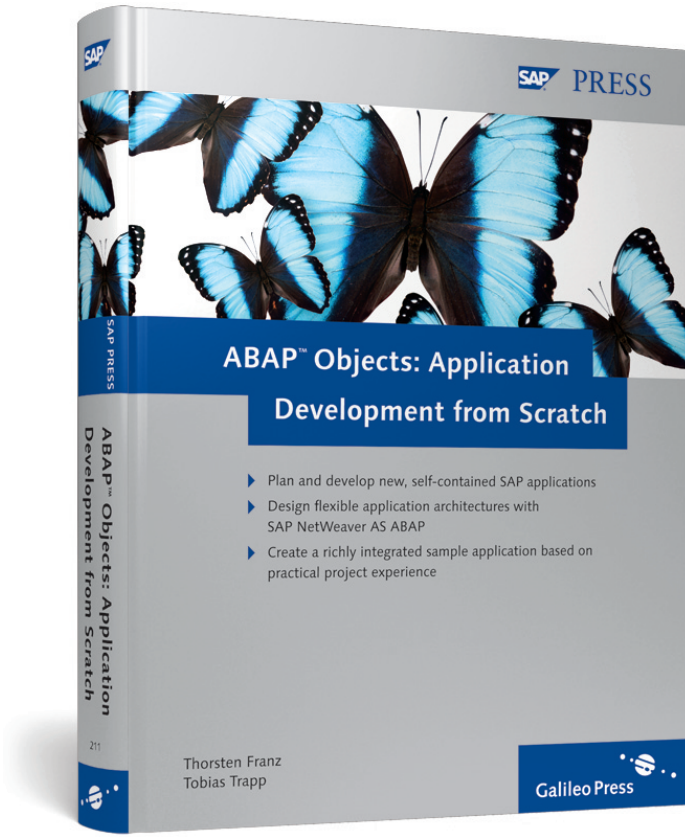
Thorsten Franz, Tobias Trapp

ABAP Objects: Application Development from Scratch



Galileo Press

Bonn • Boston

Contents at a Glance

Contents

"Objects serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation." (James Rumbaugh)

3 Application Object

Paradigm of object orientation

The paradigm of object orientation means that software is seen as a quantity of discrete objects that have a data structure and data behavior. This perspective is completely natural at the application design level, because this approach can be used to map the business entities of the conceptual design. Object-oriented programming involves creating an equivalent to the business entities in each case, such as invoices or documents. The following are characteristic features of an object designed and programmed in this way:

- An *object identity* can be defined: An object exists but once and must be uniquely identified.
- An object has properties that are encoded in *attributes*. These are defined in ABAP by ABAP Dictionary data elements.
- Objects also have behaviors that are encoded as *methods*.
- Objects of the same type (in terms of the same attributes and behavior) are categorized. They are *instances* (or also copies) of a class. In the sample application presented here, the central objects are instances of a vehicle class.
- Classes have a hierarchical structure through *derivation*. Derived classes are more specific than their predecessors. The basis classes are therefore true generalizations.

Implementing business processes flexibly

Object orientation helps you implement business processes flexibly, because you can keep several instances of business entities in an internal session at the same time. This lets you implement methods that can compare, copy, or possibly consolidate these instances.

In this chapter, you will learn about the techniques required to do this. For the Vehicle Management sample application, you will design an application object at the semantic level and create a data and object model.

In Section 3.2, Modeling the Application Object at the Database Level, we will discuss the mapping of the data model at the database level in detail. Section 3.3, Implementing Object Persistence, deals with implementing object persistence, as the name suggests. For example, how are ABAP objects saved to the database? We will introduce different types of database access layers, where the focus will be on *Object Services*. The SAP transaction concept is illustrated in Section 3.4, Transaction Concept, because you must thoroughly understand databases and SAP transactions for developing and using database access layers. The chapter concludes with Section 3.5, where we present best practices for all topics already discussed. This section will also describe specific topics, such as using change documents for persistent objects, for which you will need a number of the techniques developed in this chapter.

3.1 What Is an Application Object?

At the core of typical applications for SAP systems is a relatively complex application object whose data is saved to one or more tables and which contains specialized methods that can be used to manipulate the application object. Typical examples include business entities such as customer, invoice, document, or delivery.

Object-oriented analysis

How do you identify application objects? In almost all cases, the application object results from the product idea, whereby the application will need to manage a central business entity. If a system specification is object-oriented, you will find application objects for object-oriented analysis.

In addition, more than one application object may exist in an application system, in which case it is useful to modularize the application into subapplications. This is the topic of Chapter 5, Application Architecture.

You find application objects by breaking down an overall object model into logical parts. This procedure is the key to every successful modeling

process. Instead of being analyzed in the overall model, each more complex system is initially identified in submodels of greater cohesion.

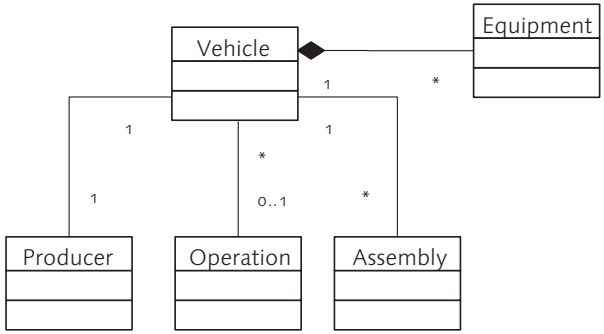


Figure 3.1 Application Object

Example of an application object

There are two central application objects in this sample application, *vehicle* and *request*. Figure 3.1 illustrates the strategic structure of the *Vehicle* application object in Unified Modeling Language (UML). The semantics are as follows:

A vehicle is normally assigned to exactly one operation where it is allowed. When a vehicle changes the operation to which it is assigned in an organization group, it temporarily might not be assigned to any operation. In fact, there is a *time frame* the user should also be able to recognize.

A vehicle has exactly one producer. To map the producer data, you can access the SAP Business Partner available on every SAP system. To map this producer, you can easily access the standard SAP system and create a field on the vehicle that you can use to refer to a business partner number.

A vehicle has a number of equipment parts, such as the type of passenger information (for example the recorded announcement of stops). However, this also includes fittings like seating, paintwork, and advertising (for example side panel, or full wrap advertising). You should model the equipment parts as independent objects. There is no provision made to enable you to assign a fitting to another vehicle.

Vehicles have assemblies. Unlike the equipment, they have a serial number, year of production, and so on. You should be able to assign assemblies to another vehicle.

Object-oriented design

The object-oriented analysis is followed by the design phase, where the object model that is used as the basis for the implementation is also developed. While the object-oriented analysis returns a domain model of the application, the result of the object-oriented design is a class model and sequence diagrams. Here, you should consider the class model and the requirements for database persistence, which must satisfy each business application object.

Different procedures exist in software technology for designing objects. Some software designers design an object model first and then develop a data model from this object model. However, the reverse is also possible, that is, you can design a data model and develop an object model based on it. Advocates of agile methods often favor incremental procedures. That is, for example, if you make modifications to the object model you will also need to make modifications to the data model, and vice versa. Agile methods in ABAP development are certainly a major challenge. If you choose this option, you will need detailed knowledge of the refactoring options of the ABAP Workbench, but should also be aware of the implications of a structural change to transparent tables.

Procedure

The procedure is as follows:

1. Starting with the class model from the analysis phase, you design the application object.
2. You then develop the data model at the database level, starting with the attributes and object model relationships.
3. Finally, you implement the object persistence, meaning you map the object at the database level.

Advantages

There are two advantages to this procedure: First, the focus early in the design phase is on the database model. You will be able to ensure in good time that it is optimized in terms of access paths, satisfies revisory factors, and can be found easily in the archive. Second, you can implement the object persistence using the *Object Services* of the ABAP Workbench.

