaction Service. Let's examine how this component makes it similarly easy to manage transactions in your applications.

How the Transaction Service Works

The Transaction Service enables you to programmatically define transaction boundaries and subtransactions. Transactions are represented by transaction objects. Transaction objects are managed by the Transaction Manager, which follows the Singleton pattern and provides a factory method for creating transaction objects. The Transaction Manager implements the interface IF_OS_TRANSACTION_MANAGER, and the transaction class implements the interface IF_OS_TRANSACTION. You only have to deal with the interface IF_OS_TRANSACTION.

You can only use a transaction object for one transaction. Follow these guidelines for using the Transaction Service to manage a transaction:

- Use the method START to begin a transaction.
- Use the method END to successfully end a transaction.

- Use the method UNDO to roll back any changes since START.

Object Services also supports nested transactions. Within a transaction, you can start a sub-transaction. If a sub-transaction is rolled back, the state of all objects changed in this sub-transaction will be as it was when this sub-transaction started. Nested transactions are useful for isolating the transactional behavior of a called procedure from its caller. Using its own sub-transaction, the called procedure can perform a local rollback without invalidating the transaction logic of its caller. A transaction that is started in another transaction is automatically a sub-transaction, which means that parallel transactions are not supported. The first started, and now running, transaction is called the "top-level" transaction.

If you call the method END on the top-level transaction, an ABAP COMMIT WORK is executed. If you call the method UNDO on the top-level transaction, an ABAP ROLLBACK WORK is executed and the state of all changed objects will be as it was before the transaction started. The Object Services runtime copies the before image of the instance to the undo buffer only when the instance is changed. If a transaction is successfully ended, the undo buffer with the before images of the instances changed in this transaction is discarded. The undo buffer uses the methods IF_OS_STATE~GET and IF_OS_STATE~SET of the persistent class to get the before image of the state of an instance and to set the state of an instance from the before image. At the start of a subsequent transaction, loaded persistent instances will be invalidated and reloaded if accessed in the subsequent transaction. To avoid this, you can create chained transactions with the methods END_AND_CHAIN and UNDO_AND_CHAIN.

To wrap up our example, let's see how you might leverage the Transaction Service. One option is to use the Transaction Service instead of the COMMIT WORK statement. **Listing 3** shows the sample code to

Listing 3: Replacing a COMMIT Statement with the Transaction Service

```
1  REPORT OS_EXAMPLE.
...
7        BOOKING_AGENT TYPE REF TO CA_BOOKING.
T1 DATA: M TYPE REF TO IF_OS_TRANSACTION_MANAGER,
T2       T TYPE REF TO IF_OS_TRANSACTION.
T3 LOAD-OF-PROGRAM.
T4   CL_OS_SYSTEM=>INIT_AND_SET_MODES(
T5     I_EXTERNAL_COMMIT = OSCON_FALSE ).
8  START-OF-SELECTION.
T6 M = CL_OS_SYSTEM=>GET_TRANSACTION_MANAGER( ).
T7 T = T->CREATE_TRANSACTION( ).
9  FLIGHT_AGENT = CA_FLIGHT=>AGENT.
10 BOOKING_AGENT = CA_BOOKING=>AGENT.
11 TRY.
T8   T->START( ).
...
T9   T->END( ).
29   CATCH CX_OS_ERROR.
30 ENDTRY.
```

Tips for Working with Object Services

The Object Services framework is a powerful tool for managing object persistence and transactions. Like any new technique, we encourage you to practice what you have learned and integrate it into your own ABAP expertise. Try to create the airline seat reservation example in your development environment.

Along the way, we hope you find the following guidelines helpful as you design and use persistent classes:

- ✓ If you want to wrap legacy table data with a persistent class, use the business key object identity. You can then access the data with the table key (i.e., a semantic key).
- ✓ If you want to make large object graphs persistent, use the GUID object identity and persistent references.
- ✓ If you want to access an object graph with a semantic key, use a class with a business key object identity and a persistent reference to the root class of the object graph.
- ✓ You can mix business key and GUID object identities. You then can access a persistent object with both the semantic key and through a persistent reference.

Application Options to Consider

The Post-Processing Framework (PPF) is a generic service in the SAP Web Application Server that was created using Object Services. It enables applications to trigger actions such as sending e-mails or creating purchase order confirmations. From the PPF perspective, both the application and the actions

accomplish this task. Lines T1-T9 , which are shown in bold, represent the code for working with the Transaction Service.

Here's how this code works. In lines T3-T5, Object Services is explicitly initialized. Note that you must do this in the event block `LOAD-OF-PROGRAM`. In lines T6-T7, a transaction object is created. The actual transaction is started in line T8 and completed in line T9. This line replaces the `COMMIT WORK` statement in our example (see Listing 2, line 30).

