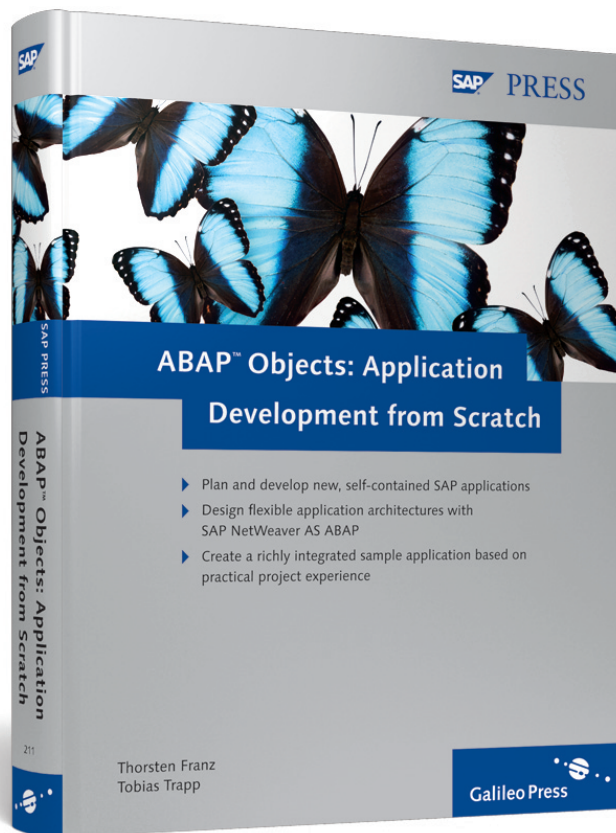


Thorsten Franz, Tobias Trapp

ABAP Objects: Application Development from Scratch



Galileo Press 

Bonn • Boston

Contents at a Glance

1	Introduction	17
2	Designing Application Systems	23
3	Application Object	51
4	Classes, Interfaces, and Exceptions	113
5	Application Architecture	135
6	Application Layer	187
7	GUI Programming	229
8	SAP Business Partner	331
9	Application Programming Techniques	391
10	Information Acquisition	447
A	Managing Development Projects	465
B	Bibliography	487
C	List of Quotations	491
D	Authors	493

Contents

Preface	15
1 Introduction	17
1.1 About this Book	18
1.2 Sample Application and Technical Prerequisites	21
2 Designing Application Systems	23
2.1 Requirements	24
2.1.1 Requirements Analysis as a Process	24
2.1.2 Functional Requirements	25
2.1.3 Non-Functional Requirements	25
2.1.4 Limits of Functional and Non-Functional Requirements	34
2.1.5 System Specification	37
2.2 General Architectural Considerations	39
2.2.1 Product Families: Separating Frames and Content	39
2.2.2 Metadata	44
2.2.3 Generative Programming	46
2.2.4 Model-Driven Architectures	48
2.3 Using the Standard SAP System	49
3 Application Object	51
3.1 What Is an Application Object?	52
3.2 Modeling the Application Object at the Database Level	56
3.2.1 Structured Entity Relationship Model	56
3.2.2 Data Modeling at the ABAP Dictionary Level	61
3.3 Implementing Object Persistence	75
3.3.1 Necessity of Database Access Layers	76
3.3.2 Object Services	80
3.3.3 Inheriting Persistent Classes	85
3.3.4 Accessing Dependent Tables	88

- 3.3.5 Query Service 92
- 3.3.6 Developing a Separate Persistence Service 93
- 3.3.7 BAPI Access Methods 94
- 3.4 Transaction Concept 95
 - 3.4.1 Special Techniques of the Classic Transaction
Concept 98
 - 3.4.2 Object-Oriented Transaction Concept 99
- 3.5 Best Practices 102
 - 3.5.1 Creating Primary Keys 102
 - 3.5.2 Modeling the Application Object in
the ABAP Dictionary 104
 - 3.5.3 Service Functions for Persistent Objects 104
 - 3.5.4 Saving Unstructured and Semi-Structured Data ... 107
 - 3.5.5 Further Considerations 110
 - 3.5.6 Key Transactions 111

4 Classes, Interfaces, and Exceptions 113

- 4.1 Advantages of ABAP Objects 114
 - 4.1.1 Defining Constants in Classes and Interfaces 115
 - 4.1.2 Function Groups versus Objects 115
 - 4.1.3 Events 116
- 4.2 Exceptions 116
 - 4.2.1 Classic and Object-Oriented Exceptions 117
 - 4.2.2 Assertions 120
 - 4.2.3 Exception Handling 121
- 4.3 Basic Principles of Object-Oriented Design 122
 - 4.3.1 Dependency Inversion 123
 - 4.3.2 Open-Closed Principle 124
 - 4.3.3 Inheritance and the Substitution Principle 126
 - 4.3.4 Testability Using Unit Tests 127
- 4.4 Classic Modularization Units 130
 - 4.4.1 Function Modules 130
 - 4.4.2 Reports 130
- 4.5 Best Practices 132
 - 4.5.1 General Considerations for Object-Oriented
Design 132
 - 4.5.2 Key Transactions 133

5 Application Architecture 135

- 5.1 Requirements for Application Architecture 135
- 5.2 Software Structuring from a Technical Perspective 137
- 5.3 How To Structure a Software System 141
 - 5.3.1 Taking Account of the Business Structure 141
 - 5.3.2 Identification of Layers 144
 - 5.3.3 Dividing Applications into Sub-Applications 145
 - 5.3.4 Creating Basic Components 146
 - 5.3.5 Dependency on SAP Standard Components 147
 - 5.3.6 Structure of the Sample Application 147
- 5.4 Package Concept 148
 - 5.4.1 Package Interfaces and Checks 150
 - 5.4.2 Visibility of Package Interfaces 153
 - 5.4.3 Structure Packages and SAP Software Components 154
 - 5.4.4 Excursion: Compatibility Problems 157
 - 5.4.5 Excursion: Naming Conventions and Namespaces 159
- 5.5 Composition of Packages 161
 - 5.5.1 Runtime Configuration of Software Components 162
 - 5.5.2 Using Enhancements to Implement Interfaces 167
 - 5.5.3 Event-Based Interfaces 170
- 5.6 Best Practices 179
 - 5.6.1 Architecture Documentation 179
 - 5.6.2 Characteristics of Package Splitting 180
 - 5.6.3 Interface Design 182
 - 5.6.4 Package Check Mode 183
 - 5.6.5 Outlook 185
 - 5.6.6 Key Transactions 185

6 Application Layer 187

- 6.1 Application Logic 188
 - 6.1.1 Implementing the Application Object 190
 - 6.1.2 Separation of Object and Process 193

