Georg Lausen | Gottfried Vossen

Models and Languages of Object-Oriented Databases

ADDISON-WESLEY

Models and Languages of Object-Oriented Databases

Models and Languages of Object-Oriented Databases

Georg Lausen
University of Freiburg

Gottfried Vossen
University of Münster

ADDISON-WESLEY

Harlow, England • Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • San Juan • Milan • Mexico City • Seoul • Taipei

Addison Wesley Longman Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the World.

Cover designed by Senate
and printed by The Riverside Printing Co. (Reading) Ltd
Illustrations by Margaret Macknelly Design
Typeset in 10/12 pt Times by 42
Printed and bound in Great Britain
by Biddles Ltd, Guildford and King's Lynn

First printed 1997

ISBN 0-201-62431-1

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloging-in-Publication Data
Lausen, Georg.
[Modelle und objekt-orientierter Datenbanken. English]
Models and languages of object-oriented databases/Georg Lausen, Gottfried Vossen.
p. cm. -- (International computer science series)
Includes bibliographical references and index.
ISBN 0-201-62431-1 (alk. paper)
1. Object-oriented databases. I. Vossen, Gottfried. II. Title.
III. Series.
QA76.9. D3L3913 1997
005.75'7--dc21 97-22149
CIP

Preface

Object-oriented databases have been a subject of research for approximately 15 years and available for commercial use for about seven years. Typically, today's information technology disciplines focus on the paradigm of object orientation. The introduction of object orientation in databases relieves users of many of the restrictions implied by existing technologies, particularly the relational technology, as may be illustrated by the following two examples:

- Because data and its relationships no longer need to be represented in table format, more appropriate modelling and processing techniques have emerged for many applications.
- From the point of view of software engineering and application development there is a discontinuity if development is based on object-oriented methods but the underlying database system is relational. Using an object-oriented database system would allow data management to be integrated smoothly into the software engineering methodology.

Furthermore, in our age of global networks, distributed information processing based on object orientation becomes more and more important, particularly aspects such as interoperability, cooperation, collaboration and integration. It is generally recognized that distributed object management is a promising approach and that the organization of data management should therefore also be object-oriented.

In this book we adopt the view that the use of object-oriented databases is principally justified by their adequacy in terms of modelling and processing. We mainly deal with the aspects of models and programming languages for object-oriented databases that are relevant to the user. Aspects concerning system implementation will be considered only if they are essential to the reader's understanding of the subject matter.

A prominent feature of this book is the emphasis on SQL and the current standardization endeavours in the field of object-oriented databases. For many years the relational query language SQL has been the standard for databases and it is likely that further developments will be also based on this language. This book emphasizes SQL because it deals with object-oriented *databases* and not with persistent programming languages. Moreover, the importance of standards is presently undisputed; however, in the database field in particular no definite direction for the future has yet emerged for defining SQL3 or the outcome of ODMG (Object Database Management Group). For this reason we discuss the subject of standards somewhat more extensively and, in view of ODMG, we also deal with the more stable background of OMG (Object Management Group).

We also include a concise description of *models* and *languages* for object-oriented databases. To do so we adopt a parallel structure: on the one hand we discuss the core aspects from an informal point of view, examining concrete examples as well as possible applications; and on the other, from a formal point of view, we precisely state and explain their purposes and interrelationships. In contrast to other books we deliberately abstain from covering the whole field, but opt instead for a selection – albeit subjective – of topics. We do believe, however, that none of the essential aspects of models and programming languages has been neglected.

This book is divided into three parts. In Part I, Chapter 1 we provide an introduction to databases, describing the evolution of data models and database systems, and also the properties of object orientation in general and object-oriented databases in particular. In Chapters 2 and 3 we define the framework of the subsequent discussion in greater detail. Chapter 2 presents the principal features of object-oriented database languages, but does not yet relate them to a specific language or product; in Chapter 3 we examine more closely the model properties of object-oriented databases and present a formal framework which accounts for problems discussed in the previous sections.

In Part II, Chapter 4 various case studies are conducted in order to examine possible ways of realizing object-oriented database systems. We do not claim completeness in our selection of systems, but, in our opinion, we cover representatives of all current major developments: that is, object-oriented systems in the tradition of databases, respectively programming languages, as well as object-relational systems. Chapter 5 is concerned with current standardization projects; we are convinced that SQL3 and also OMG and ODMG will be a basis for further standardization in the field of object-oriented database languages.

For those readers who are interested in the more formal aspects of the subject, we discuss some theoretical concepts in Part III. Algebraic approaches to language design are discussed in Chapter 6 and contrasted with rule-based concepts in Chapter 7.

The current book is an extended and corrected version of *Objekt-orientierte Datenbanken: Modelle und Sprachen*, a book written in German and published by Oldenbourg Verlag in 1996. This book in turn was based on a tutorial which we held for the Deutsche Informatik-Akademie at various locations in 1993; in addition we have incorporated material that we teach to our students at the university. Accordingly, this book mainly targets readers who are familiar with the basic concepts of databases and wish to specialize in object-oriented databases, in particular third- or fourth-year undergraduates as well as first- or second-year graduate students of computer science, or programmers and information system designers who are familiar with databases or object-oriented programming languages.

Thanks go to Monika Rengers for doing most of the figures, to Petra Weiermann for helping with the editing of the manuscript and, last but not least, to our families for their patience.

Georg Lausen, Gottfried Vossen
Freiburg and Münster, February 1997

Contents

Part I

Foundation

1 Object orientation in databases

This introductory chapter motivates object orientation in databases and database systems to give a framework for the subsequent sections. First we discuss databases and their typical application areas in order to understand the requirements for database systems. The historical development of data models and database systems shows to what extent the different system generations fulfil these requirements. Today, *relational* databases are of particular importance, but they have become a limiting factor, especially in new database applications. This in turn motivates the use of object-oriented concepts in databases, leading to so-called *object-relational database systems* as well as the development of independent *object-oriented database systems*. We offer a description of these attempts and discuss the problems and objectives which must be solved in order to obtain an adequate integration of object orientation and databases. We shall mostly concentrate on object-oriented database systems because, as far as modelling and language aspects are concerned, object-relational ones can be considered as a restricted version. In the subsequent chapters we demonstrate the state of the art of the technology and outline how it is likely to progress.

1.1 Introduction

Database management takes a central role in applications where large persistent collections of data are to be organized and maintained, and which are therefore supported by a computer-based information system. Databases have been researched, developed and employed for about 30 years in applications of very different nature and objectives.

Roughly speaking, a *database system* (DBS) is based on the idea that data and associated programs are *separated*, and thus is very different from a typical file system. However, seeming at first glance to be a contradiction, object-oriented databases aim to achieve *integration* of data and programs, but this is now achieved in an adequate methodical framework, which borrows much from abstract data types.

A database system consists of software, the *database management system* (DBMS) and one or several *databases*. The DBMS is a program system which runs in the main memory of a computer and is controlled by the respective operating system. A database is a collection of data which represents information concerning a certain real-world application; because of its size, it is usually held in secondary storage. The DBMS functions as an interface between the users and the database; it ensures that users have adequate and efficient access to the data, and that the data itself is resistant to hardware and software failures and can be persistently stored over long periods of time independent of the programs that access it.

Technically speaking, a DBMS is embedded in the software context of a general computer system as shown in Figure 1.1. It is connected to the outside world by

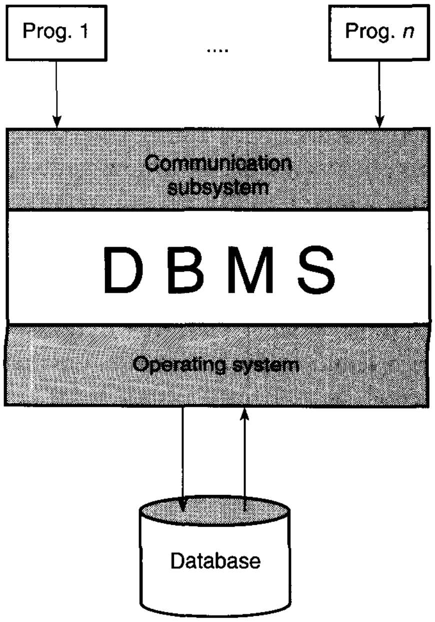


Figure 1.1 Generic view of a DBMS.

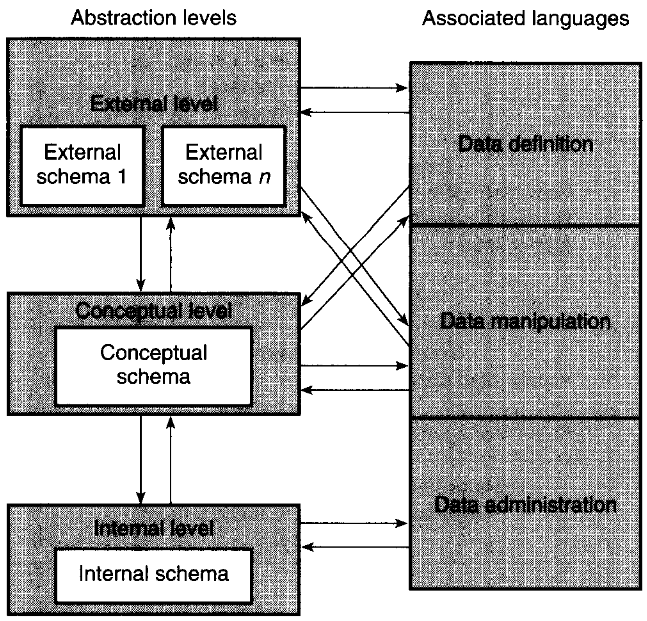


Figure 1.2 Three-level database architecture according to ANSI/SPARC.

the communication subsystem. The distinction between a database and the DBMS by which it is managed admits at least two possible ways of viewing a DBS:

- from the user's or
- from the developer's point of view.

From a user's point of view, a database is seen at different abstraction levels where different kinds of data can be identified. The generally accepted ANSI/SPARC model, for example, has the three abstraction levels shown in Figure 1.2, which are designed to achieve logical and physical *data independence*. The (lowest) internal level is concerned with the physical definition and organization of data (in the form of adequate data structures with associated access paths). The conceptual level, the next level up, describes the time-invariant general structure of a database in the conceptual *schema* and in the language of a specific *data model* which abstracts from the details of the physical level. The (highest) external level offers individual users or user groups the possibility of defining parts of the conceptual schema for their particular applications in the form of an external schema: a *view*.