Transaction Interoperability Considerations

To ensure interoperability with standard ABAP transactions, Object Services transactions are tightly coupled with the logical unit of work (LUW) concept in SAP.

Suppose a legacy application (i.e., a standard ABAP application) calls an Object Services component (i.e., a component that uses the Object Services Persistence Service). The Object Services runtime environment automatically creates and starts a top-level transaction. When the legacy application executes the ABAP `COMMIT WORK` statement, the top-level transaction is implicitly ended and the SAP LUW is finished. This scenario is called compatibility mode.

Suppose an Object Services application (i.e., an application that uses both the Persistence and Transaction Services) calls a legacy application

are represented as persistent objects, which enables uniform access to different application objects. This is a common pattern for designing generic tools.

This example is only one possibility for leveraging Object Services functionality in an application. Obviously you can take advantage of its features in many other ways. Once you feel confident using Object Services in simple ways in your applications, here are some suggestions for building more complex functionality:

- ✓ Try writing your own database access layer. Suppose your data is not stored in database tables, or you can only retrieve and save the data by means of function modules. You can map the attributes to the fields of a dictionary structure that works as a proxy. Note that you must also redefine the database access methods in the class agent.

- ✓ Try changing the behavior of the class agent. For example, you can add buffers to the class agent to retrieve multiple data per database access for performance reasons.

When to Avoid Using Object Services

Keep in mind that persistent object frameworks such as Object Services are generally not well suited for applications where performance is critical, such as those that handle millions of objects. This limitation arises from the bookkeeping overhead caused by automatic persistence management. In that case, a better option is to make your objects persistent manually by using ABAP Open SQL.*

* For more information on programming with Open SQL, see Adrian Görler and Ulrich Koch's article "Enhanced ABAP Programming with Dynamic Open SQL" in the September/October 2001 issue of *SAP Professional Journal*.

component (i.e., a standard ABAP component). When the Object Services application explicitly ends the top-level transaction, the Object Services runtime executes an ABAP `COMMIT WORK` statement and the SAP LUW is finished. The legacy component must not execute the ABAP `COMMIT WORK` statement. Otherwise, since the transaction boundaries are managed by the Object Services application, a runtime error will be raised.

Object Services top-level transactions have three update modes ("update task," "synchronous update task," and "local update task"), which correspond to the SAP LUW update modes. You must use the class method `INIT_AND_SET_MODES` of the class `CL_OS_SYSTEM` to set the update mode before the Object Services runtime environment is started. The default update mode is "update task." Alternatively, specify the transaction code for an object-oriented transaction with the "OO transaction model" field selected. Then the update mode is set when executing the transaction code.

Ready to Write Smarter ABAP Code?

In this article, we have showed you how the new Object Services layer in Release 6.10 expands the object-oriented design promise of ABAP Objects:

- ✓ The Persistence Service provides you with transparent object persistence, which means that most

aspects are handled automatically. (No more SQL code to make objects in a database persistent!)

✓ The Transaction Service gives you control over your transactions and provides you with nested in-memory transactions.

✓ You can now focus on building the business logic into your ABAP programs, instead of writing code to handle common application and system-level services.

While this article is only a starting point, we hope it gives you the confidence to try these techniques for yourself. And if you want to learn more about the Object Services layer, we recommend the online R/3 manual as an excellent reference source.

Stefan Bresch received his diploma in computer science from the University of Karlsruhe, Germany. Stefan joined SAP in 2000 and since then has been working on Object Persistence. He belongs to the Business Programming Languages Group in Walldorf and is currently working on serialization of ABAP data structures to XML representation. He can be reached at stefan.bresch@sap.com.

Christian Fecht studied computer science at the University of Saarland in Saarbruecken, where he received his Ph.D. in 1997. He joined SAP in 1998, and since then has been working in the Business Programming Languages Group. He is responsible for the ABAP runtime environment, especially for the ABAP Objects garbage collector. Christian was also involved in the design of the Object Services, and recently his focus has been on transparent persistence for Java. He can be reached at christian.fecht@sap.com.

Christian Stork studied mathematics and computer science at the Westfälische Wilhelms-University of Münster, Germany. He joined SAP in 1995 and worked for two years as a trainer, then returned to the Westfälische Wilhelms-University of Münster for his doctorate, specializing in algebraic geometry. In 2000, Christian rejoined SAP and became a member of the Business Programming Languages Group, where he works as a kernel developer. He is responsible for the implementation and maintenance of the Object Services, and is currently working on calling ABAP methods from XSLT. He can be reached at christian.stork@sap.com.