- 6.2 Customizing 196
 - 6.2.1 Basic Principles 197
 - 6.2.2 Technical Customizing 198
- 6.3 Search Services 204
- 6.4 Workflows 209
 - 6.4.1 Sample Scenario: Resubmission on a Specific Date 211
 - 6.4.2 Key Transactions 226

7 GUI Programming 229

- 7.1 Ergonomic Examples and Dialog Standards 230
 - 7.1.1 SAP R/3 Style Guide 231
 - 7.1.2 Ergonomic Examples 231
 - 7.1.3 Menu Standards 233
 - 7.1.4 Screen Layout and User Guide 233
- 7.2 Table Maintenance Dialog and View Cluster 238
 - 7.2.1 Generating and Enhancing Table Maintenance Dialogs 239
 - 7.2.2 View Clusters 250
 - 7.2.3 Tips for Handling Maintenance Views and View Clusters 258
- 7.3 Area Menus 258
- 7.4 Object-Oriented Screen Programming 261
 - 7.4.1 Pros and Cons of Subscreens 261
 - 7.4.2 Subscreens as a Modularization Unit 262
 - 7.4.3 Encapsulating with Screens 263
 - 7.4.4 Message Handling with Screens 263
 - 7.4.5 BUS Screen Framework 264
 - 7.4.6 Advantages of Object-Oriented Screens 266
 - 7.4.7 Uses for the BUS Screen Framework 267
 - 7.4.8 Normal Screens and Modal Dialog Boxes 267
 - 7.4.9 Defining Flow Logic 269
 - 7.4.10 Creating Instances 270
 - 7.4.11 Calling Screens 271
 - 7.4.12 Sequence of Processing Events 271
 - 7.4.13 Defining Your Own Screen Logic 273
 - 7.4.14 Setting Titles and GUI Statuses 273

- 7.4.15 Handling User Inputs 273
- 7.4.16 Collecting and Issuing Error Messages 277
- 7.4.17 Embedding the Business Application Log 280
- 7.4.18 Table Controls and ALV-Grids 283
- 7.4.19 Screens with Subscreen Areas 283
- 7.4.20 Defining Subscreens 285
- 7.4.21 Data Transfer Between Screen Fields and
Screen Class 286
- 7.4.22 Tabstrips 287
- 7.4.23 For Advanced Users: Selection Screens
and Screen Painter 294
- 7.4.24 Selection Screens in Conjunction with the BUS
Screen Framework 300
- 7.4.25 Outlook 306
- 7.5 Web Dynpro 306
 - 7.5.1 Basic Principles 307
 - 7.5.2 Creating a Sample Application 310
 - 7.5.3 Modification-Free Extensions Using Dynamic
Programming 317
- 7.6 Best Practices 327
 - 7.6.1 Choosing the Right GUI Technology 327
 - 7.6.2 Software Factors 328
 - 7.6.3 Key Transactions 329

8 SAP Business Partner 331

- 8.1 Background Information 331
 - 8.1.1 The Creation of SAP Business Partner 332
 - 8.1.2 Conceptual Overview 333
 - 8.1.3 First Impression 334
- 8.2 Business Partner Extension 336
 - 8.2.1 Example of an Extension 336
 - 8.2.2 Maintaining the Application 338
 - 8.2.3 Maintaining the Data Set 339
 - 8.2.4 Maintaining Tables 339
 - 8.2.5 Maintaining Field Groups 340
 - 8.2.6 Views (Transaction BUS3) 341
 - 8.2.7 Sections (Transaction BUS4) 343

8.2.8	Screens (Transaction BUS5)	343
8.2.9	Screen Sequences (Transaction BUS6)	345
8.2.10	BP Views (Transaction BUSD)	345
8.2.11	Creating Role Categories and Roles	345
8.2.12	ZVHM_BUPA Function Group	347
8.2.13	Screen 0100	348
8.2.14	Events	349
8.2.15	BDT Naming Conventions	363
8.2.16	Testing the Extension	364
8.2.17	Troubleshooting	366
8.2.18	Summary	369
8.3	SAP Locator Extension	369
8.3.1	Introduction to the SAP Locator	369
8.3.2	Aim of the Extension	370
8.3.3	Transaction LOCA_CUST	370
8.3.4	Definition of the Hierarchy	374
8.3.5	Creating an Append Search Help	375
8.3.6	Creating the Elementary Search Help	376
8.3.7	Assigning the Search Help to the Append Search Help	376
8.3.8	Creating a Function Group	377
8.3.9	Creating a Search Screen	377
8.3.10	Form Routine to Initialize the Search	380
8.3.11	Form Routine to Get Search Fields	380
8.3.12	Form Routine to Set Search Fields	381
8.3.13	Form Routine To Create the Screen Object	381
8.3.14	Creating a Function Module	382
8.3.15	Creating a Local Search Class	384
8.3.16	Providing the Search ID in the Locator Customizing	386
8.3.17	Testing the Search	387
8.3.18	Summary	388
8.4	Key Transactions	388

9 Application Programming Techniques 391

9.1	Implementing the Application Log	392
9.1.1	Log Recipients	393
9.1.2	Log Research as a Business Process	393

- 9.1.3 Business Application Log (BAL) 395
- 9.1.4 BAL Data Model 396
- 9.1.5 Application Programming Interface (API) 396
- 9.1.6 Example: Creating and Displaying a Log 397
- 9.1.7 Example: Saving the Log 399
- 9.1.8 Transaction Concept 401
- 9.1.9 Enriching Logs 403
- 9.1.10 Saving Complex Data 412
- 9.1.11 Using Additional Callbacks in the Display 415
- 9.1.12 User-Defined Buttons 416
- 9.1.13 Deleting and Archiving Logs 417
- 9.1.14 Summary 417
- 9.1.15 Additional Information 417
- 9.2 Parallel Processing of Applications 417
 - 9.2.1 Use case 418
 - 9.2.2 Prerequisites 420
 - 9.2.3 Asynchronous Remote Function Call (aRFC) 424
 - 9.2.4 Parallelization with Background Jobs 437
 - 9.2.5 Parallelization with the Parallel Processing Tool
BANK_PP_JOBCTRL 439
 - 9.2.6 Summary 444
 - 9.2.7 Additional Information 444
- 9.3 Key Transactions 445

10 Information Acquisition 447

- 10.1 SAP Service Marketplace 447
 - 10.1.1 SAP Help Portal 447
 - 10.1.2 SAP Support Portal 448
 - 10.1.3 SAP Developer Network 448
- 10.2 ABAP Keyword Documentation 450
- 10.3 SAP Design Guild 451
- 10.4 Internal Workings of AS ABAP 451
 - 10.4.1 Debugging 451
 - 10.4.2 Information Sources in the SAP System 453
 - 10.4.3 Runtime Analysis 454
 - 10.4.4 Database Trace 457
 - 10.4.5 Environment Analysis 458

10.5 Knowledge Management 459
10.6 Key Transactions 461

Appendices 463

A Managing Development Projects 465
 A.1 Roles in Development Projects 465
 A.1.1 The Role of the Chief Software Designer 465
 A.1.2 Frameworks and Tools 466
 A.2 Quality Management 467
 A.2.1 Risk Management 467
 A.2.2 Development Guidelines 469
 A.2.3 Code Inspections and Enhancement of the
 Code Inspector 470
 A.2.4 Creating a Documentation 480
 A.2.5 Enabling a Check 481
 A.2.6 Software Test 482
 A.2.7 Documentation 483
 A.2.8 Key Transactions 486
B Bibliography 487
C List of Quotations 491
D The Authors 493

Index 495

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

3 Application Object

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:

Paradigm of object orientation

- ▶ 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.