In this architecture, the central aspect of a database is the conceptual level and its data model. Usually, a data model has a *specification component* to define the structural aspects of an application and its semantics, and an *operational component* which allows the manipulation of these structures. As a consequence, the user is provided with specific languages with which he or she can define, manipulate and manage database structures. Figure 1.2 shows these languages or language parts and the respective levels to which they belong.

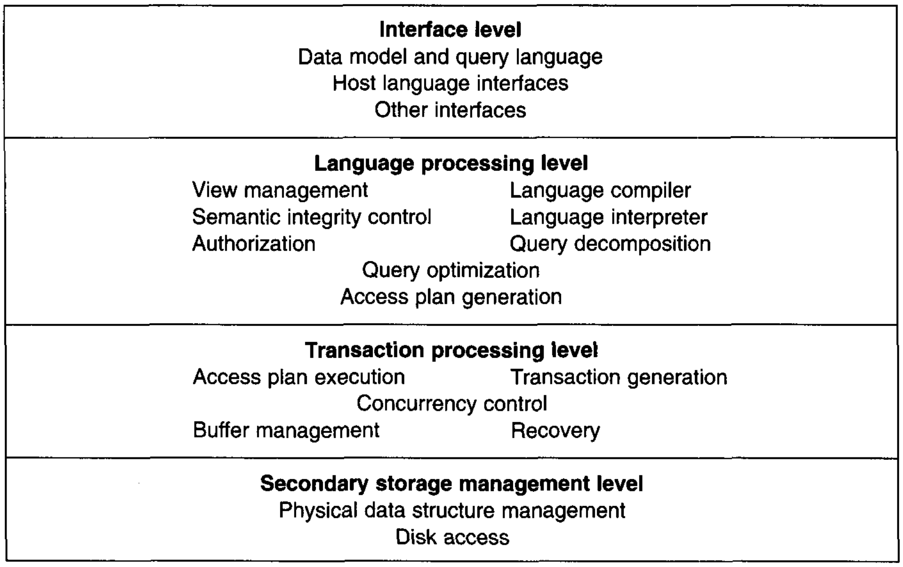


Figure 1.3 Functional DBMS levels.

From a system developer's viewpoint, a DBMS has to cover many functional aspects, which in part stem directly from the user's view. In principle, the functionality of a DBMS has to be regarded as a four-level structure (compare Figure 1.3). The interface level makes interfaces available to the various classes of user, including the database administrator, ad hoc user and application programmer. The language processing level is responsible for dealing with the various kinds of task to be performed by a database (such as queries and updates). A query, for example, is typically decomposed into a series of elementary database operations, which then undergo *optimization* in order to avoid unacceptably long execution times. An executable query or executable program is transferred to the transaction processing level, which controls concurrent access to a database which may be shared by many users (*concurrency control*) and simultaneously makes the system resistant to certain kinds of error (*recovery*). Finally, the secondary storage management level is responsible for physical data structures (for example, files, pages and indices) and disk accesses.

Based on this general description of the two principal ways of viewing a database, four major areas can be identified on which database research and development have concentrated for the past 25 years.

- *Data models*: Data models provide appropriate concepts for the abstraction of real-world applications and the modelling of data structures and semantics in connection with them. Up to the present day, data models have undergone a remarkable evolution.
- *Database languages*: The topic here is the provision of languages which make it possible to access and operate on databases in a suitable manner.

It is essential to understand that database languages, compared with general programming languages, can have only a limited expressive power because, from a practical viewpoint, efficiency is of major importance. Languages like the relational standard SQL (see Section 1.3), for example, have no loop construct. But, if necessary, the query language of a DBS can be *embedded* in a general programming language which is always equipped with control structures and thus ensures full computational power. What then generally happens is that the so-called *impedance mismatch* is encountered, since databases and programming languages have different data types. The data type *set*, for example, is typically not available in programming languages; at the programming language level, set operations on a database must be decomposed into tuple operations on the elements of the set by using a *cursor*. The abstraction level of the database language is lost at this point, and this loss is then quite often responsible for awkward and difficult-to-maintain programs.

- *Transactions and concurrency control*: Databases allow concurrent access to shared data and therefore must offer appropriate synchronization and restart strategies. For this purpose, the transaction concept has been developed, which provides the basis for various implementation possibilities according to the specific application demands.
- *Data structures*: An integral component of the database system is a secondary storage management facility. Here the development of appropriate data structures and access techniques is essential. Nowadays there exists a broad knowledge of efficient techniques for data storage and retrieval.

The development and evaluation of these aspects has always been led or influenced by the requirements of the various database system application areas. Commercial applications, especially in business and administration, were of particular significance. We mention the following examples:

- banks and insurance companies
- public administration
- libraries
- booking and reservation systems (START and AMADEUS)
- materials management and control, stock control
- personnel management
- order entry and accounting
- residents registration system and information services
- production planning and management

These areas share numerous characteristics which make database technology an efficient tool; for example:

- simple data sets of fixed format, which can be adequately described at a logical level, particularly by employing record-oriented data models;

- simple data types for the values of the fields or attributes describing a data set (numerical as well as alphanumeric types are often sufficient);
- parametrizable, predefinable queries which describe highly repetitive tasks to be executed by the database;
- in-place updates where old values are replaced by new ones without retaining the corresponding old value;
- short transactions which represent tasks to be performed by the database (queries or update operations) in executable form; these generally have to be processed at a high rate (for example, 1000 transactions per second).

For the same reasons that motivated the transition from file systems to database systems in the above-mentioned application areas over 20 years ago, for example,

- physical and logical data independence,
- redundancy-free storage,
- central integrity control,
- languages at a high level of abstraction,

today new areas demand DBS support to an increasing degree. The following list includes only a few examples:

- the CAx fields, for example CAD, CAM, CAE, CASE;
- office information systems;
- geographical information systems;
- experimental data recording and evaluation;
- multimedia systems (for images, text, language, data and video).

These areas have their own typical characteristics which are either not supported or only poorly supported by today's DBS. We must distinguish application-specific characteristics from application-independent ones; the latter include:

- the necessity to model highly structured information, which implies the ability to represent complex objects and data types;
- the desire to model behaviour: that is, object-specific or type-specific operations on the structures which can be described by the underlying data model.

Taking into account that the possibilities for defining structures which are provided by a traditional data model are often severely limited compared with those available to a programming language, it is not surprising that more recent research on data models attempts to incorporate paradigms known from programming languages. In this context, object orientation, which is the subject of this book, is of particular interest.

1.2 Historical development

The historical development of database systems can be divided into three generations. The first generation (late 1960s and 1970s) was characterized by the separation of *logical* and *physical* information. For the first time, data models were used to describe physical structures from a logical viewpoint. In particular, the hierarchical model and the network model were developed on the basis of graph concepts. We shall not discuss these models in detail, since they are of little interest today.

The second generation is characterized by *relational systems*, which became commercially available in the early 1980s. These systems are based on a new and simpler approach to data organization, the *relation* or *table*, which allows a considerably clearer distinction between a logical and a physical data model. Relational systems offer a high degree of physical data independence and include powerful languages, even though they have limited expressive power. Physical data independence means that the physical storage of data is transparent (in the sense of invisible) to the user and may in principle be changed without also changing the logical view of the data. Relational languages are *set-oriented* (as opposed to record-oriented) and can thus be non-procedural or *declarative*. In a relational database the user sees the data as tables. The example in Figure 1.4 shows a small database for a library application. The *Book* relation describes the books available in a library, the *Reader* relation describes the readers known to the library, and the *Lending* relation relates books to readers. Set-oriented processing means that the tables of a relational database can be manipulated in their entirety by special operators; there is no need to iterate tuple by tuple through the relation. Since relations are an important and well-known mathematical concept and may also, for example, be understood as predicates of mathematical logics, this generation of database systems was a topic of

Book	*InvNr*	*FirstAuthor*	*FurtherAuthors*	*Title*	...
	123	Date	n	Intro DBS	
	234	Jones	y	Algorithms	
	345	King	n	Operating Syst.	

Reader	*Reader no.*	*Name*	...
	224	Peter	
	347	Laura	

Lending	*Inv. no.*	*Reader no.*	*Expiry date*
	123	225	07-22-94
	234	347	08-02-94

Figure 1.4 Example of a relational database.

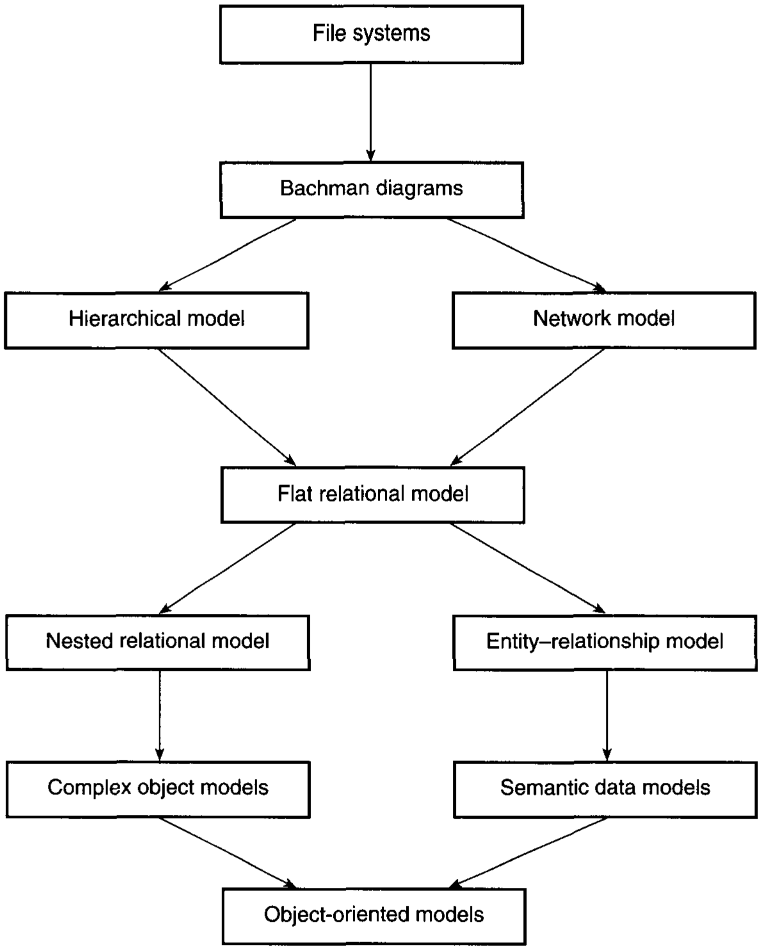


Figure 1.5 Evolution of data models.

thorough research from a theoretical viewpoint, starting in the early 1970s shortly after Codd had proposed the model.