In the following text, we will discuss the aspects you must take into account when designing application objects, such as the granularity of the object model.

Granularity of the object model

In most cases, the application object is not an instance of an individual class, but instead consists of compositions and associations to other classes. Examples include header and item data that you frequently find in SAP ERP applications. It may prove useful to implement these types of compositions at the level of structured data types, rather than using a wide variety of classes, and therefore use the options of internal tables and, if necessary, also deeper data structures. This lets you reduce the number of runtime objects as well as easily access quantities of object attributes as internal tables. This approach is particularly beneficial in performance-intensive applications. This procedure is often used in EDI processes, where, for example, the *Data delivery (external)* application object has a quantity of raw data in an external data format that cannot be implemented as an independent object. In fact, after typing this raw data and instancing a new *Data delivery (internal)* application object, you must not model temporary data, such as address components (which may involve you having to update the master data at a later stage) as separate objects.

Using business objects from the standard SAP system

In the next step, you should try to use business objects from the standard SAP system. You must define a producer for a vehicle in the sample application. It would be useful here to map the possible business partners as SAP Business Partners and include the business partner number as a reference in an attribute in the `Z_CL_VEHICLE` class. An example of the class model obtained using this approach is shown in Figure 3.2.

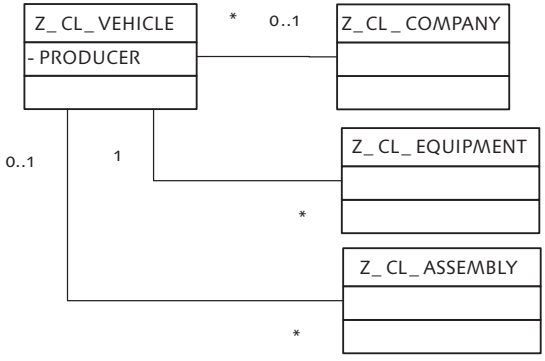


Figure 3.2 Design of an Application Object

The Z_CL_VEHICLE class has an attribute that specifies a business partner number in the producer role.

Virtual attributes

Not all attributes of an object must have a one-to-one correspondence to a field in a database table. In addition to database attributes that represent the fields of assigned database tables, do you also want virtual attributes that will be calculated, or read from other tables? In this case, you do not need to take the corresponding attributes into account in the data model. After you have finished designing the application object, you can begin modeling it at the database level.

3.2 Modeling the Application Object at the Database Level

The database design is a key component of every business application. You can develop it based on a model for object-oriented design. Alternatively, you can develop a semantic data model, implement it based on the ABAP Dictionary, and derive an object model from this. You will learn about this procedure in conjunction with the Structured Entity Relationship diagram in Section 3.2.1.

You must be particularly thorough when creating data models. It is difficult to modify the data model of an application that is already live because you have to write comprehensive conversion programs. If you have just archived live data, the effort required will be even greater, because you will have to check the effect on archiving objects (Transaction AOBJ) when a modification is made.

Two aspects

You focus on two aspects when modeling data: the data model must enable you to access data efficiently and have a simple structure so that it can be evaluated by DataSources for SAP NetWeaver BI, or enable you to display archives using methods of the standard SAP system.

3.2.1 Structured Entity Relationship Model

You can use the *SAP Data Modeler* integrated into the ABAP Workbench to create data models. Figure 3.3 shows a small portion of the SAP_10130 SAP Business Partner data model . This example also illustrates that you can use data models to document existing applications.

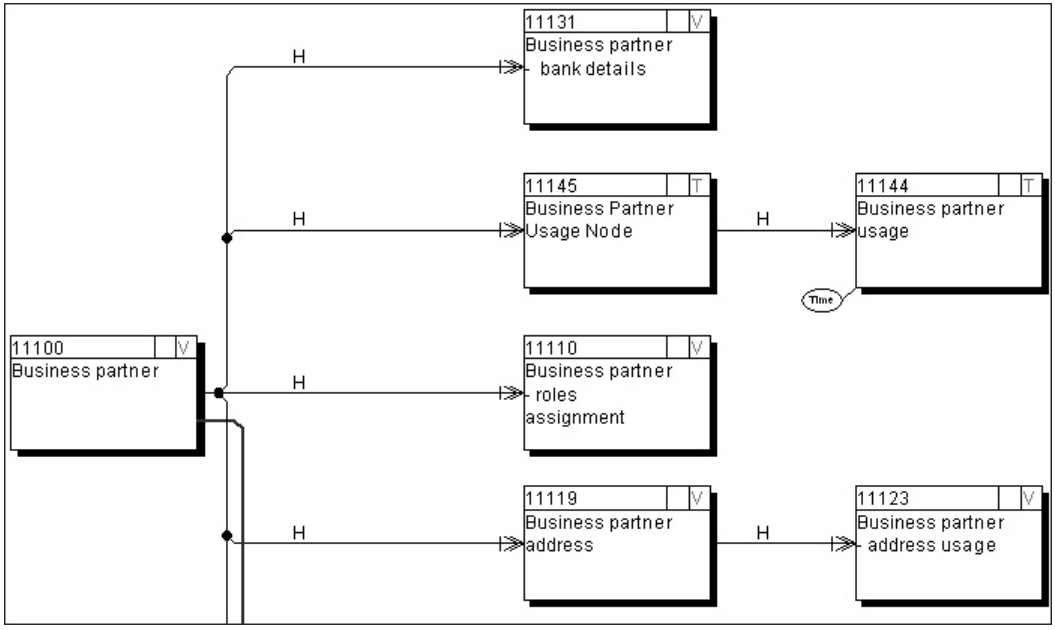


Figure 3.3 Partial SAP Business Partner Data Model

Unlike the Entity Relationship diagram, the *Structured Entity Relationship diagram* (SERM) consists of directional arrows. An arrow means that the target entity depends on the source entity in terms of the relationship of a header record to item records. An item record can therefore only exist if a header record exists (this is called *existence dependency*). In this case, the letter "H" above the arrow stands for "hierarchical". The double or single arrowheads indicate that several entities, or only one single entity, participate(s) in the relationship. A single line signifies that the relationship is optional, meaning that an element may also be missing on the target side.

Data Modeler

You call the Data Modeler from Transaction SD11, but it is also integrated into the ABAP Workbench. You can display data models or their entity types in Transaction SE80 under Business Engineering • Data Models or Business Engineering • Entity Types.

You will use the Data Modeler to develop an SERM diagram from the design model shown in Figure 3.2. You will create a data model called ZVEHICLE first.

You will set up a ZVEHICLE business object in Chapter 6, Application Layer; therefore, it will be useful to assign the data model using this approach.

To do this, you will create entity types for all classes:

Creating entity types

- The ZVEHICLE entity type corresponds to the vehicle data.
- ZOWNER defines the assignment of a vehicle to a company. A vehicle is only assigned to one company at a time; you can use the time frame to map different transfers of vehicle ownership.
- ZEQUIPMENT represents the equipment data. Like ZOWNER, this has a hierarchical (that is, existence-based) dependency on the vehicle entity type, although no time dependency is involved here.
- ZASSEMBLY represents assemblies. These have a referential relationship to the vehicle entity (letter "R" above the arrow). Letter "C" (for "conditional") specifies a conditional relationship. This modeling demonstrates that an assembly can also exist without a vehicle if it is removed from one vehicle and built into another vehicle later.

Figure 3.4 shows the data model, visualized by the SAP Data Modeler.