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.

Implementing business processes flexibly

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 sub-applications. 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.

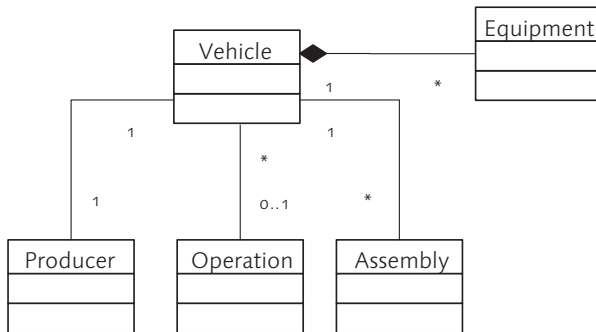


Figure 3.1 Application Object

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:

Example of an application object

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.

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.

Granularity of the object model

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.

Using business objects from the standard SAP system

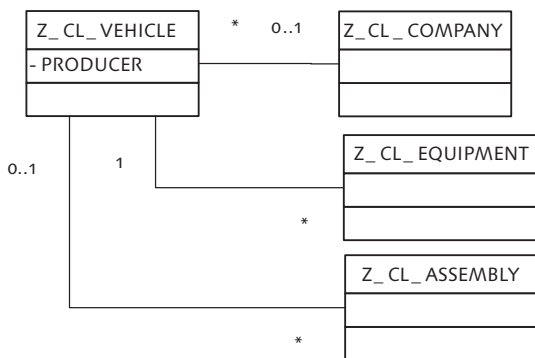


Figure 3.2 Design of an Application Object

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.

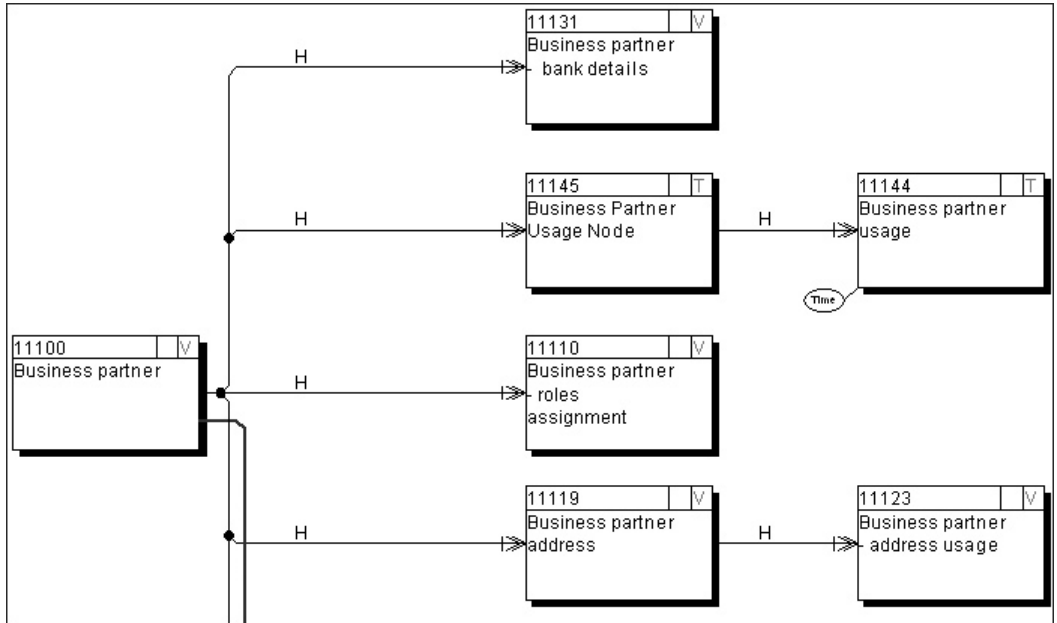


Figure 3.3 Partial SAP Business Partner Data Model

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.

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.

Data Modeler

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.

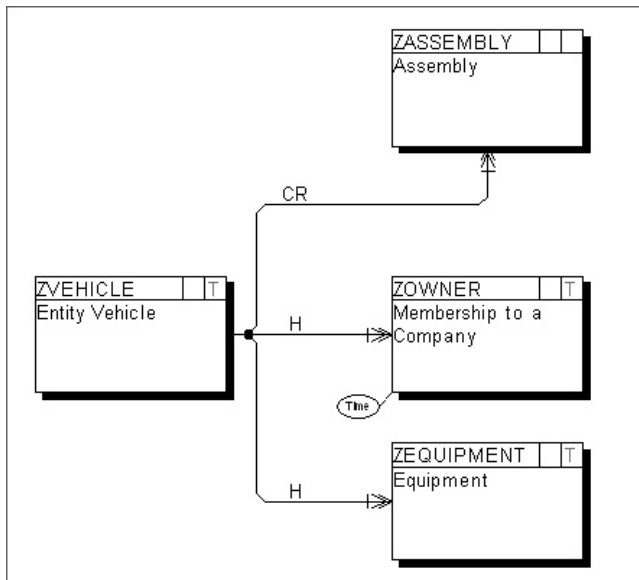


Figure 3.4 Vehicle Data Model

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.

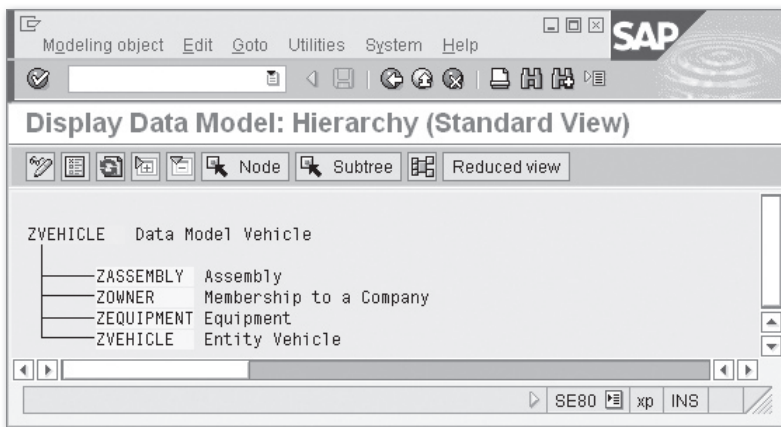


Figure 3.5 Data Model Entities

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.