While database systems were mainly used in business and management applications initially, it was recognized in the early 1980s that advantages could be gained from databases in scientific, technical, office and other fields as well. However, relational systems reach their limitations in applications like CAD, CASE or CIM; this is due to several reasons – discussed in more detail later – which triggered the development of new technology. Today, this development is in full swing; object-oriented, object-relational and extended-relational database systems can be considered as a third generation, which is now starting to become commercially available. This generation far exceeds the possibilities of purely relational systems and, among other things, aims at an appropriate integration of programming languages and databases.

Apart from the evolution of systems, a second closely related line of evolution can be observed which concentrates on *data models* (Figure 1.5). The motivation for the development of entity–relationship (ER) models, semantic models or complex object models was as follows: the desire to have graphic design support; decreasing interest in the record-oriented approach; establishment of a clearer distinction between logical and physical concepts; reduced semantic overloading of types of relationships by explicit introduction of specific constructs (for example, IsA type relationships, that is, *specializations*, in contrast to component relationships, or *aggregations*); and a sometimes only indirect availability of abstraction mechanisms in record-oriented models.

Whereas the nested relational model is a direct generalization of the flat relational model, the ER model and the subsequently developed semantic models use a different technique to describe given applications. The principal objective initially was to establish ways of describing objects with complex structures more adequately and of recording semantic information. This is precisely what so-called complex object models achieve; these models differ from semantic models most in the availability of type constructors. Object-oriented models which allow structure modelling as well as the modelling of object behaviour are the most recent development in the evolution of data models.

1.3 Relational databases and SQL

The systematic study of database principles began with the introduction of the relational model. Like all other data models, the relational model has a structural part (schemata with dependencies) and an operational part (relational algebra). In the relational model the logical and physical levels (according to the ANSI/SPARC model) are independent of each other. The only basic construct on the logical level is the *relation*; relations can be illustrated for the user in the form of tables (see Figure 1.4). In this section, we should like to introduce some common concepts of the relational model and the language SQL. This is to justify the study of object-oriented databases and also to take into account the likelihood that, in all probability, SQL – appropriately extended – will play an important role in object-oriented databases.

1.3.1 Structural aspects

A relational database is described at the conceptual level by a *schema* in which the structural aspects of the database are defined. A relational database schema consists of a set of relational schemata which are related to each other by interrelational constraints. A relational schema itself consists of a name, a set of attributes and their domains, and a set of intrarelational constraints; the relations corresponding to the schemata consist of tuples which contain the data values and fulfil the inter- and intrarelational constraints. A relation is usually represented in the form of a table. The terms used in SQL for relation, tuple and attribute are *table*, *row* and *column*.

The SQL command for the definition of data is `CREATE TABLE`, whereby new relational schemata (in the context of a previously declared database schema) can be defined. For each such schema, at least key definitions and entity and referential integrity can be distinguished as integrity constraints. The syntax of this command is as follows:

```
CREATE TABLE table-name
( column-name-1 type [ NOT NULL | UNIQUE ]
[, column-name-2 ... ]
   .
   .
   .
[, UNIQUE ( list-of-column-names ) [, UNIQUE ... ] ]
[, PRIMARY KEY ( list-of-column-names ) ]
[, FOREIGN KEY ( list-of-column-names )
REFERENCES table-name-2 [( list-of-column-names ) ]
[, FOREIGN KEY ... ] ]
[, CHECK ( condition ) [, CHECK ...] ] )
```

Several other options, which are not of interest here, are omitted. First, the attribute names and their type are defined; furthermore, each attribute can optionally be declared free of null values or unique (that is, all its values in a relation have to be different). The following data types are available: CHAR, VARCHAR, INT, SMALLINT, DEC, FLOAT, BIT, DATE, TIME, TIMESTAMP.

According to the above syntax, an integrity constraint in the definition of a table can be a primary key definition, a foreign key definition or a check clause; SQL also has domain and attribute conditions and so-called *assertions*. In this section we restrict ourselves to the description of keys and foreign keys.

Generally, *key* (PRIMARY KEY) means an attribute or attribute combination whose values uniquely identify the tuples of any relation of the respective schema. A *foreign key* (FOREIGN KEY) defined for a schema *R*, however, describes an attribute or attribute combination which is a key in *another* schema *S* (REFERENCES); the meaning of a foreign key relationship between *R* and *S* via attributes *X* is that the *X* part of *R* is a subset of the *X* part of *S*. A foreign key definition defines in particular an inclusion dependency between two tables. It is possible to define in SQL what is to be done if the subset condition between the respective tables is violated by a delete or update operation.

1.3.2 Updates and queries in SQL

After tables have been defined, (new) tuples may be entered using the INSERT command, which in its simplest form looks as follows:

```
INSERT INTO table-name
[ ( list-of-column-names ) ]
VALUES ( data-items )
```

The values of the respective tuple are the data items separated by commas. If new values are to be entered for certain attributes but not for all, the respective columns must be indicated explicitly; the remaining columns are filled with null values.

The following commands for deleting or updating all tuples that satisfy a certain condition are more or less self-explanatory:

```
DELETE FROM table-name
[ WHERE condition ]

UPDATE table-name
SET column-name-1 = expression-1
[, column-name-2 = expression-2 ] ...
[ WHERE condition ]
```

In SQL only one type of command can be used to query a database: the SELECT command. This command has many options to define simple or complex queries, but we discuss only a few basic ones here. The SELECT command has the following simple basic structure:

```
SELECT     attribute list        (* output *)
FROM       relations list        (* input *)
WHERE      condition
```

The SELECT clause indicates the attributes onto which the projection should be made: that is, which column names the result should have. The FROM clause indicates from which relations these attributes and their values should be taken: that is, which operands should be addressed in order to answer the query. The (optional) WHERE clause indicates selection conditions with which the result should comply, or conditions that specify which tuples from different relations are to be assembled.

Executing a SELECT command is basically done in the following three steps:

(1) Derive a Cartesian product of the operand tables indicated in the FROM clause.

(2) Based on this intermediate result, evaluate the conditions stated in the WHERE clause.

(3) Project the result from step (2) onto the attributes indicated in the SELECT clause; DISTINCT may be added to avoid duplicate tuples in the final result.

We shall illustrate the effect of the SELECT command by looking at some operations of the relational algebra:

(1) Let R be a relational schema with a set of attributes X, where $\{A_1, ..., A_k\} \subseteq X$. The *projection* of R onto $\{A_1, ..., A_k\}$ is expressed by

```
SELECT DISTINCT A1, ... , Ak FROM R
```

(2) Let R be as in (1), where $A, B \in X$. The *selection* of R with respect to condition $A = a$ is expressed by

```
SELECT DISTINCT * FROM R WHERE A = a
```

Analogously, the selection with respect to condition $A = B$ is expressed by

```
SELECT DISTINCT * FROM R WHERE A = B
```

(3) Let R and S be relational schemata with equal sets of attributes. The *union* of R and S is expressed by

```
SELECT DISTINCT * FROM R
UNION SELECT DISTINCT * FROM S
```

Analogously, the *difference* between R and S is expressed by

```
SELECT DISTINCT * FROM R
EXCEPT SELECT DISTINCT * FROM S
```

(4) Let R be a relational schema with attributes $A_1, ..., A_n, B_1, ... , B_m$ and S a relational schema with attributes $B_1, ..., B_m, C_1, ... , C_l$. Then the *natural join* of R and S, which joins those tuples from R and S which have equal B-values, is expressed by

```
SELECT DISTINCT A1, ... , Am, R.B1, ... , R.Bm, C1, ... , Cl
FROM R, S
WHERE R.B1 = S.B1 AND ... AND R.Bm = S.Bm
```

Since attributes from different relational schemata may have identical names, the dot-notation $R.B$ is used to specify which occurrence of the respective attribute is meant. With respect to the natural join, of course, we could have used $S.B$ as well. The WHERE clause then explicitly states that tuples from the different relations must have identical values with respect to the shared attributes. A simpler formulation of the natural join, which is possible in SQL2, is:

```
SELECT * FROM R NATURAL JOIN S
```

1.4 An example application

This section describes an example application to which we shall frequently refer in the following sections and chapters. It will help us to identify a series of shortcomings of the relational model; moreover, it will motivate the new modelling capabilities in database systems as they are in fact provided by object-oriented technology.

In our example application we want to establish a database for vehicle manufacturers which should contain information on *companies* which manufacture vehicles, *vehicles* and especially *automobiles* currently manufactured, *persons* owning vehicles and *employees* who have a position in a company.

First of all, we are going to make this general description precise by listing the attributes that characterize these entities:

- *companies* have a name, head office, several subsidiaries and a president;
- *subsidiaries* have a name, office, manager and employees;
- *vehicles* have a model name, colour and manufacturer;

- *automobiles* are composed of a drive and car body, with the drive consisting of an engine and gearing, and the engine being characterized by its horsepower and cubic capacity;
- *persons* have a name, age and domicile and possess a private fleet;
- *employees* are persons with specific qualifications, salary and family members.

1.4.1 Representation in the relational model

Based on the description given above, we want to represent this application at an abstract level; we realize that the relational data model offers only insufficient support, because

- *compound attributes* (for example, a person's or company's address) cannot be represented directly; the components have to be declared individually as attributes;
- *set-valued* attributes (for example, subsidiaries of a company) must be differentiated from single-valued ones and represented in a different relational schema;
- *aggregations* (like the drive of an automobile) and *specializations* (employees in comparison to persons) require individual relation schemata equipped with special integrity constraints;
- the introduction of *artificial keys* will be necessary if the attributes are not sufficient to obtain a unique identification (for example, the company name cannot serve as a key, because companies may carry the same company name in different countries).