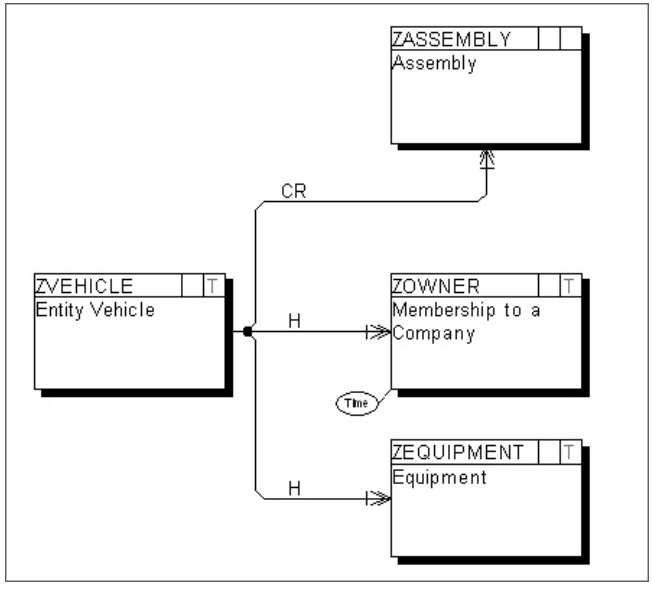


Figure 3.4 Vehicle Data Model

Creating data models

You can create this data model using the ABAP Workbench by first creating the entity types displayed in the figure. Other documentation options are also available here, for example, you can create SAPscript texts for all entities and for the data model. For each entity type, define detailed relationships (using the button of the same name or F6) to other entities that correspond to the arrows shown in Figure 3.4. You still have to assign the entity types to the data model using the Hierarchy button, as you can see in Figure 3.5. You can still subsequently move the entities if you think that the positioning selected by the Data Modeler is unclear.

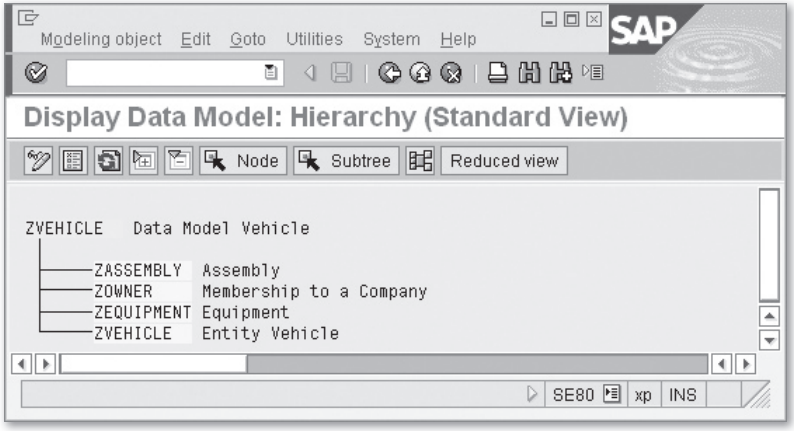


Figure 3.5 Data Model Entities

Data models of business objects

You will find the Data Modeler in the SAP Library under the keyword *Data Modeler (BC-DWB-TOO)*. The SAP SERM model also provides other relationship types (aggregating and external) and specializations for modeling inheritances. You can also create submodels, hierarchically nest them, use external relationships to refer to them and link data models to business objects to ensure that you achieve an integrated description of the application object, consisting of object behavior and object data, in the SAP system. SERM models are generally quite suitable for data modeling composite objects, because they are in the third normal form in terms of database theory.

Normalization can be seen as optimizing the data model, which results in redundancies and anomalies being avoided when you access databases.

We will not be able to discuss the theory of normal forms as it is beyond the scope of the book.

The SAP Data Modeler enables you to assign entity types to ABAP Dictionary elements such as transparent tables or views. This means that you can link SERM models to the database model. Mapping the SERM model in the ABAP Dictionary is the topic of Section 3.2.2, Data Modeling at the ABAP Dictionary Level.

Normalization

Knowledge about database normalization is an important basis for each data modeling, as well as an essential requirement here. Information about this topic can also be found in every book about databases.

Normalized data models correspond to the semantic data model from the business blueprint, and SAP SERM models are usually already in the third normal form. Normalized data models do not have any redundancies and no anomalies occur for database operations. However, they are not optimized in terms of database access. In this type of optimization, you specifically narrow down the most common requests, and also accept redundancies in the data model, to minimize the number of requests. You achieve this by keeping fields in database tables redundant and saving the results of calculations (such as aggregated values at the database level) in the database. The following forms of denormalization are widely used in SAP development:

Forms of denormalization

- The most common form of denormalization is keeping attributes redundant. You usually save the last and first names of business partners or classification criteria from other tables.
- For temporary data, you can duplicate tables for which you expect large data volumes. If these tables are distributed to different partitions at the database system level, parallel access can be accelerated. You can use Customizing to control which tables to select for which transaction data.
- If you are sure that a number of attributes will not be populated in most cases, you can create a separate table for them and create the corresponding records only as required.

Denormalization [+]

The following rules apply for optimization of the database model:

- You must not use denormalization lightly; Denormalization is only useful for optimizing frequently-performed access to databases.
- Denormalization is often avoided for write-only applications to prevent change anomalies.
- Read-only applications are often considerably denormalized to prevent joins, aggregations, and calculations at the database level.

3.2.2 Data Modeling at the ABAP Dictionary Level

The ERM and SERM data models are characteristically semantic. You must develop a data model out of them that consists of transparent tables. The relevant aspects required for this are described in the following text, and include defining primary and foreign keys, indexes, database locks, saving unstructured data, and so on.

Primary Key

Each transparent table contains one primary key with a unique constraint. Two table rows must have different primary keys. Master and transaction data should also contain the client in the primary key to ensure that the application system has multitenancy capabilities.

Unique constraint

To create primary keys, you can define either specialized or technical keys. A specialized key is created by the attributes that define the identity of an object as a whole. You must use specialized keys with caution because if the specialization changes, or if you discover that the key has been modified, extensive recoding will be necessary.

Number ranges enable you to generate keys for application objects. This also allows you to store specialized information in the keys by defining subobjects and including fiscal years. You define number ranges in Transaction SNRO. Note the following when using number ranges:

Number range objects

- **Specialized components**
  Specialized components in a number range help persons responsible when they are processing business objects, because they can memorize these components, if necessary. Specialized components also sup-

port developers and support. Under no circumstances should a program evaluate these specialized components because this would cause the disadvantages of specialized keys to be inherited. There can be exceptions if you want to merge data from different SAP systems as part of a merge project. However, this requires assigning the numbers disjointly in the different systems, which involves additional coordination effort.

- **Assigning external numbers**
  You should assign the numbers outside of the SAP system using an *external number assignment*. These types of cases are common for migration projects or external data transfers from other systems.
- **Deactivating buffering**
  If you want assigned numbers to have the character of a document, you must deactivate main memory buffering to ensure that the numbering is consistent, even if the Logical Unit of Work (SAP LUW) were to terminate. A *logical unit of work* (or SAP transaction) is a sequence of related database operations, which are discussed in detail in Section 3.4, Transaction Concept. For more information about number range buffering, refer to the *SAP Number Range Buffering (BC-CST-NU)* section in the SAP Library.
- **Assigning a number**
  The `NUMBER_GET_NEXT` function module accesses the database and requires a certain amount of access time for activated main memory buffering.
- **Maintaining interval statuses**
  Each customer must maintain the statuses of number range intervals. The statuses cannot usually also be delivered with the software, because the current values are also transferred in this way. This often makes no sense in the target system, or is even risky if you want to install the software again.[1]

1 You can, of course, use a report to set the initial statuses of the number ranges by manipulating the transparent `NRIV` table. However, this approach is extremely risky, because multiple uses mean that the uniqueness of the numbers may no longer be possible. Consequently, you may no longer be able to determine primary keys. We therefore advise you to always write these programs as securely as possible to eliminate the risk of statuses being reset that have already been maintained.

Transaction SNRO

You create number range objects in Transaction SNRO, as shown in Figure 3.6. In this case, you define a domain for the number length, buffer the numbers in main memory, and specify to issue a warning if only ten percent of the numbers are still available.