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.

Transaction SNRO

Figure 3.6 Creating Number Range Objects

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

Intervals

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.

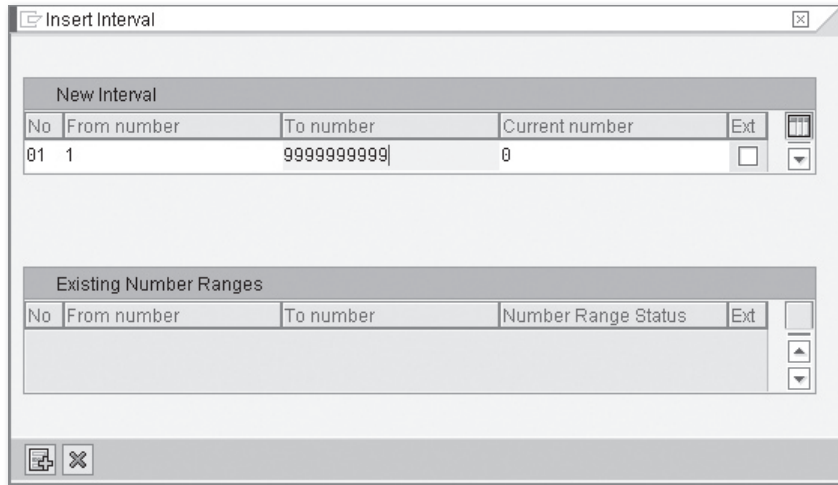


Figure 3.7 Creating Number Range Intervals

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 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.

Global Unique Identifier (GUID)

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 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. Dirty reads

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 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 *enqueue 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. ABAP lock concept

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

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.

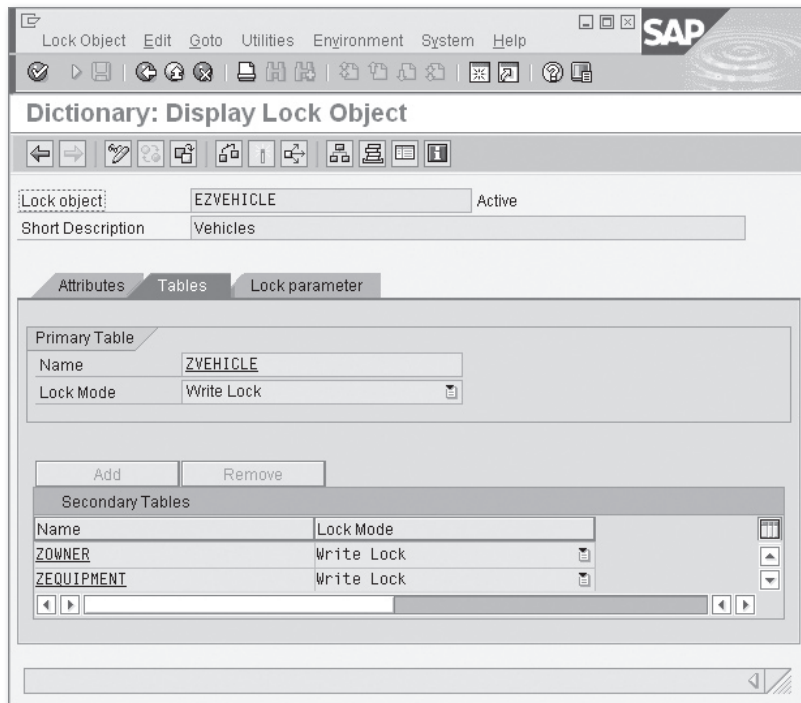


Figure 3.8 Creating a Lock Object

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.

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.

Enqueue modules

The following lock modes exist:

- ▶ 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.
- ▶ 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.

Exclusive lock and exclusive but not cumulative lock

Shared locks

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.

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:

Removing locks

```
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

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

Accelerating read database access

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.

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.

Semantics

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.

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.

Benefits

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

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.

Design errors

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.

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

Management service and search service

► **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 XPRG 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.

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.

Complex search
criteria

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.

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.

Reservations

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.

Creating persistent classes

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.

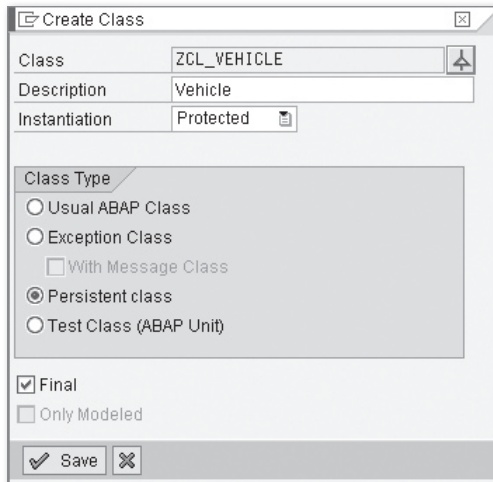


Figure 3.9 Creating a Persistent Class

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.

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:

Using the checking agent

```
METHOD if_os_check-is_consistent.
  TRY.
    set_chdate( i_chdate = sy-datum ).
    set_ctime( i_ctime = 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 = lr_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

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.

Use it sparingly

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

- ABAP
 - Class*, 213
 - Dictionary*, 61, 104, 142, 150, 183, 187
 - Keyword documentation*, 450
 - Lock concept*, 67
 - Objects*, 114, 116
 - Unit*, 128
 - Workbench*, 213
- ACID
 - Properties*, 97
- Activation error, 152
- Additional fields, 406
- Administrability
 - Customizing contents*, 29
 - Live system*, 28
- Administration
 - Environment*, 29
 - Tool*, 175
- Administrator
 - Cockpit*, 29
 - Specialist*, 28
- Aesthetics, 135
- Agile method, 25, 54
- ALV grid, 229, 276, 283
- Analysis
 - Object-oriented*, 52
- Application
 - Administrator*, 394
 - Architecture*, 135
 - Layer*, 187
 - Log*, 401
 - Logic*, 188
 - Object*, 19, 51, 52, 53, 55, 56, 104, 190, 195, 196
 - Object, example*, 53
 - Programming*, 391
- Application data
 - Check*, 44
 - Merge*, 33
- Application Link Enabling (ALE), 76, 189
- Application Programming interface (API), 396
- Architecture, 179
 - Documentation*, 179
 - Model-driven (MDA)*, 48
- Archive
 - Migration*, 194
 - Search*, 54, 74
- Archive Development Kit (ADK), 74, 166, 417
- Archive information system, 74
- ArchiveLink, 192, 193
- Archiving, 73, 414
 - Criteria*, 74
- Area menu, 29, 258, 259, 260
- AS ABAP, 17, 451
- ASSERT, 120
- Assertion, 120, 121
- Association, 55
- Attribute, 51
 - Public*, 92
 - Transient*, 84, 89
 - Virtual*, 56
- Authorization, 77
 - Check*, 77, 189
 - Concept*, 190
 - Object*, 189