Let us have a closer look at *companies*. The attribute *name* is unproblematic in the relational representation if, for example, the data type VARCHAR is sufficient. The attribute *head office* is to be understood as an address, consisting of *street* and *location*. Since such a structure cannot be represented in the relational model, the attribute *head office* must be eliminated and replaced by *street* and *location*. Since a company can have several subsidiaries, the attribute *subsidiaries* is set-valued. In order to avoid redundancies at the tuple level (repetition of all company details for every subsidiary), it is necessary to use a second relation in which only the subsidiaries of every company are listed. But in order to establish a relationship between the company's subsidiaries and its other details, this decomposition requires the introduction of a (in this case artificial) key; for this purpose we choose the new attribute *companyID*. Subsidiaries are themselves structured as described above; we can proceed in the same way as with the company attributes, but the set-valued attribute *employees* requires a third relational schema. The *president* of a company is an employee of that company and employees are special types of persons. This is represented in the relational manner by defining *president* as a foreign key with respect to *employee* and introducing a new relation for the specialization relationship between *employee* and *person*. Note that relation *SubsEmpl* can be regarded as an example of a relational representation of an aggregation relationship. The special semantics of an aggregation forces us to state attributes explicitly as foreign keys.

Based on these considerations, we arrive at the following representation of company information in the relational model:

```
CREATE TABLE Company
  (CompanyID INT NOT NULL,
  Name VARCHAR NOT NULL,
  Street VARCHAR NOT NULL,
  Location VARCHAR NOT NULL,
  President INT NOT NULL,
  PRIMARY KEY (CompanyID),
  FOREIGN KEY (President) REFERENCES Employee (EmplNo)
  );

CREATE TABLE Subsidiary
  (CompanyID INT NOT NULL,
  NameSubs VARCHAR NOT NULL,
  Street VARCHAR NOT NULL,
  Location VARCHAR NOT NULL,
  Manager INT NOT NULL,
  PRIMARY KEY (CompanyID, NameSubs),
  FOREIGN KEY (CompanyID) REFERENCES Company,
  FOREIGN KEY (Manager) REFERENCES Employee (EmplNo)
  );

CREATE TABLE SubsEmpl
  (CompanyID INT NOT NULL,
  NameSubs VARCHAR NOT NULL,
  Empl INT NOT NULL,
  PRIMARY KEY (CompanyID, NameSubs, Empl),
  FOREIGN KEY (CompanyID) REFERENCES Company,
  FOREIGN KEY (NameSubs) REFERENCES Subsidiary,
  FOREIGN KEY (Empl) REFERENCES Employee (EmplNo)
  );

CREATE TABLE Person
  (PersNo INT NOT NULL,
  Name VARCHAR NOT NULL,
  . . .);

CREATE TABLE Employee
  (EmplNo INT NOT NULL,
  . . .
  FOREIGN KEY (EmplNo) REFERENCES Person (PersNo));
```

It should be noted that these relations are not the only possible relational representation. But a representation in this form is essential if certain quality requirements (as given by the normal forms, for example) are to be taken into account.

As one consequence of such a schema, defining queries may become complicated. If you want to know, for example, whether an employee named Lacroix is employed in the Ghent subsidiary of the Ford company this will be represented in SQL as follows:

```
SELECT EmplNo
FROM Company, Subsidiary, SubsEmpl, Employee
WHERE Company.Name = 'Ford'
  AND Company.CompanyID = Subsidiary.CompanyID
  AND Subsidiary.Location = 'Ghent'
  AND Subsidiary.CompanyID = SubsEmpl.CompanyID
  AND Subsidiary.NameSubs = SubsEmpl. NameSubs
  AND SubsEmpl.Empl = Employee.EmplNo
  AND Employee.Name = 'Lacroix';
```

The evaluation of this query could be done as follows: first, the Ford subsidiary in Ghent is determined; the set of this subsidiary's employees is then compared with the set of all employees in order to find out whether a person named Lacroix is in the first set.

1.4.2 Non-relational representation

We shall now leave the relational world and outline how a more adequate representation of our example can be achieved if at least the following are supported by a data model:

(1) *Type declarations* are as flexible as in programming languages.

(2) Objects can be *uniquely identified* independently of their attribute values. To this end so-called *object identifiers* are provided, which allow *referencing* between objects.

(3) *Reuse* of information is possible by

- *referencing* objects from different locations so that new information can be assembled from existing information, or in other terms, new objects are built out of already existing ones, which become their *components*, and can be considered as their *subobjects*. This technique is called *aggregation*, or sometimes *association*.
- defining *specializations* between certain information units, which then allow *inheritance* of structure and behaviour.

To sketch a representation which makes use of these additional features, we now switch to a notation which is quite common for type declaration in programming languages:

Notation	*Meaning*
[.....]	tuple type
{.....}	set type

We assume the availability of the base types `String` (for strings) and `Integer` (for integer numbers). We describe companies as follows:

```
Company: [
  Name: String,
  Headoffice: Address,
  Subsidiaries: { Subsidiary },
  President: Employee ]

Subsidiary: [
  Name: String,
  Office: Address,
  Manager: Employee,
  Employees: { Employee } ]

Address: [
  Street: String,
  Location: String ]
```

First let us look how *aggregation* is applied. A Company is built out of a string (Name), an object of type Address (Headoffice), a set object with element type Subsidiary (Subsidiaries) and an object of type Employee (President). Note that we refer directly to the name of the type. This has the effect that for a concrete company the value of attribute Headoffice is a reference to the concrete object of type Address, the value of attribute Subsidiaries is a set of references to objects of type Subsidiary, and the value of attribute President is a reference to an object of type Employee. The object identifiers needed for such a referencing scheme may be implemented by unique names at a logical level or by (virtual) addresses at a storage level. Note that because we associate attributes with named types there is no longer a need for a `FOREIGN KEY` clause.

With regard to persons, we would like to distinguish those that are also employees; for employees, in addition to all the properties of persons, more properties are of interest. Each employee thus *specializes* a person, or in other terms, *is also* a person. We express this kind of relationship as follows:

```
Person: [
  Name: String
  Age: Integer,
  Domicile: Address,
  Fleet: { Vehicle } ]

Employee is-a Person: [
  Qualifications: { String },
  Salary: Integer,
  Familymembers: { Person } ]
```

Here, by means of *inheritance*, the structure of persons is reused for employees. Thus, all the attributes defined for persons – Name, Age, Domicile and Fleet – are also defined for employees in addition to their specific attributes Qualifications, Salary and Familymembers. Finally, we describe vehicle information in an analogous way:

```
Vehicle: [
  Model: String,
  Manufacturer: Company,
  Colour: String ]

Automobile is-a Vehicle: [
  Drive: VehicleDrive,
  Carbody: String ]

VehicleDrive: [
  Engine: OttoEngine,
  Gearing: String ]

OttoEngine:[
  HP: Integer,
  cc: Integer ]
```

To give an impression of how querying becomes more elegant and concise once representation has become more adequate, we show next how the relational query from the preceding section could be stated in an object-oriented adaptation of SQL:

```
select e
from e in Employee, c in Company, s in Subsidiary
where c.Name = 'Ford' and
s in c.Subsidiaries and
s.Office.Location = 'Ghent' and
e in s.Employees and
e.Name = 'Lacroix'
```

In this query we follow references by so-called *path expressions*. Note how the richer structuring possibilities in the object-oriented setting achieve conciseness (cf. also Sections 2.2 and 2.3 for more details).

1.4.3 Findings

Our example has shown the limitations of the relational model, even for data modelling in traditional applications, especially when *adequate* modelling is to be achieved. The fact that in the relational model only the two constructors *tuple* and *set* are available to structure information, where the set constructor can be used only once to define a relation as a set of tuples, is particularly problematic. We realized that referencing, aggregation and specialization help to represent information more adequately, but are not provided in the relational model. The result is unnecessarily complicated structured schemata and, as a further consequence, awkward queries.

The use of artificial keys, which are needed solely for identification purposes, implies the problem that their values have to be assigned and managed by the *user*; therefore there is no independence of key values and attribute values which is guaranteed by the system.

As a further aspect, conventional databases allow access to stored information only via the operators of the respective user language. It is not possible to bind special operators or even procedures to certain entities or information units, with which access and processing are performed. Referring to the above example: using the relational model we can operate only with the representation of relations in tables; only via an application program is it possible, for example, to show the user a map with the head office of the company marked on it. Certainly, it would be nicer to bind such a program directly to the representation of the company.

Such considerations lead from structural aspects to *behavioural* ones. *Abstract data types* give us the necessary paradigm to *encapsulate* structure *and* behaviour. It is obvious that databases should benefit from such a technique.

1.5 Object-oriented databases

The discussion in the previous section revealed problems when traditional database systems are used. We now argue that *object-oriented* database systems are better suited because they have mechanisms to avoid these problems. First, we state the requirements which must be fulfilled. As a (possible) answer we then describe the paradigm of object orientation as well as the principal properties of an object-oriented database system.

1.5.1 Requirements

Against the background of the example application described in the previous section, a suitable database system is expected to have at least the following functionality:

- The structure of objects to be modelled can be nested as required; different structures of that kind can reference each other or be introduced as specializations of others.
- Object identity is supported to allow objects to be distinguished independently of their attribute values.
- Objects can be equipped with behaviour which links specific operations exclusively to these entities.
- The structure *and* behaviour of objects can be inherited hierarchically.

Moreover, as an additional functionality, it is desirable that both the structure and the behaviour definitions be *extensible* in a logical sense, which means that even after completion of the database design new attributes and new operations can be defined and added to the existing set. Such a feature supports the evolution of a database over time.

1.5.2 The paradigm of object orientation

In general, object orientation in databases combines concepts from various areas, including at least the following:

- From *programming languages*: abstract data types and the encapsulation principle, also the computational completeness of a programming language.
- From *software technology*: code extensibility and code reusability, and the principle of modularization.
- From *artificial intelligence*: ideas and approaches to knowledge representation, techniques and methods of classification.
- From *databases*, or rather *data modelling*: nested relations or generalizations of the relational model as proposed, for example, in connection with semantic data models.