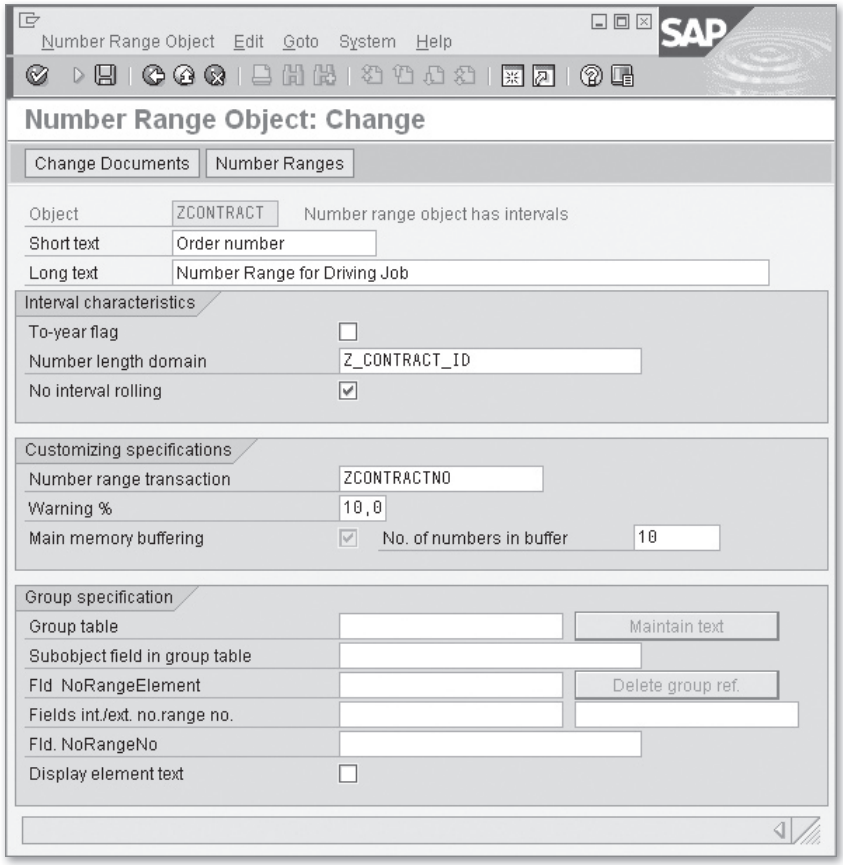


Figure 3.6 Creating Number Range Objects

Intervals

After creating the number range object, you create the intervals. Intervals must not overlap; exceptions are only allowed if there is a reference to the fiscal year. If you assign a number, you must specify an interval where the generated numbers should be located. Alternatively, you can specify that the numbers roll back. This means that when you start an interval the numbers are assigned again from the beginning as soon as the upper limit is reached (Transaction SNUM, see Figure 3.7). You can

also identify intervals as "external". In that case, the number is assigned by the user or an external system. This type of number range interval can also contain alphanumeric characters if this is allowed by the domain used. You can use the NUMBER_CHECK function module to check whether a number belongs to a certain number range interval.

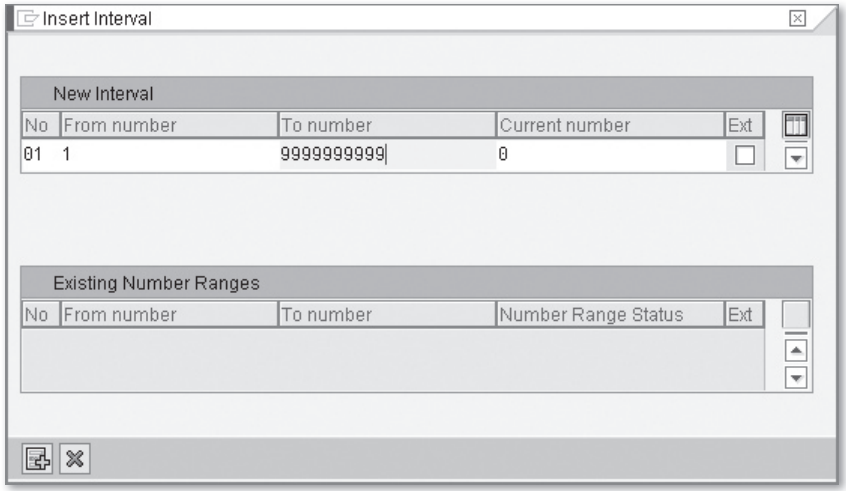


Figure 3.7 Creating Number Range Intervals

Fiscal years

You can also classify number ranges according to fiscal year. You can classify number range objects by subobjects. In the example, you can create a separate domain for different request types and assign your own intervals for each characteristic value.

Number range groups

Other structuring is also possible, using *number range groups*. The standard example in Materials Management is the MATERIALNR number range object for the material master, for which the material type (MTART data element) determines the number range interval using Table T134 (which is specified in the number range object).

You will find plenty of examples of this procedure in the SAP ERP environment. However, we advise against overdoing this by assigning numbers with too many specialized aspects. We will use a negative example to confirm this: Assume that you assign numbers for different materials depending on the vendor. As a result, you receive a key (either struc-

tured or concatenated as a fixed-length string) that contains a year, the domain value depending on the vendor, the interval number, and the actual number. You must first make sure that none of the parts of the key for specialized processes are evaluated in the application. If this is actually the case, there may be a risk of nasty surprises if the specialization changes, for example, if two vendors merge. Experience shows that the risks of complex number range objects together with subobjects and groups outweigh the advantages in most cases, unless sophisticated data modeling was maintained.

For more information about number ranges, refer to the SAP Library under BC Extended Applications Function Library • Number Ranges.

Global Unique Identifier (GUID)

Global Unique Identifiers (GUIDs) enable you to generate technical keys. Using GUIDs makes using number ranges effortless, and you get a purely technical key that enables you to merge data from different live instances of the application system again. You create GUIDs using the `GUID_CREATE` function module of the `SYGU` function group. As of Release 7.0, GUIDs have the benefit of distribution properties, which means that you can easily receive packaging options as part of parallel processing. We also recommend this key assignment for application objects that are used in cross-system processes and for which uniqueness is therefore essential. We examine this aspect in Section 9.2.2, Prerequisites, as part of parallelization strategies.

Based on experience, GUIDs are not suitable for display in screens. They are also not user-friendly for the person responsible; numbers are easier to use.

Defining External Keys

You can define external keys in the ABAP Dictionary but they are not represented by external keys in the database. Consequently, you cannot guarantee the referential integrity of the dataset using database tools.

However, this does not mean that you should forego defining external keys. You use them to document the data model of the application system. You can also use external key relationships to implement

F4 help and input checks on dialog boxes. You can generate complete maintenance dialogs specifically for maintaining Customizing tables and simple master and transaction data, as you will see in Chapter 7, GUI Programming.

Mapping Inheritance

Options

Different options are available for expressing the inheritance of classes at the database level. You can use separate tables for each class and map the entire hierarchy in a single table:

- **Mapping an entire class hierarchy in one table**
  The advantage of mapping an entire class hierarchy in a table is that one class can be converted easily into another class. This is the case in the example here when a vehicle changes its type. Database queries are also usually easy to organize. However, the disadvantage is that often, a lot of memory space is wasted in the database because not all attributes are required in every class. Perhaps more annoying than the waste of memory space is the lack of transparency: You cannot easily identify to which class attributes belong.
- **Mapping a class to a table**
  Mapping a class to a table comes closest to object-oriented modeling. In addition, you can change each class irrespective of other classes, and delete and add attributes as you wish. The disadvantage of this approach is that the number of database tables is very high, and access can be complicated. The effort required also increases if you want to add attributes to or delete them from all tables when you modify the data model. Queries can be very complex when you have a wide range of tables.