B

- Background
 - Job*, 437, 445
 - Processing*, 132
 - Work process*, 436
- BANK_PP_JOBCTRL, 439
- Basic
 - Class*, 165
 - Component*, 146
 - Function*, 146
- Basis component, 198

- Batch program, 131
 - Blog, 449, 461
 - BOR object, 192, 213
 - BPX, 448
 - Breakpoint, 121
 - Bundling technique, 72, 78
 - Business Add-In (BAI), 126, 167, 168, 169, 170, 181, 183
 - Business Application Log (BAL), 33, 117, 130, 392, 396, 407
 - BAL_DB_DELETE*, 415
 - BAL_DB_LOAD*, 397
 - BAL_DB_SAVE*, 397
 - BAL_DSP_LOG_DISPLAY*, 396, 397, 413
 - BAL_DSP_PROFILE_SINGLE_LOG_GET*, 405
 - BAL_LOG_CREATE*, 413
 - Detailed information*, 407
 - Interactive functions*, 394
 - Log display*, 395
 - Log object*, 396
 - Log research*, 394
 - Log sub-object*, 396
 - Business Application Programming Interface (BAPI), 45, 94, 142, 189, 454
 - Business Data Toolset (BDT), 45, 93, 261, 333
 - Application object*, 333
 - Application owning the table*, 333
 - Applications*, 338
 - BUPA application object*, 337
 - Current memory*, 348
 - Data sets*, 339
 - Direct input*, 46
 - Event*, 350, 351, 352
 - Field group*, 340
 - Field grouping*, 45
 - Global memory*, 348
 - Naming conventions*, 363
 - Participating application*, 334
 - Regenerate subscreen containers*, 366
 - Screens*, 343
 - Screen sequence*, 345
 - Section*, 343
 - Tables*, 339
 - Views*, 341
 - Business object, 55, 59, 192
 - Delegation*, 126
 - Business Object Builder, 126, 192
 - Business partner, 19, 331
 - Role*, 332, 366
 - Business process, 51
 - Business Process Expert (BPX), 448
 - Community*, 448
 - Business Server Pages (BSP), 229, 327
 - Business Workplace, 230
 - BUS Screen Framework, 261, 264, 265, 267, 300, 388
 - Button, 314
 - User-defined*, 416
- ## C
-
- Calculation method, 219
 - Callback, 396, 415
 - Interface*, 167
 - Method*, 428
 - Routine*, 410, 411, 427
 - Routine, specify*, 411
 - CALL DIALOG, 427
 - CALL SCREEN, 268, 427
 - CALL SUBSCREEN, 285
 - CALL TRANSACTION, 427
 - CALL TRANSFORMATION, 48, 108
 - Cascading Stylesheets (CSS), 328
 - Change document, 105, 106
 - Check
 - Class*, 474
 - Enabling*, 481
 - Function*, 27
 - Mode*, 152
 - Report*, 259
 - Checking agent, 83, 90
 - Checkpoint group, 120, 121
 - Class, 51, 113, 137
 - Abstract*, 115
 - Agent*, 104
 - CL_ABAP_EXPIMP_DB*, 414
 - CL_ABAP_OBJECTDESCR*, 203
 - CL_ABAP_ZIP*, 109
 - CL_APL_ECATT_TDC_API*, 482
 - CL_AUNIT_ASSERT*, 128

CL_BATCH_EVENT, 131
CL_BUS_ABSTRACT_MAIN_SCREEN, 267
CL_BUS_ABSTRACT_SCREEN, 270, 301
CL_BUS_ABSTRACT_SUB_SCREEN, 285, 303
CL_BUS_LOCATOR_SEARCH, 382
CL_BUS_LOCATOR_SEARCH_SCREEN, 382
CL_BUS_MESSAGE, 277, 279
CL_BUS_TAB, 287
CL_BUS_TABSTRIP, 287, 293
CL_BUS_TABSTRIP_TAB, 293
CL_OS_CA_COMMON, 82
CL_OS_SYSTEM, 101
CL_SFW_EVT_EVENT, 173
CL_SYSTEM_TRANSACTION_STATE, 79
Diagram, 162
Final, 115
Model, 54
Persistent, 81, 82, 85, 87, 88
 Coaching, 460, 466
 Code inspection, 470
 Automatic, 471
 Code Inspector, 128, 156, 470, 471, 480
 Enhancement, 472
 Message suppressing, 480
 Cohesion, 114, 194
 COMMIT WORK, 67, 80, 81, 85, 94, 98, 101, 107, 131, 401, 402, 427
 COMMUNICATION RECEIVE, 427
 Compatibility problem, 157, 159
 Component, 307
 To be embedded, 325
 Componentization, 309
 Constant, 115, 200, 203
 Evaluation, 201
 Constructor
 Abstract, 127
 Container operation, 217, 218
 Control
 Flow, 119
 Loop, 427
 Parameter, 197
 Program, 124

Cross-sectional function, 168
 CRUD interface, 191
 Customer enhancement, 168
 Customizing, 26, 29, 34, 129, 163, 166, 196, 197, 259, 443
 Access layers, 198
 Check function, 27
 Customer, 197
 Developer, 197
 Initial, 197
 Table, 165
 Technical, 198

D

Data
 Archiving, 74
 Cluster, 108, 413, 414
 Complex, 412
 Exchange process, 37
 Model, 54, 56, 58, 59
 Model, modifications, 75
 Modeling, 61
 Origin, 45
 Quality, 110, 111
 Semi-structured, 107
 Tabular format, 326
 Unstructured, 107
 Data area
 Retractable, 237
 Database
 Access layer, 76
 Buffering, 71
 Index, 71, 72
 Level, 56
 Persistence, 54
 Trace, 457
 Data Modeler, 56, 57, 192, 453, 484
 Deadlock, 68
 Debugger, 451, 452
 Layer debugging, 452
 Update debugging, 367
 Decorator pattern, 191
 Default interface, 156
 Virtual, 156
 Deliverability, 31