With regard to the first point it should be noted that database languages are in general *not* complete, because they do not allow computable queries to a database to be expressed. Extensions of traditional database languages may be inefficient, and when they are embedded in programming languages they suffer from a loss of abstraction. Object orientation in databases tries to overcome these problems. In particular the aim is to reduce or even eliminate the *impedance mismatch* between database languages and programming languages already mentioned in Section 1.1.

In brief, object orientation as a paradigm is based on the following five principles:

(1) Each entity of a given application is modelled as an *object* with its own *identity*, which is distinguished from the *value* of the object. As a consequence, objects may be composed of other objects and objects may be referenced from several other objects; the latter case is known as *object sharing*.

To illustrate this aspect, Figure 1.6 shows two classes, *Company* and *Employee* (the rows are to be filled with objects). One employee of the company is its *President*; if the company is a joint-stock company, there has to be a principal shareholder. The important thing is that these two persons can be identical (although the principal shareholder of a company is not necessarily its employee).

(2) Each object *encapsulates structure* and *behaviour*. The structure of an object is given by so-called *instance variables* (*attributes*) whose values may be scalar, sets, compound or references to other objects. An object's set of values constitutes the (usually time-dependent) *state* of the object. The behaviour of an object results from the *methods* which can be executed on the

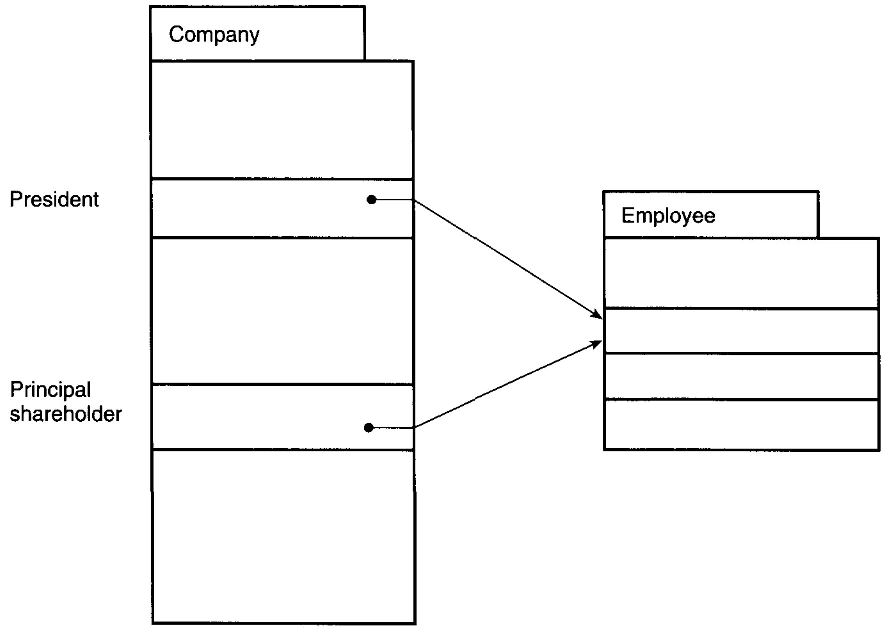


Figure 1.6 Principle of object sharing.

object. It should be noted that frequently a distinction between attributes and methods is made only at the model level, not at the language level.

(3) The state of an object can be accessed by passing *messages*; access to the state in this form may be exclusive or optional. If an object receives a message that it understands, the execution of an associated *method* is initiated. Basically, objects communicate by exchanging messages; this is also known as *message passing*.

(4) Objects with a common structure and common behaviour are grouped into *classes*; in general, each object is an *instance* of one class. Message passing between objects is usually realized based on information defined in the class. When message passing takes place, we call the class that provides the implementation of the method that responds to the message the *receiver class*.

A typical sequence during message passing is illustrated in Figure 1.7, in which an object from Class 2 sends message Mess 2 to an object from Class 1. The receiver consults its class on how to react to this message; because Class 1 knows an implementation for Mess 2, namely method Meth 2, it replies to the object with this method which is then executed by the object.

(5) Classes are arranged in a *hierarchy* which is implied by defining a class as a specialization of one or several other classes. The subordinate classes are called *subclasses* and the superordinate classes are referred to as *superclasses*. Subclasses *inherit* the structure and behaviour of their superclasses. If a subclass provides a specific structure and behaviour, this will *override*

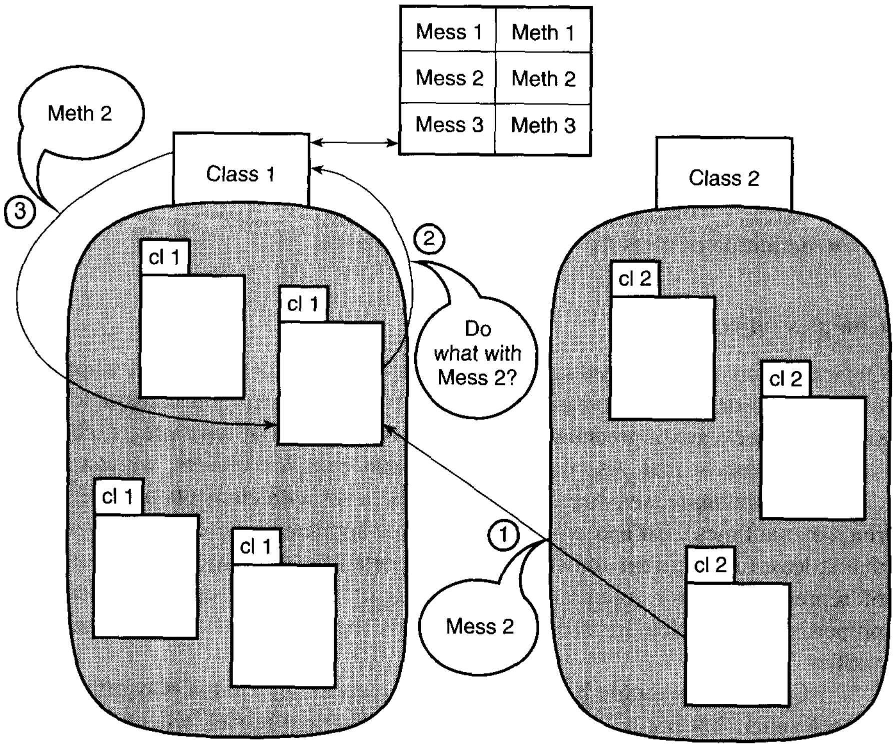


Figure 1.7 Principle of message passing.

inheritance. It should be noted that there are exceptions to the inheritance rule mentioned so far. Sometimes certain values, so-called *default values*, or simply *defaults*, may also be inherited. On the other hand, *class attributes*, that is, attributes which are not applicable to individual objects but to the set of *all* objects, and class methods might *not* be inherited. Inheritance *conflicts* may arise if *multiple inheritance* takes place: for example, if a class inherits attributes with the same name but different types from two superclasses, a conflict occurs which must be solved by the programmer or the system. Analogously, there may be conflicts between methods with the same names and different implementations.

These principles characterize the paradigm of object orientation as it is applied in different contexts, for example in programming languages, operating systems or databases. For databases especially, it is important to combine adequately the paradigm itself with the typical properties of databases. We now discuss the principal properties of object-oriented databases in more detail; these properties can be classified into *object-oriented properties* (OO properties), which cover *data model* and *language* features, and *database system properties* (DB properties), which ensure that an object-oriented database system is above all a database system. We shall discuss these properties in turn.

1.5.3 OO properties: modelling features

The following features of object-oriented data modelling are important:

- ability to model complex objects,
- support of object identity,
- distinguishing of types and classes,
- definition of class hierarchies with inheritance.

Complex objects

Objects which are structured in a complex way (in brief, *complex objects*) originate from atomic or already constructed objects by applying certain constructors. The simplest objects are of predefined *base types*, for example of type integer, character, string, boolean or real. As constructors to build complex objects, we can usually distinguish the *tuple*, *set*, *bag* (multiset, that is, a set with elements possibly occurring several times), *list* and *array*. The example application described earlier shows that at least tuple and set constructors are important; sets are used as a natural form of representation for (unordered) collections of real-world entities, respectively properties, and tuples are used as an obvious representation of the properties of entities.

Constructors should be applicable to objects in *any* kind of way orthogonally to each other (but not as, for example, in the relational model, where the set constructor is applicable only to tuples and the tuple constructor only to atomic values). Finally, support of complex objects must also offer appropriate *operators* to work with these objects; in particular, it must be possible to operate on whole objects or parts of objects.

Complex objects possess an internal structure: that is, they are composed of simpler components (possibly recursively via several steps). The values of components can be either part of the object's value (*complex value*) or linked to the object by references (*aggregation*). The advantage of the latter procedure is the possibility of reusing information by object sharing.

Complex objects occur in a natural way in most of the new application areas considered for database systems. Think, for example, of the structure of a VLSI circuit, a car body or an aeroplane wing. Such structures have to be flattened in relational systems, so that they can be mapped onto collections of flat tuples. Thus, referring to objects as a whole is unnecessarily difficult, because the relevant information is likely to be composed of several relations. The need to model complex objects, however, is by no means restricted to new database applications; in conventional applications also (as portrayed in our running example), complex objects arise out of the desire to map the given reality onto database structures that are as detailed and adequate as possible.

It should be noted at this point that currently available object-oriented database systems widely support complex objects in their definition languages, and in particular allow constructors to be applied in an orthogonal way. If there are system-specific limitations, these seem to be tolerable.

Object identity

Each object in the real world has an existence or *identity* which is independent of its actual values. To capture this aspect for each object a system-supported identity is maintained by assigning a so-called *object identifier* to each object. The identity assigned to an object remains unchanged throughout the object's entire lifetime; as a consequence, the identity of an object is different from its value, which may be changed.

In programming languages, pointers (memory addresses) fulfil a function similar to object identity; a database system supporting identity, however, goes one step further, ideally by not binding identity to a storage address.