Inheritance relationships

In many business applications, inheritance relationships are expressed by different composition relationships to other objects. For example, the parent class is usually abstract and represents the header data of an entity. The different derived classes all have the same header data but may have different item data. The problem of mapping inheritance does not arise in these situations.

Locks

You use database locks to ensure that several internal sessions (users or jobs) do not access the same object at the same time, which could create inconsistent statuses, or situations where only data written by the last person would be saved. What effects could not having locks have? The first is lost modifications. When two programs read a value from the database in a local variable in parallel and change and write this value back, the last transaction that made modifications overwrites the modifications made by the previous transaction. Even if only one transaction writes data, while another transaction accesses read-only data, you will not be able to rule out inconsistencies in the second transaction. You must also protect reading processes against writing processes.

Dirty reads

Dirty reads are another problem. If a transaction is working on an entry and later executes a `ROLLBACK WORK`, written entries will be reset again in the meantime. If another process is using this data before it is "released" by `COMMIT WORK`, there is a risk that invalid data will be used.

Locks can have different granularities. For example, you can lock single records, but you can also lock entire tables or use different lock concepts, namely *optimistic* and *pessimistic* locks. The pessimistic lock concept involves locking a database before each access, whereas the optimistic lock concept means that you generally forego database locks and use a time stamp to check whether a record has changed in the meantime. This saves time and effort required to manage database locks. *Isolation levels* of the database management system are closely associated with this.

ABAP lock concept

ABAP development also has a lock concept that is not based on database locks and is described in the SAP Library under *The SAP Lock Concept (BC-CST-EQ)*. Here, you manage locks in main memory on an e*nqueue server* that exists on the central instance of the SAP system or on a separate server. If a work process is not running on the instance of the enqueue server, a lock request is sent through the dispatcher and the message server. Nevertheless, managing these locks does not require as much effort as managing database locks, because you manage them in main memory.

Deadlocks

`INSERT`, `UPDATE`, `MODIFY` or `SELECT ... FOR UPDATE` set database locks for database modifications only. This type of database lock is held until

the next database commit, but not necessarily until the end of the LUW. These locks can be the cause of *deadlocks* if other processes access the same data and each process is waiting for the other process to release the lock again.

SAP locks and database locks

However, there are other differences between SAP locks and database locks. You can also create SAP locks for records that do not yet exist in the database. You must use SAP locks when you work with updates, because the database modifications are only implemented in the update and no database locks can exist beforehand.

Creating lock objects in the ABAP Workbench

You create lock objects in Transaction SE80, as shown in Figure 3.8. For example, create a somewhat more complex lock object for which you can lock several tables of an application object, that is, the equipment parts for a vehicle (`ZEQUIPMENT` table) and the owner (`ZOWNER` table). As a result, additional lock parameters are created for the components of the primary key of all tables involved.

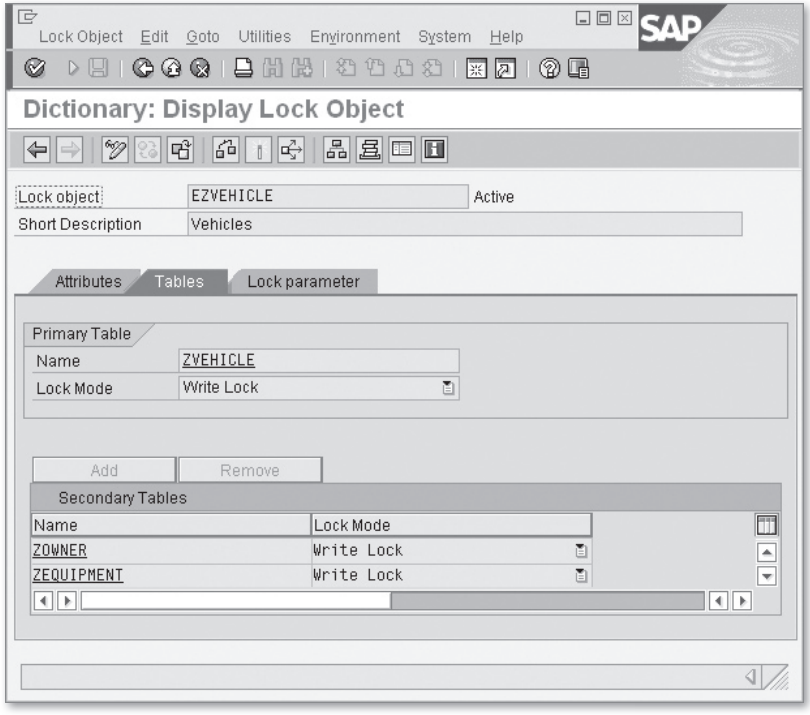


Figure 3.8 Creating a Lock Object

Secondary Tables with Lock Objects [!]

If you create the primary keys of the dependent tables in such a way that they contain the primary key fields of the original table, and if you also define foreign key relationships, you can include the dependent tables as secondary tables in the lock object and also lock these tables. If this is not the case, you may have to define additional lock objects for dependent tables.

Enqueue modules

You generate an enqueue module for a lock object. This enqueue module has several standard parameters. The `mode` parameter specifies the lock mode. Some lock parameters you use to specify the area to be locked include the `_scope` parameter, which describes the interaction of locks and updates. You can use the `wait` parameter to configure the number of repeat attempts and the `_collect` parameter to buffer the lock requirements locally in a lock container before the `FLUSH_ENQUEUE` function module collects and transfers them into the lock table.

The following lock modes exist:

Exclusive lock and exclusive but not cumulative lock

- The *exclusive lock* ("E" lock mode) can be requested several times per session by the lock owner but there must be no shared or exclusive locks from other lock owners. Exclusive locks cumulate and must be released again several times.
- The *exclusive but not cumulative lock* ("X" lock mode) can be requested only once in the internal session, provided there are no exclusive or shared locks from the same lock owner or other lock owners.

Shared locks

- You can have several *shared locks* ("S" lock mode) in parallel but there must be no additional exclusive locks. You can use shared locks to protect displayed data records from being modified.
- *Optimistic locks* are shared locks you can convert to exclusive locks by calling a lock with the "R" lock mode. In that case, other existing locks will be invalidated.

Optimistic Lock Concept

Because the optimistic lock concept is relatively new, we will explain it in more detail at this point. SAP's optimistic lock concept sets shared locks, and therefore protects against modifications (provided other optimistic locks were not converted into exclusive locks). The SAP optimistic lock concept is suitable if different processes set shared locks, of which only a small number actually want to modify data later. This is the exact analogy for the lock concept from classic database development. There, you forego database locks and use time stamps to query whether a modification occurred in the meantime.

You call the lock module as shown in Listing 3.1.

```
DATA:
  ls_msg     TYPE        scx_t100key,

CALL FUNCTION 'ENQUEUE_EZVEHICLE'
  EXPORTING
    mode_zvehicle   = iv_enqmode
    mode_zequipment = iv_enqmode
    mode_zowner     = iv_enqmode
    id              = lv_id
    _scope          = iv_scope
    _wait           = iv_wait
  EXCEPTIONS
    OTHERS          = 3.
IF sy-subrc <> 0.
  ls_msg-msgid = sy-msgid.
  ls_msg-msgno = sy-msgno.
  ls_msg-attr1 = sy-msgv1.
  ls_msg-attr2 = sy-msgv2.
  ls_msg-attr3 = sy-msgv3.
  ls_msg-attr4 = sy-msgv4.
  RAISE EXCEPTION TYPE zcx_locking_error
     EXPORTING        textid = ls_msg.
ENDIF.
```

Listing 3.1 Calling a Lock Module

The central component of this listing is the call for the generated `ENQUEUE_EZVEHICLE` lock module, to which you transfer four parameters: the lock mode in the `IV_ENQMODE` parameter with the "E" default value, the `IV_`

SCOPE parameter with the "2" default value and the IV_WAIT parameter with the SPACE default value because generally, you will want to repeat unsuccessful lock attempts.