- Denormalization, 60, 61, 93
 - Forms*, 60
 - Dependency
 - Cyclical*, 143
 - Derivation, 51
 - Deserialization, 125
 - Design
 - Application system*, 23
 - Object-oriented*, 54
 - Design pattern, 191
 - Developer
 - Requirements*, 43
 - Development
 - Class*, 138
 - Coordination*, 43
 - Environment*, 137
 - Guideline*, 469
 - Test-driven*, 128
 - Dialog
 - Modal*, 267
 - Standard*, 230
 - Work processes*, 436
 - Dialog text, 410
 - Maintenance*, 408
 - Direct access, 74
 - Dirty read, 67
 - Dispute case, 40
 - DOCTYPE switch, 328
 - Documentation, 138, 480, 483, 484
 - Mandatory components*, 484
 - Document maintenance, 408
 - Domain-Specific Language (DSL), 46
 - DTD validation, 109
 - Duplicate, 188
- E**
-
- Easy Access Menu, 453
 - EDI procedure, 40
 - Editor
 - Graphical*, 408
 - Text-based*, 408
 - Encapsulation, 168, 185, 263
 - Enhancement, 167
 - Spot*, 168
 - Enqueue module, 69
 - Enqueue server, 67
 - Entity relationship
 - Diagram Model, structured*, 57
 - Entity type, 58
 - Environment analysis, 458
 - Environment information, 403, 404
 - Ergonomics, 230, 231, 232
 - Error
 - Potential*, 179
 - Process*, 194
 - Situation*, 116
 - System error*, 117
 - Temporary*, 117, 174, 178
 - Type*, 178
 - Unexpected*, 178
 - Error messages
 - Collecting, issuing*, 277
 - Evaluation report, 259
 - Event, 116, 170, 226, 314
 - Block*, 306
 - Handler*, 276, 277
 - Linkage*, 171, 172, 173, 226
 - Linkage, asynchronous*, 213
 - Trace*, 175, 176
 - Type linkage*, 226
 - Event queue, 175, 226
 - Browser*, 176, 177
 - Exception, 113, 116, 117, 121, 177, 426
 - Checked*, 118
 - Classic*, 117
 - Design*, 121
 - Dynamically checked*, 119
 - Handling*, 173
 - Hierarchy*, 122
 - Interface*, 116
 - Linked*, 122
 - Object-oriented*, 118
 - Unchecked*, 119
 - Exception class, 118
 - CX_BO_ERROR*, 174
 - CX_BO_TEMPORARY*, 174
 - CX_DYNAMIC_CHECK*, 119
 - CX_NO_CHECK*, 119
 - CX_STATIC_CHECK*, 118
 - CX_SWF_EVT_INVALID_EVENT*, 173
 - CX_SWF_EVT_INVALID_OBJTYPE*, 173

CX_SY_NO_HANDLER, 119
PREVIOUS parameter, 122
EXIT FROM STEP-LOOP, 427
EXPORT, 414
EXPORT TO DATA BUFFER, 108
extended Computer Aided Test Tool (eCATT), 482
Extending
 Check variant, 473
Extensibility, 26
Extension
 Modification-free, 93
External key, 65
Extract Transform Load (ETL), 33

F

Field grouping, 45
Field modification, 340
Fixture method, 128
Flexibility, 26
Flow logic, 269
Forum, 449
Framework, 40, 43, 133
Function code, 274, 275
Function group, 115
Function module, 130, 166, 206, 442
 BAL_DB_SAVE, 402
 BAL_DSP_LOG_DISPLAY, 415
 BAL_DSP_PROFILE_DETLEVEL_GET, 406
 BAL_DSP_PROFILE_NO_TREE_GET, 406
 BAL_DSP_PROFILE_POPUP_GET, 406
 BAL_DSP_PROFILE_STANDARD_GET, 406
 BAL_LOG_CREATE, 396
 BAL_LOG_MSG_ADD, 396
 BAPI_TRANSACTION_COMMIT, 99
 BAPI_TRANSACTION_ROLLBACK, 94, 99
 BP_CALCULATE_NEXT_JOB_STARTS, 438
 BP_JOBVARIANT_OVERVIEW, 131, 437

BP_JOBVARIANT_SCHEDULE, 131, 437
BP_START_DATE_EDITOR, 438
BUPA_DIALOG_SEARCH, 383
BUP_BUPA_BUT000_GET, 353
BUS_MESSAGE_STORE, 353
BUS_PARAMETERS_ISSTA_GET, 353
BUS_SEARCH_GENERATOR_MAIN, 47
Call, 425
DB_COMMIT, 96
DISPLAY_XML_STRING, 109
FIMA_DECIMAL_MONTHS_AND_YEARS, 157
FLUSH_ENQUEUE, 69
FREE_SELECTIONS_DIALOG, 206
GET_PRINT_PARAMETERS, 437
GUID_CREATE, 65
JOB_OPEN, 437
JOB_SUBMIT, 438
NUMBER_CHECK, 64
NUMBER_GET_NEXT, 62
OWN_LOGICAL_SYSTEM_GET, 33
SAP_WAPI_START_WORKFLOW, 226
SPBT_DO_NOT_USE_SERVER, 426
SPBT_GET_PP_DESTINATION, 426
SPBT_INITIALIZE, 425
TR_OBJECTS_CHECK, 258
TR_OBJECTS_INSERT, 102, 258
TR_SYS_PARAMS, 36
VIEWCLUSTER_MAINTENANCE_CALL, 258
VIEW_MAINTENANCE_SINGLE_ENTRY, 249

G

GENERATE REPORT, 47
GET BADI, 168
Global Unique Identifier (GUID), 65, 103, 422
Graphical User Interface (GUI)
 Interface, 37
 Programming, 204, 229
 Status, 266
 Technology, 46, 327, 328

I

Icon, 235, 409
 ID, 278
 Implementation Guide (IMG), 29
 IMPORT, 414
 Include, 115
 Multiple use, 244
 Program, 244, 255
 Index, 72
 Information hiding, 114, 119, 150
 Inheritance, 66, 85, 114, 126
 Hierarchy, 87
 INSERT FOR UPDATE, 67
 INSERT REPORT, 47
 Integration test, 482
 Integrity
 Referential, 65
 Interface, 37, 113, 115, 123, 137, 162, 165, 167, 169
 BI_EVENT_HANDLER_STATIC, 171, 172, 174, 191
 BI_OBJECT, 213, 215
 BI_PERSISTENT, 214, 215
 Design, 182
 Event-based, 170, 192
 External, 142
 IFARCH21, 192
 IF_BADI_INTERFACE, 169
 IF_OF_CHECK, 83
 IF_OS_CA_INSTANCE, 82
 IF_OS_CA_PERSISTENCY, 81, 82, 87, 92
 IF_OS_CHECK, 90, 91, 107
 IF_OS_FACTORY, 82
 IF_OS_STATE, 83, 89
 IF_OS_TRANSACTION, 85, 100
 IF_SERIALIZABLE_OBJECT, 203
 IF_WORKFLOW, 172, 213
 Public, 138
 Publish & Subscribe, 170
 Status, 233
 Type, 156
 Violation, 152
 iXML Library (XML Library), 108