With object identity it becomes possible to distinguish whether two objects are *equal* or *identical*: in the first case the objects have the same values; in the second it is one and the same object. We shall illustrate this aspect by looking at the following relational table:

Employee	*Name*	*Salary*
	Peter	50K
	Susan	60K

Assume this table is updated to become

Employee	*Name*	*Salary*
	Peter	60K

There are various possible reasons for this update, which are difficult to comprehend solely by looking at the result; for example:

- Peter's salary changed and Susan was dismissed.
- Peter and Susan were dismissed, and a new employee with the name Peter and a new salary was employed.

If, on the other hand, the employees are considered as objects which have a unique identity (in addition to their values), the update would be easy to understand. The next example shows that unique identity allows the existence of several objects with equal values. If, for example, you want to add to the following table

Parent	*Child*
Peter	Laura
Susan	Laura

the fact that another mother named Susan has a child called Laura, it becomes necessary to introduce an artificial key attribute if that object identity is not provided by the system.

In addition to the above discussion, the support of object identity has (at least) one further useful implication: it supports *object sharing* (*aggregation*). Two *different* objects can have *shared* (identical) subobjects as components; graphically, a complex object can be represented as a *graph* whose nodes represent object identities and whose edges represent references to other objects. Updates to shared subobjects then need to be made only once with respect to each corresponding compound object.

All these implications could also be achieved by means of artificial keys in relational systems; the fundamental difference is that object identity is maintained *by the system* and thus relieves the programmer of that responsibility.

Types and classes

In object-oriented databases the concepts of *type* and *class* are borrowed from the area of programming languages and reflect the differentiation between a *value* and an *object* at an abstract level.

A *type* is a time-invariant description of a set of possibly complex values. A type can thus be a base type (for example, integer, real or boolean) or structured; no specific operations are bound to a type apart from generic ones (for example, '+' for the type integer). In relational database terminology a type represents a relational schema which defines the admissible relations.

Following this tradition, a type in an object-oriented database system only describes the structural part of a class. A class then encapsulates structure *and* specific behaviour; in particular, operations for the creation and deletion of objects are provided. Moreover, class is a run-time concept such that the set of objects in existence at a given point in time, the so-called *instances*, can be allocated to it as its *extension*.

To give examples, most of the structures in our running example will directly correspond to classes (Section 1.4.2). This means that we can associate objects with each of the resulting class names (for example, *person* or *company*). This is arbitrary in so far as we do not make a further distinction between classes and *named types* in this case; for example, you will not necessarily want to consider each individual address (of a company or person) as an object and thus you may prefer to use address as a named type rather than as a class, so that concrete addresses will appear as *values* and not as objects.

Inheritance

The schema of a database describes at an abstract level a given real-world application area, which is composed of objects which in general are related to one another. We have already come across two ways of expressing relationships, namely aggregation and specialization. A *specialization* defines classes that are subordinate to others (IsA relationship), whereas an *aggregation* expresses component relationships between classes. We now concentrate on specialization.

If one object is more specific than another, it normally has properties that do not apply to the more general object. One example is *employees*, who differ from *persons* by having a *salary*. In other words, each employee *is* (*also*) a person, but with additional, more specific properties. Object-oriented databases have the concept of *inheritance* based on a *class hierarchy* for adequate modelling of such relationships: if the objects of class K_1 are more special than those of class K_2, K_1 will be defined as a *subclass* of K_2 with the effect that K_1 inherits the structure (and the behaviour and, if applicable, the default values) of K_2; thus, each object of the class K_1 possesses all properties which have been defined for class K_2 and possibly additional, more specific properties which have been defined for class K_1. We have already discussed inheritance as one of the five principles of object orientation and will come back to it in Section 2.4, where we look at some interesting aspects in more detail. Here we are going to demonstrate by means of an example how inheritance affects reusability of behaviour.

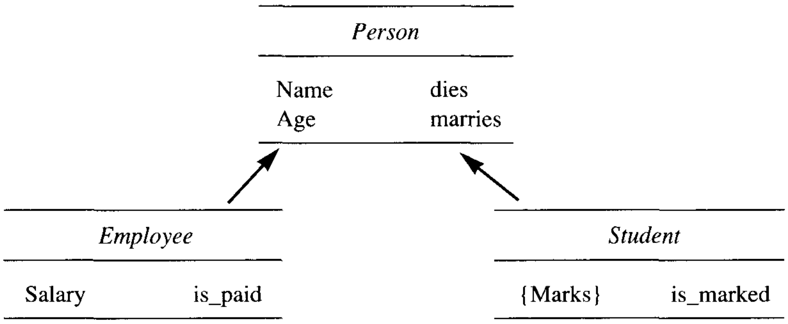
We suppose that the administration of a university keeps data on employees and students, with each of these objects being characterized by certain attributes and certain operations applicable to them. In a system without inheritance (for example, in a relational system) you would arrive at roughly the following representation:

Employee	
Name	dies
Age	marries
Salary	is_paid

Student	
Name	dies
Age	marries
{Marks}	is_marked

This means that six programs must be written.

In an object-oriented system, by using inheritance you would arrive at the following representation, which accounts for the common characteristics of persons, employees and students:



Now, only four programs must be written.

Remarks

According to our discussion, an object-oriented database system must provide, in particular, a *data model* which covers the already mentioned modelling properties. We would like to study the significance of these properties to our running example.

First of all, we notice that the previously given description of our application can be represented unchanged in an object-oriented data model; the only requirement is that class *names* can also be used as *type* names, which is generally possible with current systems. Once again we shall study the previously given description of *companies*:

```
Company: [
  Name: String,
  Headoffice: Address,
  Subsidiaries: { Subsidiary },
  President: Employee ]
```

We can regard this as a definition of a class with the name Company, whose objects should have the four specified attributes, here arranged as a tuple type. The type of the attribute Name is atomic, but all other attribute types represent references to objects of other classes; this is only expressed by stating the name of the respective class as the type of the relevant attribute. Company *objects*, that is, instances of the class Company, are characterized by an identity and by a value for each of the above attributes, where each value of the attribute Subsidiaries is a set of references to objects of class Subsidiary.

By and large, we can illustrate the schema of our example application, automobile sales, with the graph shown in Figure 1.8. A single arrow shows an aggregation, a double arrow a specialization, and a * signifies a set-valued attribute. For subclasses only the specific attributes are represented, not the inherited ones. Figure 1.9 shows possible extensions to the classes of this schema; object identities are written in the form #*n* with *n* being a natural number.

So far we have assumed that class definitions remain unchanged over time. In fact, *schema evolution* in databases is traditionally considered to be an exceptional case. Because data is persistent, changing the structure of a relation, for example by adding a new attribute, may raise severe problems since a large amount of data has to be reorganized. Object-oriented design is inherently evolutionary, so that modifying the schema is no longer an exception. As a consequence, object-oriented databases have to provide mechanisms that enable changes in class definitions to be accommodated more smoothly. To some extent, schema evolution is supported by inheritance. To give an example, adding a new structure and behaviour can be achieved by assigning it to a new class, which is then linked to the old class as a subclass in order to reuse already existing structure and behaviour by means of inheritance. In general, however, restructuring a class hierarchy or class definitions may be inevitable, which may then be responsible for complex and time-consuming adaptations of previous definitions.

1.5.4 OO properties: language features

Apart from modelling features (complex objects, object identity, types and classes, class hierarchy with inheritance), an object-oriented database system must have certain language features:

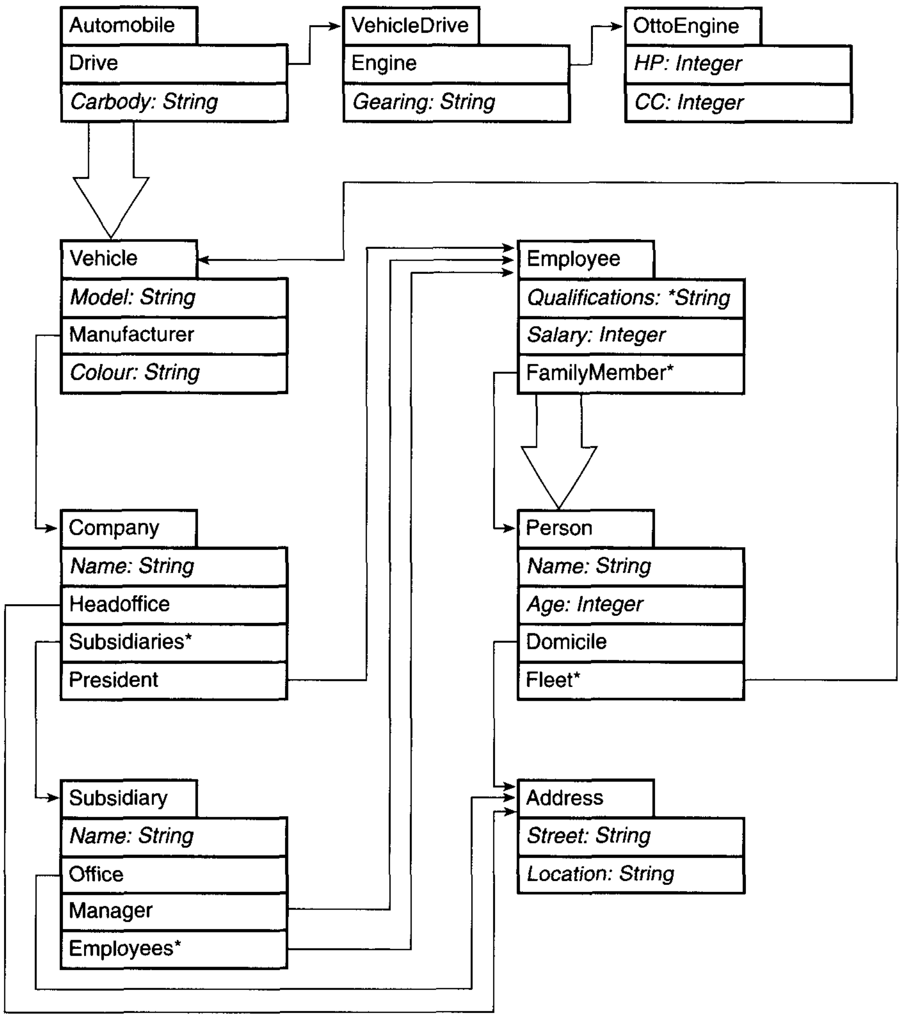


Figure 1.8 Class structure of automobile sales.