You can also use this lock object to delete the dependent tables, because no values are transferred for the NR and TIME parameters, which, together with the ID, create the primary key of the tables involved.

The "2" value of the SCOPE parameter is the default value for updates and causes the lock to be forwarded to the update process.

Removing locks

If locks are not transferred to the update and deleted there, you must remove them. You do this similarly to the way you set locks using generated function modules. The following sample source code shows the sequence of the call:

```
CALL FUNCTION 'DEQUEUE_EZVEHICLE'
  EXPORTING
    mode_zvehicle   = iv_mode
    mode_zequipment = iv_mode
    mode_zowner     = iv_mode
    id              = lv_id
    _scope          = iv_scope
    _synchron       = iv_synchron.
```

The iv_synchron parameter used here means that the lock module might wait until the entry is actually deleted from the lock table. However, this is only useful in a few cases; therefore, you use a method to make this parameter optional when the removal of a lock is being encapsulated, and set SPACE as the default parameter.

Indexes and Buffering

Accelerating read database access

Two techniques for accelerating read database access include buffering and using database indexes. Buffering and using indexes are two mutually exclusive concepts. In one case, you want to keep table contents in main memory; in the other, your objective is to be able to access the database efficiently. Buffering is a difficult topic for which you should refer to SAP-specific literature such as *SAP Performance Optimization Guide* by Thomas Schneider (SAP PRESS, 2008). However, we have compiled

the most important features you need to be aware of with regard to buffering:

- ABAP SQL commands exist that read past the table buffer.
- Buffered items are held at the level of the application server and must be synchronized with one another. This means it can take some time until a database modification appears on another application server.

Suitability

Based on experience, buffering is most suitable for small transparent tables that have only a few entries and are not modified very often, for example, Customizing tables or some types of master data.

You can define database indexes in the ABAP Dictionary. Because you need resources to manage these indexes, you should only create them for the relevant access paths. These are often known at design time; otherwise you should determine them using runtime analysis (Transaction SE30). For more information, see ABAP • Analysis Tools • Runtime Analysis in the SAP Library.

[!]

Disadvantages of Indexes

The time an index saves when reading is lost when writing. Inserting many individual data records is particularly intensive when you have to maintain the indexes each time. When you load data into SAP NetWeaver BI, you therefore delete the indexes before writing and create them again when all data is loaded into the system. Even if you use these strategies in application programming only in the rarest of cases, you should always consider this factor and define database indexes as economically as possible. You should also use bundling techniques, which we discuss in Section 3.3.1, Necessity of Database Access Layers.

Determining runtime gains

These tools and Transaction ST30 (global performance analysis) are used to determine runtime gains on live systems. For example, you can store tables, which are often accessed in parallel, on different disks. However, you can generally only perform these types of optimization for a specific live system.

NULL Values

Transparent tables can contain NULL values. They can be tricky for the uninitiated because at first glance (for example, in Transaction SE16),

they are difficult to differentiate from initial values. If you load a data record with NULL values into a working area using `SELECT`, initial values will also appear in the corresponding fields. You can only access NULL values in the `WHERE` condition using `SELECT` if you use `IS NULL`; the `IS INITIAL` check or the check for inconsistencies with another value will not work.

Semantics

Before you learn how to create NULL values, we will explain the semantics of these values. A NULL value can represent an unknown value. For example, an object can have a certain characteristic but you do not know it at the moment. NULL values can map non-existing values. A typical situation is where you cannot use an attribute in a specific case.

However, a NULL value can also mean that you do not know whether a value exists. Every employee experiences the standard example of this situation when their company telephone system is being changed and their old number is no longer valid. There is a transition period where the employee may not know whether there is a new number or, if there is a new number, he may not yet know it.

How do you create NULL values? In the ABAP Dictionary, you can specify whether you want every column of a transparent table to be filled with initial values or NULL values. In the second case, NULL values are added when you create a new column or insert entries using a view that does not contain these fields.

Benefits

There are limited benefits to NULL values for an ABAP programmer because they have no equivalent in ABAP. NULL values are implicitly converted into INITIAL values when loaded into working areas or internal tables so you cannot tell whether there was a NULL value. This is why these values are often used only temporarily after structural modifications in the ABAP Dictionary, when a new column has to be converted by a migration report that must decide between entries that have not yet been processed and entries processed by the initial content.

Archiving

Not many applications delete business data; for auditing and legal reasons, deleting data is not supported. Instead, deletion indicators are set for the business objects to be deleted and these objects will be excluded

from future searches and processing. This increases the volume of data in the database, response times for database queries rise, backups take longer, and sometimes you have to get new hardware for the database server. The best way to prevent this is to avoid data and to delete temporary data. If you have exhausted these options, you will have to archive data. For more information, refer to the example given in the SAP Library under *CA-ARC Archiving*.

Archive Development Kit

SAP provides an API for archiving functions with the *Archive Development Kit* (ADK). You can define archiving objects for an application object. You create write, read, and delete programs for these archiving objects, which write the data to be archived into an archive file, evaluate this data in a second process, and then delete it from the database. *Archive indexes are created in the process.* These are transparent tables that contain the object key of the objects to be archived, a reference to the archive file, and the item in the file, from which the object can be read. For the sake of completeness, we must mention that XML archiving is also possible. In this case, you create an XML document for each document to be archived and save it in a file. The scale of the data to be archived increases because of the XML markup. You can use methods of the standard SAP system to search in the archive index and look for corresponding data in a view similar to the Data Browser, that is, perform a direct access. It is unusual to restore data to the live system after it has been archived.

How to handle archiving in detail is beyond the scope of this book; therefore, we will only refer to the factors that can affect the data modeling of the application:

- **Archiving criteria**
  Are there simple criteria you can use to decide when you want an application object to be archived? You must ensure that there are no references of active application objects to archived objects. This could cause the program to behave incorrectly.
- **Archive search**
  What criteria should you use to be able to search for database entities in the archive? Should the most important attributes be displayed in the archive index to ensure that direct access to the object you are looking for is not necessary? Assuming that direct access is required, are the methods of the Archive Information System (Transaction SARI)

in the standard SAP system sufficient, or do the separate dialogs have to be archive-enabled?

- **Modifications to the data model**
  Should you expect modifications to the data model? Can you make these modifications in such a way that you do not have to modify the archiving programs? Will you need to modify archived data?

3.3 Implementing Object Persistence

Design errors

In the previous sections, you have seen how you can model application objects at the database level. Some developers question why additional development work is required. Is it not enough to access the database using Open SQL and manipulate individual table entries? Do additional wrappers not mean increased development effort, reduce performance and make programs more complicated? These arguments may apply in individual cases but generally turn out to be hazardous design errors. Specifically, the problem is that, in most cases, you can no longer reverse the conversion of Open SQL calls scattered in the application. This argument applies in particular when other applications access the database tables without access layers.

In the following sections, we will first explain the need to encapsulate access to the application object. You will then implement an access layer for the modeled application object using Object Services. Finally, we will discuss when it makes sense to implement your own persistence mechanisms.

Management service and search service

You need the following services to be able to use persistence mechanisms easily and efficiently:

- **Management service**
  You must ensure that several, possibly contradictory, runtime objects do not exist in the same internal session. This type of *management service* should also contain a cache to enable you to access the managed objects more quickly.
- **Search service**
  A management service should also have a *Search service* that enables you to search for objects based on specialized criteria and load them

into main memory. A search should also be able to return only a constant number of objects to enable you to work in packages if you have large sets of objects.