J

Job scheduling, 437

L

Layer, 144
 Debugging, 452
 Model, 39, 144, 181
 List
 Interactive, 131
 Load distribution, 438
 Locator, 233
 Locator Framework, 267
 Lock, 67, 68
 Management, 105
 Module, 70
 Object, 68, 69, 178
 Object, create, 68
 Remove, 71
 Lock concept
 Exclusive but not cumulative lock, 69
 Exclusive lock, 69
 Optimistic lock, 69
 Shared locks, 69
 Log, 391, 393
 Archive, 417
 Create and display, 397
 Default profile templates, 406
 Delete, 417
 Display, 406
 Enrich, 403
 Handle, 396
 Header, 412
 Object, 396
 Persistent, 391
 Recipient, 393
 Research, 392, 393
 Save, 399, 400
 Sub-object, 396
 Logging, 403
 Logical Unit of Work (LUW), 62, 81, 95, 96, 131
 LOGPOINT, 215

M

Main package, 141, 147, 149, 150
 Main screen, 267
 Maintainability, 139
 Maintenance
 Dialog, 241
 View, 239, 241, 250, 253, 258
 Management service, 75
 Mapping, 80
 Mass data
 Ability to process, 93
 Capability, 27
 Master data tables, 197
 Memory
 Bound, 483
 Lack, 483
 Referenced, 483
 Memory Inspector, 213, 482
 Menu Painter, 233
 Menu standard, 233
 Message
 Class, 95
 Handling, 263
 MESSAGE, 427
 Metadata, 44
 Method, 51
 Asynchronous, 221
 Migration, 73, 78, 102, 142, 143
 Mini SAP system, 444
 Mock object, 130
 Model View Controller (MVC), 308
 MODIFY FOR UPDATE, 67
 Modularization, 26, 77, 114
 Unit, 130
 Module
 Memory, 115
 Test, 128
 Monitoring, 423
 Tool, 175
 Multitenancy, 27, 198

N

Namespace, 149, 159, 160

Naming conflict, 160
 Naming convention, 159, 160, 469, 470
 Normalization, 59, 60
 NULL value, 72, 73
 Number assignment
 External, 62
 Number range, 44, 61
 Buffering, 62
 External interval, 64
 Fiscal year, 64
 Groups, 64
 Interval, 63, 336
 Interval, create, 64
 Interval, disjunctive, 103
 Interval, statuses, 62
 Management, 44
 Object, 61, 63
 Object, create, 63

O

Object
 Identity, 51
 Lifecycle, 193
 Management, 141
 Model, 54
 Orientation, 51, 122, 166
 Persistence, 75, 141
 Persistent, 85, 104, 190
 Selection, 233
 Service, 192
 Object search service, 35
 Object Services, 78, 80, 101
 Checking agent, 83
 Class agent, 82, 87, 91, 92, 93
 Inheritance types, 86
 Mapping, 80, 104
 Object initialization, 89
 Query service, 87, 88, 92
 Transaction concept, 100
 Transaction service, 100
 Transient attributes, 84
 Type identifier, 86
 OK field, 274
 OO event, 275

Open-closed principle, 124, 127
 Open source project, 461
 Organizational management, 210

P

Package, 143, 147, 149, 150, 161, 181, 183

Concept, 135, 138, 140, 148, 485
Cyclical dependency, 143
Cyclical usage, 181
Declarative, 146
Hierarchy, 150
Interface, 453
Main package, 149
Name, 149
Splitting, 180
Strong encapsulation, 185
Structure, 138, 142, 147
Structure package, 149

Package Builder, 151

Package check, 153, 183

Client, 152
R3ENTERPRISE check mode, 156, 157
RESTRICTED check mode, 152, 157, 184
Server, 152, 184