- encapsulation;
- overloading, overwriting and late binding;
- ad hoc queries;
- computational completeness;
- version management;
- extensibility.

Vehicle

	Model	Manufacturer	Colour
#10	Golf	#41	red
#11	323	#42	blue
#12	R4	#43	green
#13	Coach	#44	black

Automobile

	Drive	Carbody
#10	#20	Sedan
#11	#21	Hatchback
#12	#22	Sedan

VehicleDrive

	Engine	Gearing
#20	#30	Shifting
#21	#31	Automatic
#22	#32	Automatic

OttoEngine

	HP	cc
#30	80	1600
#31	70	1800

Company

	Name	HeadOffice	Subsidiaries	President
#41	VW	#80	{ #50,#51 }	#60
#42	Mazda	#81	{ #52 }	#61
#43	Renault	#82	{ #53 }	#62
#44	Wunder	#82	{ }	#62

Subsidiary

	Name	Office	Manager	Employee
#50	main	#80	#60	{#68,...}
#51	south	#83	#65	{ #65,...}
#52	D	#84	#66	{ #66,...}
#53	RFA	#84	#67	{ #67,...}

Address

	Street	Location
#80	Fabrik 1	Berlin
#81	Adzam 5	Tokyo
#82	1'rue Oui	Paris
#83	Solarweg 1	Kehl
#84	Asiaweg 7	Münster

Person

	Name	Age	Domicile	Fleet
#60	Slim	40	#80	{ #10, #11 }
#61	Chubby	50	#81	{ }
⋮	⋮	⋮	⋮	⋮
#68	Thin	35	#82	{ #12 }

Employee

	Qualifications	Salary	FamilyMembers
#65	{ A levels }	8000	{ #61 }
#66	{ Apprenticeship }	9000	{ #63, #64, #68 }
#67	{ Diploma, Doctorate }	8500	{ }
#68	{ CourseA }	7000	{ }

Figure 1.9 Possible extensions to the classes shown in Figure 1.8.

Encapsulation

In higher programming languages *encapsulation* generally serves two purposes: differentiation between the *specification* and *implementation* of an operation, procedure or module; and the desire to achieve modularity. The principal idea is to form *abstractions* which *hide* certain details from the outside world, in particular details of the implementation.

A descriptive example of the encapsulation principle taken from everyday life is the radio: certain knobs are mounted on its housing, for switching it on or off, for station selection and for volume control. The radio operator only needs to know which knob is provided for which function; but he or she does not need to know how this function is realized within the radio.

With programming languages the encapsulation concept has been developed and examined particularly in connection with *abstract data types* (ADTs); in an ADT this concept is especially useful for differentiating between the interface visible to the outside world and the internal implementation of a special data structure (for example, a pushdown store). In the context of databases, at the schema level this concept means that a class contains structural information (in the form of its assigned type) as well as *behaviour*. In concrete terms, the latter means that *methods* which are understood by the objects of a class can be allocated to that class. This behaviour is visible to the outside world in the form of a set of so-called *signatures*, which indicate the *message names* that can be sent to the class, possibly along with some argument types and a result type. If a class receives such a message, it executes the method allocated to the message.

Note that *logical data independence* can be achieved in this way, because a method implementation can be changed without changing the interface.

Overloading, overwriting and late binding

Since the same method, but with a different implementation, may be defined several times in a class hierarchy, method names can be *overloaded*. If a message is sent to an object, the most specific implementation of the method should be executed; this implementation *overwrites* all the other possible implementations. To implement such overwriting, a *late binding* of implementations to method calls is required, which defers the decision about what is to be executed to run time.

In particular, due to the encapsulation principle, the use of the same message names in different contexts is quite familiar. Think, for example, of a message with the name *display*, which

- should be understood by *person* objects to mean that the attribute values of the person receiving the message should be displayed in the form of a table;
- and by *automobile* objects to mean that a three-dimensional picture should be displayed.

Figure 1.10 summarizes what we have learned so far about classes: a class encapsulates structure (type) and behaviour (messages with allocated methods) and is

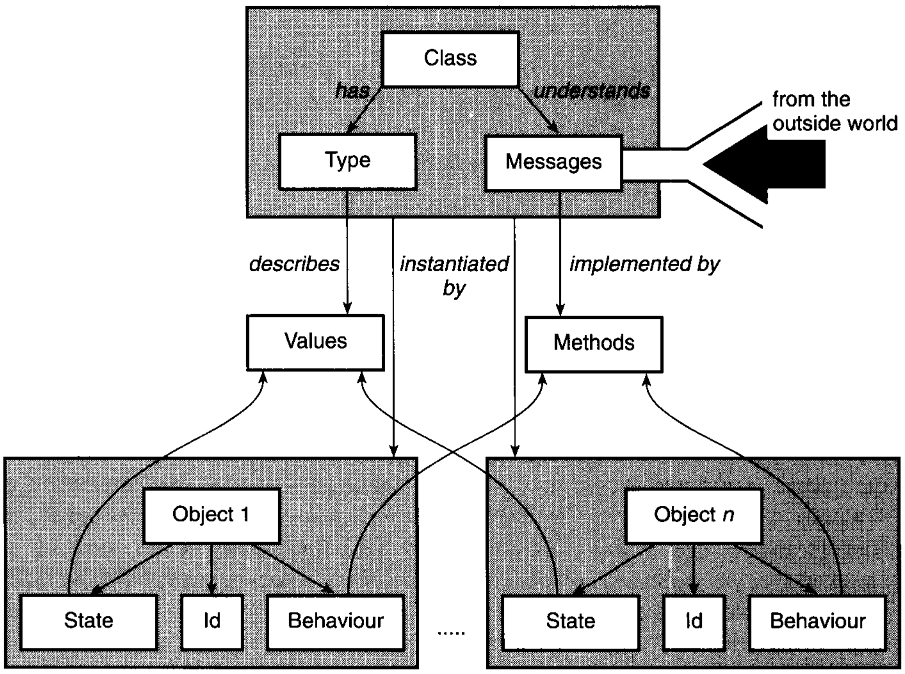


Figure 1.10 Classes and their objects.

instantiated by objects, which have an identity, a state and a behaviour. Note that modelling and language features are closely connected.

Ad hoc queries

One of the most distinctive features of database systems is set-oriented descriptive languages which enable the user to make so-called *ad hoc* or dialogue queries. Compared with programming languages, such a language generally has only limited computational power but a high level of abstraction, such that even inexperienced users can, in most cases, work satisfactorily with a database. SQL is the standard today for relational database systems.

Ad hoc access to objects is also desirable for object-oriented databases, because the user would not want to write a program for every individual query. However, such a language will at least be expected to meet the requirements used in relational systems, for example *universality*, *descriptivity* and *optimizability*; we discuss these concepts in Chapter 2. In Chapter 4 we discuss examples of object-oriented languages with these properties. It will also become evident that so far no agreement has been reached on *how* these properties can be achieved. We should like to point out that for object-oriented databases – like relational databases – a distinction can be made between *algebraic*, *rule-based* and *calculus-based* languages (see Chapters 6 and 7).

Computational completeness

The demand for *computational completeness* (Turing computability) of an object-oriented language originates in the desire to overcome the *impedance mismatch*. Viewed from a programming language perspective this is trivial, because in a programming language *every* computable function can be expressed on the given input data. Conventional databases, particularly for reasons of efficiency, are generally very limited in their computability. Object-oriented systems offer a good chance to overcome these limitations, because by binding methods to classes a link to a programming language can be established. This is especially the case when a language like Smalltalk or C++ is provided for method *implementation*.

Version management

The system functionality of an object-oriented database system has to cover a series of wider-ranging demands than those required of conventional systems. This is mainly due to the new application fields in which this new database type can primarily be found. In new applications like CAD or CASE, for example, you find *versions* of individual design objects which are created and possibly discarded during the design process. In the course of development, versions of objects are combined to form *configurations*, which eventually result in products to be manufactured. If a database system is to support such an application environment adequately, it has to offer version management.

Extensibility

An object-oriented database system provides the user with a set of predefined types and constructors which can be used for data or object modelling and also for writing applications. Although numerous applications can thus be covered, there will often be demands for further types, and possibly also constructors, which are specific to a particular application. In this case, extensibility allows the user to define new types and constructors and thereby to adapt the system to the user's application domain. It is important that such extensibility is supported by the system, so that there is no difference in the use of predefined constructors and new types of constructors.

The extensibility aspect just described refers only to the conceptual level of an object-oriented database system. Ideally, such a system would also support extensibility at the internal level. This means that you can, for example, introduce new storage structures to the system if the given application requires it or if it seems appropriate. Think, for instance, of data structures for storing pictures or multimedia data, which are generally not available a priori.

1.5.5 Object-oriented database system properties

Finally, we examine the system properties of object-oriented database systems; we discuss the concepts common to database systems and explain these briefly.

Persistence

In a database system data is always stored in a *persistent* manner: that is, the data or objects on which the user or programmer works survive the execution of a process and can then be used again in other processes. The persistence of data and objects is guaranteed automatically by a database system; unlike, for example, the execution of non-database programs, the user does not need to worry about whether persistence is present.

In an object-oriented database system, persistence should be orthogonal in the sense that each object, independent of its type, can become persistent. Furthermore, it might be desirable for the user to be able to distinguish between *persistent* and *transient* objects, or for objects to become persistent only for the duration of a predetermined time interval.

Multi-user control

In order to realize multi-user access and ensure a certain degree of fault tolerance, database systems implement a *transaction concept.* With this concept several users can have simultaneous access to shared data, with each user believing that he or she has exclusive access to it. The transaction processing component of a database system ensures that transactions are processed according to the so-called *ACID principle*:

A: they are executed *atomically* (according to the all-or-nothing principle);

C: they *preserve* the *consistency* of the stored data;

I: they are *isolated* from other, simultaneously executed transactions and do not see inconsistent or uncommitted data;

D: transaction effects are *durable* (*persistent*), which means they survive any kind of damage to the data that may occur after the transaction has been successfully completed (committed).