3.3.1 Necessity of Database Access Layers

Object-oriented access layers

Database access layers are an old and established concept. They were previously implemented by BAPI interfaces, which you still require if you want data to be read by Remote Function Call (RFC) or Web services, or to be manipulated. However, there are also much simpler reasons for implementing database access layers:

- **Transaction mechanisms**
  One important reason is the transaction mechanism itself. If you only work with Open SQL access without using update techniques, this means that the data is irreversibly saved to the database each time an implicit commit is performed. Where dialog applications are concerned, you are therefore not in command of transaction control and cannot guarantee data consistency. This also applies in batch programs. In a batch program, you cannot execute any more RFCs for accessing data in external systems or for the internal parallelization of the processing, because these implicitly perform a database commit.[2] This also makes it more difficult to enhance the programs. An implicit commit must not be performed in any enhancement by a BAdI implementation. To avoid these unwanted consequences, you would have to buffer the database modifications and persist them at the end of the LUW, which already very closely reflects a database access layer.
- **Subsequent processes**
  Another problem with explicit database access is caused by the fact that you cannot implement any new subsequent processes for data modifications for the application system. If we assume that, for reasons of revision security, you want to write change documents for certain fields, replicate these changes by Application Link Enabling (ALE), or provide the delta queue when connecting to SAP NetWeaver

2 The same problem can occur when you debug, because database commits are also performed there. If you save data to the database without updates, the debugging of the application is risky because inconsistencies may occur.

BI, you must add these functions to all of the points in programs that conduct database updates. You must not include subsequent processes in the area of data replication. Loose event linkages (for example, for starting workflows) are also possible. These are presented in Section 5.5.3, Event-Based Interfaces, under the keyword Publish & Subscribe interfaces. In practice, you will no longer be able to perform a subsequent implementation into a live application, unless you perform a complete redesign.

- **Robustness**
  Implementing a database access layer can also make an application more robust. For professional reasons, you are often not allowed to write any values into a table, but rather the programs must require minimum data plausibility in order to work. A typical example is a situation where certain fields are not allowed to contain any initial values; filling these fields with the wrong values equates to an error situation. You cannot assume that everyone who fills the database tables is fully aware of these conditions. Therefore, it is absolutely essential that you explicitly encode these conditions in one location and check them before the `INSERT`.

- **Modularization**
  Hiding details about the data model is another important reason for creating an access layer. Developers often want to have the freedom to modify and optimize a data model in the future if performance problems were discovered in mass tests. If this means that you have to adjust SQL calls at every location where a database is accessed, you will find it very difficult to modify the data model because the application system is not sufficiently modularized.

- **Checking authorizations**
  You may require specific authorizations to read sensitive data. For example, if a VIP indicator is set for customer-related master data, only users with special authorizations can have full access to this data. In such a case, data access modules must perform certain authorization checks and may not necessarily offer all data. You cannot currently avoid bypassing the programming of data access modules. However, by defining package interfaces, you can find out through the Code Inspector whether interfaces were bypassed in the program-

ming (see Chapter 5, Application Architecture). In Release 7.0 you can use dynamic `SELECT` statements or generated programs to avoid defined database access functions in package interfaces and access tables directly. You can use the Code Inspector to check the use of these potentially "dangerous" commands.

Alternatively, you have situations where you do not want to use a database access layer:

- You want to create a report for a user department that will determine entries according to specified criteria in one or more tables and output them in an ALV grid.

Migration

- As part of a migration you want to load a dataset into the SAP system in as short a time as possible.
- In some tables, you have to modify certain data records and fill additional attributes. You can do this using an XPRA program when making an ABAP Dictionary modification to a table structure.
- Within data archiving, you have to write large quantities of data into an archive file within a short time frame, and then delete this data in the database and create search indexes.

Even if the examples indicate that you often work directly with Open SQL tools in the database in mass process, you should not make this procedure the norm. There is also no reason to impose severe restrictions at the development phase of an application system already, simply because implementing a database access layer requires conceptual and implementation effort.

Are there reasons against Object Services?

You can drastically reduce the implementation effort by using Object Services. Strangely, object-oriented access layers have a bad reputation that is entirely unfounded. This poor reputation is based on the fear that an object could be too finely detailed, resulting in lots of individual database operations in mass processes, drastically affecting performance. However, there are solutions to this problem in the form of bundling techniques.

Bundling techniques

The standard SAP system's Object Services use bundling techniques using generic update tasks. The data to be modified is collected and transferred as a binary-formatted table to an update module that accesses

the database dynamically. This procedure shows that you can perform mass modifications through a framework and completely hide it from the user.

If you use your own persistence mechanisms, you can save the modifications to be updated in an ABAP class or in the global memory of a function group and then execute the `PERFORM … ON COMMIT` command to implement them as mass inserts into a form routine of a function group. In this case, we specifically recommend that you use the non object-oriented `PERFORM … ON COMMIT` construction and not the modern variant by creating an `on_commit` method as the event handler for the `TRANSACTION_FINISHED` event of the `CL_SYSTEM_TRANSACTION_STATE` class using the `SET HANDLER on_commit` command. The reason for this is the following: Experience has shown that you cannot guarantee with certainty that updates started in the event handler are also executed in the same SAP transaction.

One problem with object-oriented access layers can be that the instancing of an object requires an overhead, which in some cases cannot be justified. This is the case, for example, if the user is searching for objects based on certain criteria and a number of attributes are displayed for this in an ALV grid. After the user selects an individual object, this is specifically loaded and shown in a display dialog. Another application scenario where instancing is not required is packaging as part of parallel processing. You want to determine packages of objects but do not need any time-consuming instancing to do this. A *search service*, that is, a class that can provide primary keys of object instances without instancing objects, can be useful in both of these application scenarios.

Complex search criteria

A search service can only consist of a static method of an application object that queries the database using direct `SELECT` statements. However, there are also application systems where you have to implement very complex search criteria that are required for controlling automatic processes. In the most complex case, these types of search services include objects that were instanced but whose status differs from the status in the database. You can even implement these techniques across sessions using *shared objects*, although the effort required to implement this type of object is usually too great. For more information about shared objects, refer to *ABAP – Shared Objects* in the SAP Library.

Object Search Services

Based on experience, search services are the basis of numerous application scenarios in business programming and guarantee high-performance, object-oriented programs. Their implementation will therefore also be the topic in the following chapters.

3.3.2 Object Services

No uniform persistence mechanism

In individual software development, either a lot of effort is invested in creating persistence layers, or commercial products are used. These software products implement mappings of objects to tables, of a database-independent persistence, and of caches and supported transactions. In ABAP development, a uniform persistence mechanism was previously not widely accepted. This was mainly because database updates occur predominantly in update modules and BAPIs that are developed individually. It was only relatively late in the game that the option of persistent ABAP classes was made possible with Object Services and an object-oriented framework made available for transactions. These functions are described in *ABAP Objects* by Horst Keller and Sascha Krüger, but also in the SAP Library under ABAP Object Services. The aim of the persistence layer is to be able to work with objects without having to worry about persistence. You instance objects and access their attributes using SET and GET methods. You then collect the modifications and write them to the database in separate update modules after the `COMMIT WORK`. The result of this procedure is that you do not need separate update modules. You can also use Object Services with classic update techniques, but you must take into account that, after a `COMMIT WORK`, a persistent object is invalidated and has an initial status. This means that accessing the object in an update module will fail if this object was already accessed in the LUW.

What must you keep in mind when using Object Services?

- **Mapping**
  The options for mapping the application object to transparent tables are somewhat restricted. For special techniques such as deriving or mapping different classes to a transparent table, refer to the SAP Library.

- **Query service**
  A query service using the methods of the `IF_OS_CA_PERSISTENCY` interface is available, but it always instances the objects found and is therefore not always suitable for mass processes.
- **Service interfaces**
  The user must program different service interfaces such as lock management or creating change documents.
- **Transaction concept**
  The Object Services were incorporated into the classic transaction concept. There are different transaction modes, including an object-oriented transaction concept that makes a transaction manager available in the form of an API. This lets you implement transparently complex scenarios such as nested transactions, however, there is no coexistence with the classic LUW concept: The `COMMIT WORK` or `ROLLBACK WORK` commands will cause a runtime error. We present the aspects of transaction control in detail in Section 3.4.