Package interface, 149, 150, 151

Extending visibility, 153
Visibility, 153, 154

Packaging, 420, 422

Criteria, 423

Parallelizability, 419, 420

Parallelization, 391, 417, 439

Background jobs, 437

Parallel processing, 132, 444

Framework, 445

Performance optimization, 28, 419

PERFORM ON COMMIT, 79, 99, 107

Persistence

Layer, 93
Mapping, 86
Mechanism, 80, 88
Service, 93

Plug-in, 125

Polymorphism, 163

Polymorphy, 85, 87, 104, 114

Primary key, 61, 69, 102, 103

Principle of information hiding, 138, 139, 148, 177

Print list, 131

Print process, 193

Product design, 145

Product family, 39

Product idea, 23, 136

Programming

Generative, 46, 47
Object-oriented, 113
Principles, 114

Prototyping, 468

Pseudo comment, 480

Publish & Subscribe interface, 36, 162, 170, 183, 192, 213, 221

Q

Quality management, 467

Query service, 81, 92

R

REJECT, 427

Release

Change, 468
External, 49
Planning, 139, 140

Reload operation, 315

Remote Function Call (RFC), 76, 94, 95

Asynchronous, 417, 424, 435, 444

Server group, 425, 438

Server group, development guideline, 470

Renaming development objects, 139

Report, 130, 442

BDT_ANALYZER, 366

For reuse, 442

RS_PACKAGE_TREE, 152, 156

RSVIMT_NON_UC_VIM_AREAS, 258

SBAL_ARCHIVE, 417

- SBAL_ARCHIVE_DELETE*, 417
 - Variant*, 132
 - Repository Information System, 458
 - Requirement, 25
 - Analysis*, 24
 - Functional*, 25
 - Non-functional*, 26
 - Requirements analysis*, 24
 - Requirements management*, 41
 - Responsibility, 143, 193
 - Return code, 426
 - Reuse Library, 450
 - Risk management, 467
 - Robustness, 77
 - ROLLBACK WORK, 81, 94, 402, 403, 427
 - Runtime
 - Analysis*, 454, 455
 - Configuration*, 162
 - Error*, 427
 - Run Time Type Information (RTTI), 200, 203
- S**
-
- Safety facade, 117
 - SAP_ABA, 49, 146, 154
 - SAP_BASIS, 49, 146, 154
 - SAP Business Partner, 55, 56, 57, 233, 260, 331, 332
 - Basic Customizing*, 335
 - Business partner relationships*, 333
 - Business partner role*, 332
 - Business partner views*, 345
 - Define Business partner roles*, 345
 - Role*, 347
 - Role category*, 346
 - SAP Business Workflow, 261
 - SAP Business Workplace, 210, 224
 - SAP Community Day, 449
 - SAP Design Guild, 451
 - SAP Developer Network (SDN), 448
 - Subscription Program*, 450
 - SAP GUI for HTML, 209
 - SAP Help Portal, 447, 448
 - SAP Library, 447, 448
 - SAPlink, 47, 124, 125
 - SAP Locator, 369
 - Search applications*, 370
 - Search categories*, 372
 - Search help hierarchies*, 374
 - Search IDs*, 372, 386
 - Search screen*, 377
 - SAP LUW, 401, 403
 - SAP NetWeaver Business Intelligence (SAP NetWeaver BI), 41
 - SAP Note, 49, 448, 449
 - SAP R/3 icon, 237
 - SAP R/3 Style Guide, 231
 - SAP Records Management, 108
 - SAP release note, 50, 448
 - SAPscript, 407
 - SAP Service Marketplace, 447
 - SAP software component, 154
 - SAP standard, 137, 147, 159
 - SAP Support Portal, 448
 - SAP Web Flow Engine, 209
 - Saving, 107
 - Complex data*, 412
 - Mode*, 403
 - Scalability, 27, 419
 - Scaling
 - Linear*, 418
 - SCI message, 480
 - Screen, 263
 - BAdI*, 262
 - Call*, 271
 - Class*, 286, 303, 304, 305
 - Field*, 286
 - Layout*, 233
 - Logic*, 273
 - Painter*, 265, 286, 294
 - Programming*, 229
 - Programming, classic*, 229
 - Programming, object-oriented*, 261
 - Structural logic*, 264
 - Script engine, 453
 - Search criteria
 - Complex*, 79
 - Returning*, 208
 - Search function, 164

- Search help, 199, 234
 - Elementary*, 199, 374
 - Search service, 35, 75, 79, 80, 93, 161, 162, 163, 164, 165, 191, 204, 206
 - SELECT FOR UPDATE, 67
 - Selection
 - Criterion*, 294
 - Dynamic*, 204
 - Screen*, 294, 295, 296, 299, 300, 301
 - SELECT-OPTIONS, 294
 - Separating responsibilities, 41
 - Serialization, 125
 - Service, 168
 - Service class, 194, 195, 196
 - SET HANDLER, 277
 - SET PF-STATUS, 300
 - SET UPDATE TASK LOCAL, 94, 99
 - Shared
 - Buffer*, 414
 - Memory*, 414
 - Object*, 79
 - Simple transformation, 108, 110
 - Single-step task, 190, 220
 - Software
 - Architecture*, 180
 - Quality*, 467
 - Structuring*, 135, 137, 145
 - Technology*, 136, 137
 - Test*, 482
 - Software component, 154, 162
 - BBPCRM*, 146, 158
 - SAP_HR*, 146
 - Sorting, 406
 - Specification, 25, 37, 38
 - SQL trace, 458
 - Standard SAP system, 49
 - Start report, 442
 - STOP, 427
 - Structured Entity Relationship diagram (SERM), 57
 - Structure package, 149, 154, 155
 - Check*, 156
 - Style guide, 451
 - Sub-application, 145
 - SUBMIT, 427
 - Subscreen, 261, 262, 264, 267, 285, 286, 289, 296, 298
 - Area*, 283
 - Class*, 286
 - Subsequent process, 76
 - Subset, 252, 253
 - Substitution principle, 126, 127
 - Supply function, 326
 - System error, 117, 174, 178
 - Neutralization*, 177
 - System specification, 37
 - System test
 - Automatic*, 34
- ## T
-
- T100 message, 95
 - Table
 - Control*, 283, 369
 - Dependent*, 88
 - Table maintenance, 239, 241, 242, 243, 249
 - Dialog*, 238, 239, 240, 249
 - Table view element, 326
 - Tabstrip, 235, 236, 287, 289, 324
 - Tag interface, 169, 203
 - Test
 - Class*, 129
 - Customizing*, 129
 - Method*, 128
 - Top include, 115
 - Transaction
 - Control*, 98, 102
 - Mechanisms*, 76
 - Security*, 97
 - Transaction concept, 95, 401
 - Classic*, 98
 - Object-oriented*, 99
 - Transport system, 241, 258
 - Type identifier, 86
- ## U
-
- UML
 - Component diagram*, 142
 - Unique constraint, 61

- Unit test, 127
- Universal worklist, 210
- Update, 98
 - Module*, 92
 - Task*, 402
- UPDATE FOR UPDATE, 67
- Usage
 - Cyclical*, 143, 181
- Use access, 149, 155
- User guide, 233
- User interface, 230
 - Graphical*, 229
- Utility class
 - Global*, 104

V

- Verification, 35, 36
- View
 - Context*, 326
 - Embedding*, 321
- View cluster, 238, 249, 250, 251, 254, 255, 256, 258
 - Call*, 257
 - Event maintenance*, 254
 - Functions*, 255

W

- WAIT UNTIL, 427
- WAIT UP TO, 427
- Web Dynpro, 229, 306
 - Application*, 310
 - Code Wizard*, 307, 315
 - Component controller*, 311
 - Component-interface*, 311
 - Dynamic programming*, 317
 - Explorer*, 311
 - Interface view*, 311
 - Naming convention*, 470
 - Programming model*, 308
 - Window*, 311
- Web Dynpro ABAP, 48
 - Framework*, 48

- Web service, 45, 141, 189
- Wf-XML, 210
- WHERE condition, 208
- Where-used list, 173, 399, 451
- Wiki, 461
- Workflow, 33, 37, 87, 103, 126, 142, 189, 193, 209, 212, 213
 - Asynchronous methods*, 221
 - Container*, 216, 217
 - Container operation*, 217
 - Customizing*, 34
 - Event*, 170, 173, 174
 - Event linkage*, 175
 - General task*, 216
 - Log*, 225
 - Multistep task*, 222
 - Receiver type*, 171
 - Resubmission*, 211
 - Single-step task*, 190, 210, 220
 - Start date*, 220
 - Substitute rules*, 210
 - Techniques, object-oriented*, 213
 - Template*, 216
 - Terminated events, single-step task*, 222
 - Test*, 223, 224
 - Work item preview*, 210
- Workflow Builder, 218, 222
- Working area, 105
- Work item, 224
- Wrapper, 191

X

- XML, 107, 125
 - Archiving*, 74
 - Clobbering*, 108
 - Library*, 108
 - Shredding*, 109
- XSL Transformation, 48

Z

- ZIP compression, 109