Reservations

Many developers have reservations about Object Services, due primarily to a perception of poor performance. The basic technique was optimized in this case. For example, rather than vast numbers of update modules being executed, the data to be updated is instead bundled and written into a generic update task, to which binary serialized data is transferred. This ensures that mass updates are implemented in a small number of database operations.

Creating persistent classes

An example of using Object Services in the sample application is shown in the text that follows. You create a persistent class called `ZCL_VEHICLE` in the ABAP Workbench, as shown in Figure 3.9.

You then define the persistence mapping using the Persistence button (or Ctrl + F4). A dialog box called Add Table/Structure appears, where you enter the name of the `ZVEHICLE` transparent table you created earlier. Finally, you select all table fields sequentially in the lower Persistence Representation area. Each field then appears automatically in the middle drop-down box, where you can set the visibility. Select protected visibility for all fields, but only private visibility for the modification information because you want to be able to access the attributes of a class easily

through a working area. This, however, is only one service function that you will develop in the following section.

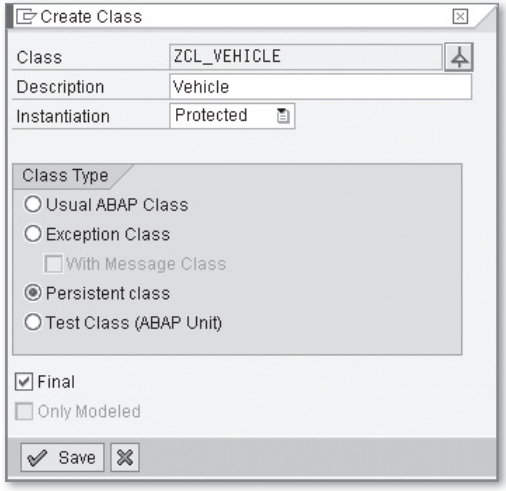


Figure 3.9 Creating a Persistent Class

Subsequently, two classes will be generated automatically: The `ZCA_VEHICLE` class agent and its `ZCB_VEHICLE` basis class. The class agent is a singleton object and therefore only exists once in an internal session. The actual database access takes place in the basis class agent; as does the object-relational mapping using the `MAP_LOAD_FROM_DATABASE_KEY( )`, `MAP_LOAD_FROM_DATABASE_GUID( )` or `MAP_SAVE_TO_DATABASE( )` methods. The basis class agent is derived from the `CL_OS_CA_COMMON` class; therefore, you can redefine methods and also access the logic of a persistent class to define your own object-relational mapping, for example. The most important interfaces for the user are `IF_OS_FACTORY`, `IF_OS_CA_PERSISTENCY` and `IF_OS_CA_INSTANCE`.

Implementing requirements

First, you will implement several requirements in the persistent class that will make it easier to work with the persistent class:

1. All equipment parts for a vehicle are to be automatically loaded from the database into an internal table when you instance objects. This means that you will be able to display these parts easily and save them using mass inserts when you save the vehicle data.

2. The change information (changed by, date, and time) should be determined automatically at the time of the COMMIT WORK.

Using the checking agent

The second item is easy to implement. You implement the IF_OF_CHECK interface in the ZCL_VEHICLE class and create an implementation of the IS_CONSISTENT( ) method. You execute this *checking agent* after COMMIT WORK and before you execute the update task. You normally use this to implement consistency checks for an object and, in the case of an emergency, prevent data from being saved. You use it to set the change information:

```
METHOD if_os_check~is_consistent.
  TRY.
      set_chdate( i_chdate = sy-datum ).
      set_chtime( i_chtime = sy-uzeit ).
      set_chuser( i_chuser = sy-uname ).
    CATCH cx_os_object_not_found .
*     The object has been deleted.
  ENDTRY.
  result = oscon_true.
ENDMETHOD.
```

To ensure the IS_CONSISTENT( ) method is executed, you must register it. You do this by redefining the INIT( ) method of the IF_OS_STATE interface (see Listing 3.2).

```
METHOD IF_OS_STATE~INIT.
************************************************************
* Purpose        : Initialization of the transient state
*                  partition.
* Version        : 2.0
* Precondition   : -
* Postcondition  : Transient state is initial.
* OO Exceptions  : -
* Implementation : Caution!: Avoid Throwing ACCESS Events.
************************************************************
* Changelog: 2000-03-07   : (BGR) Initial Version 2.0
************************************************************
* Modify if you like
************************************************************
DATA:
  lr_tm type REF TO if_os_transaction_manager,
  lr_t  type REF TO if_os_transaction.
```

```
  lr_tm = cl_os_system=>get_transaction_manager( ) .
  lr_t = lr_tm->get_current_transaction( ).
  lr_t->register_check_agent( me ).
ENDMETHOD.
```

Listing 3.2 Registering the Checking Agent

Transient attributes

You can also use the initialization routine to load the equipment parts of a vehicle into a *transient attribute* that is not managed by the persistence service. Transient attributes are calculated at runtime and can be used to save data of dependent objects (see Section 3.3.4, Accessing Dependent Tables).

The source code from Listing 3.3 shows an example of how you create, modify, and delete a vehicle object in a transaction.

```
DATA:
  lr_vehicle_agent TYPE REF TO zca_vehicle,
  lr_vehicle       TYPE REF TO zcl_vehicle,
  lv_vehicle_id    TYPE        z_vehicle_id.

lr_vehicle_agent = zca_vehicle =>agent.
CALL FUNCTION 'GUID_CREATE'
  IMPORTING
    ev_guid_22 = lv_vehicle_id.
TRY.
    lr_vehicle = lv_vehicle_agent->create_persistent(
        i_id = lv_vehicle_id ).
    TRY.
        lr_vehicle->set_seats( i_seats = '2' ).
      CATCH cx_os_object_not_found.
*     Perform exception handling.
    ENDTRY.
 lr_vehicle_agent->delete_persistent( lv_vehicle_id ).
 CATCH CX_OS_OBJECT_NOT_EXISTING .
*  Perform exception handling.
ENDTRY.
COMMIT WORK.
```

Listing 3.3 Using Object Services

At the end of the transaction, the modifications are collected and written to the database by an Object Services update task. You can work

with the object in this way, without having to worry about persistence factors. (Using Object Services in the update would also generally cause an exception.) In the example, you used Object Services in the classic transaction mode instead of using the object-oriented transaction service of the `IF_OS_TRANSACTION` interface. You can execute the Object Services within classic LUWs. In contrast, the `END( )` method of the Object Services that completes an LUW implicitly calls a `COMMIT WORK` so that classic updates can also be performed in an object-oriented transaction. These aspects are discussed in Section 3.4.

Deleting Persistent Objects [!]

When you delete persistent objects, note that deleting an object that does not exist or no longer exists does not trigger an exception, but instead only causes an update module to terminate.

3.3.3 Inheriting Persistent Classes

Use it sparingly

You use inheritance when there is a specialization relationship between a class and a number of subclasses. However, you should only use inheritance sparingly at the application object level, because there is a risk that you may link two business entities too closely together. If, in hindsight, your decision turns out to be incorrect, the cost of changing the program and database model of the database will in most cases be far greater than the benefits gained through reuse.

Nevertheless, inheritance is a powerful tool:

- **Polymorphy**
  You can derive persistent classes and keep the mapping to achieve polymorphic behavior, for example, redefining methods: In the application sample presented here, you can avoid there being instances of coaches with standing room, define special checks, or implement display transactions.
- **Enhancing using additional attributes**
  If you have to include additional attributes in a derived class that do not have meaningful semantics in the basis class, you can create them in an additional transparent table. This prevents transparent tables

Index

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z