The transaction concept has proved itself in database systems as the paradigm for the synchronization of concurrent access; moreover, techniques for transaction processing, which can be efficiently implemented, are available.

In principle, this situation does not change with object-oriented database systems, but there are some new problems which require special treatment. If an object-oriented system is applied in a CAD application, *long transactions* may occur which represent the work of individual designers. There are obvious reasons why the demand for atomicity, for example, is tenable only to a limited extent for such transactions. The fact that executions of transactions can no longer be regarded as straightline programs has to be taken into account, because method calls within the executions of transactions may cause complex nestings to occur.

Recovery

The transaction concept is also used to provide fault tolerance in a database system. When an error occurs which, for example, corrupts the content of the database buffer, the system must be able to restore a consistent state; this also applies to certain other software errors and even to hardware errors. For this purpose, a database system normally has certain recovery protocols at its disposal. But it may also be necessary to account adequately for the specific situations which may occur in object-oriented database systems and which are dependent on the chosen architecture (see Section 4.1).

Secondary storage management

In database systems data is stored on secondary storage media, and sophisticated functionality is provided to access data and transfer it between main and secondary storage. These techniques deal with, for example, index management, the clustering of data, buffer management, access path selection and even query optimization. All these functions are transparent to the user, which means that physical data independence is achieved. However, the realization of these functions is closely linked to the performance of the system.

1.5.6 Remark

A *data model* which is based on the object-orientation paradigm can be defined and implemented in different ways; for example, many models are available commercially. It therefore seems attractive to consider *standardization* in this field; this is already being pursued by the *Object Database Management Group* (ODMG) in particular. The convergence of rather different data and object models to a model with common features can now be seen; the ODMG proposal's main intention is to lay down this trend (see Chapter 5).

1.6 Bibliographical notes

General introductions to the field of databases and database systems are, amongst others, Date (1995), Elmasri and Navathe (1994), Silberschatz et al. (1997) and Vossen (1994). Cattell (1994) describes the requirements that data models and database systems need to fulfil in order to run new applications. The relational model is based on Codd (1970). Introductions to SQL, especially the system-independent standard, are given by, amongst others, Date and Darwen (1993) and Melton and Simon (1993). The basic principles of transaction processing in database systems are discussed by Bernstein et al. (1987) and Gray and Reuter (1993).

The use of the object-orientation paradigm in databases and the basics of object-oriented data models and database systems are discussed by, amongst others, Bertino and Martino (1991, 1993), Dittrich et al. (1991), Gupta and Horowitz (1991), Kemper and Moerkotte (1994), Khoshafian (1993), Khoshafian and Abnous (1990), Kim (1990), Kim and Lochovsky (1989), Loomis (1995), and Zdonik and Maier (1990). An important contribution to the question of what an object-oriented database is was made with the *Manifesto* by Atkinson et al. (1989). This paper clarified for the first time which properties are necessary, optional or still open, and gave a definition of object-oriented database systems which is still valid today.

The running example introduced in Section 1.4 of this book is based on Kifer et al. (1992).

2 Aspects of object-oriented database languages

In this chapter we elaborate on query languages for object-oriented databases. First, we discuss certain language features which seem to be important in the context of object orientation. As a framework we use an SQL-like language. In particular, we examine in detail two characteristic aspects of such languages: support for *navigation* in object-oriented databases with so-called *path expressions*, and the *inheritance* of attributes and methods and its impact on *static type-checking*. We begin by stating some general requirements.

2.1 General requirements

Object-oriented database languages are expected to fulfil the same requirements as languages based on other data models, particularly the following:

(1) *Universality*: The language should not be designed with a view to being applied for a specific purpose; instead it should be universally applicable.

(2) *Descriptivity*: The language should be characterized by a high level of abstraction independent of implementation details, and in particular it should support set-oriented access.

(3) *Optimizability*: For an underlying system it must be possible to optimize expressions of the language prior to execution; therefore, appropriate optimization rules and strategies must exist.

(4) *Closedness*: It must be possible to describe every result of a language expression within the given data or object model; this ensures that the result of a query can be used as input for a subsequent query.

(5) *Completeness*: Every concept of the data model in question must have a processing counterpart.

(6) *Genericity*: The language should contain generic operators (for example, selection, projection and set operations) which can be applied to values depending only on the type structure.

(7) *Expressive power*: In particular, the language should surmount the restrictions imposed on relational languages; that is, recursive traversing of object sets should be possible and Turing-completeness should also be guaranteed.

(8) *Extensibility*: The language should support both user-defined and system-defined types.

Two different approaches to language design can be observed, both of which try to comply with these requirements.

Approaching the issue from the database perspective, attempts have been made to extend SQL in order to overcome well-known restrictions and to accommodate the principle of object functionality. We shall adopt this point of view in this chapter, indicating how such an SQL-type language can be expanded for use with object-oriented databases; we shall come back to this approach later in our discussion of Illustra, SQL3 and O_2.

On the other hand, from the programming language perspective, attempts have been made to increase the functionality of object-oriented languages by incorporating database functionality. One aspect of special importance here is *persistence*, which in traditional programming languages is at best realized indirectly (for instance, by manipulating external files). For databases, however, persistence is an essential feature, because stored data survives the execution of a transaction. This route leads to the development of so-called database programming languages, which we shall exemplify using GemStone Smalltalk and the persistent C++ from ObjectStore.

2.2 Desirable properties

To make our discussion of object-oriented database languages more concrete, we now carry out an exercise to extend SQL. In particular, we restrict ourselves to select-from-where expressions; embedding such expressions into procedural languages will be considered in our discussion of concrete systems in Chapter 4. When extending SQL the following aspects seem to be important:

- Values have to be distinguished from objects: that is, object identities.
- Objects may have a complex structure which demands special navigational support.
- Some properties of objects may be defined by inheritance.
- Because of set-valued attributes and class extensions, sets play a more prominent role.
- Creation of objects.

However, the reader should be warned: because we believe that it is still too early to predict what kind of object-oriented SQL will finally be defined, in the remainder of this chapter we do not make reference to a concrete system or standardization proposal – this is the topic of Chapters 4 and 5. The following examples are based on our running scenario (see Figure 1.8).

2.2.1 Elementary access to objects

Every object has an identity independent of its value. Consequently, a database may contain different objects of equal value and equal behaviour. Access to objects can be motivated by various reasons: for instance, access to the objects themselves or to the values of their attributes. This distinction is explained in the following example.

Example 2.1

The following query asks for the identities of those vehicles whose attribute `Model` has the value `Tipo`:

```
select f
from f in Vehicle
where Model = 'Tipo'
```

The use of variable `f`, which is introduced in the from-clause and referred to in the select-clause, signals that we are looking for object identities in the answer. Because identities as such are not of interest in general (especially when they are generated by the system rather than assigned as names by the user), the content of variable `f` may be used for further processing. Obviously, there must be a linguistic distinction between access to objects and access to the attribute values of an object. The following query asks for the values of all attributes of vehicles with the model value `Tipo`:

```
select *
from Vehicle
where Model = 'Tipo'
```

2.2.2 Access to complex objects

So far we have discussed query-types referring to objects of *one* class. In general, objects may have a complex structure: for example, they may be defined by aggregation. Thus referencing other objects becomes necessary when we want to decompose an object into its subobjects or to access the properties of encapsulated objects.

Example 2.2

In this example we want to search for all blue vehicles manufactured by Ford:

```
select *
from Vehicle
where Colour = 'blue'
and Manufacturer.Name = 'Ford'
```

One should note that attribute `Manufacturer` of class `Vehicle` references class `Company`. The value of `Manufacturer` is an object identity from class `Company`. Access to the attributes of this referenced object is possible using a so-called path expression (as shown in the above example). We look at path expressions more closely in Section 2.3.

Queries of this kind principally reflect a situation which also occurs in network databases, where the user must be able to navigate through the schema because objects of a particular class have attributes whose values reference objects of another class. Obviously, such navigation chains may be long and even cyclic, depending on the database schema.

Example 2.3

Let us also allow path expressions in the select-clause:

```
select President.Salary
from Company
where Headoffice.Location = 'Rome'
```

This query obtaines the salaries of all presidents of companies with their head office in Rome.

Path expressions may also occur in comparison expressions:

Example 2.4

The following query searches for all companies whose head office address is identical to the address of its president:

```
select *
from Company
where Headoffice == President.Domicile
```

Note that `Domicile` is defined for `President` by inheritance from `Person`. In cases where the head office and the domicile of the company's president are objects of class `Address`, object identity (rather than value identity) will be tested. Using '==' instead of '=' is meant to mark this distinction. If, on the other hand, only the location is of interest, it is necessary to test the equality of values:

```
select *
from Company
where Headoffice.Location = President.Domicile.Location
```

2.2.3 Explicit join

With the help of path expressions, predefined relationships between objects can be traversed. In order to deal with these situations, relational databases require a join operation. Clearly, object-oriented databases must still be able to process such explicit joins, because generally not all the relationships which are of interest have been predefined in the schema:

Example 2.5

We are interested in persons and vehicles where the name of the person is identical to the name of the president of the company that manufactures the car:

```
select p,f
from p in Person, f in Vehicle
where p.Name = f.Manufacturer.President.Name
```

As one can see in Figure 1.8, there is no predefined relationship that could have been used directly in this context. Here `Name` is defined for presidents by inheritance.

2.2.4 Equal treatment of attributes and methods

When looking at the properties of an object, one often cannot distinguish between attributes and methods with no parameters. It is possible, therefore, to refer to methods and attributes in the same way. Let us suppose that for class Vehicle, in addition to the attributes, there is a parameterless method PresentValue which determines this value when requested. In Example 2.1, the use of * produces the result of applying PresentValue to the respective objects, as well as the values of the attributes.

Methods may be also called with parameters:

Example 2.6

Suppose that we are interested in managers' deputies and that there is a boolean method `Deputy` with a parameter which allows us to test, for all managers, whether the manager currently under consideration is a deputy.

```
select a1, a2
from a1 in Employee, a2 in Employee
where a1.Deputy(a2)
```

Note that method `Deputy` actually has two parameters, but since we are using a path expression, the first parameter appears before the name of the method and is not syntactically treated as a parameter of the method.

2.2.5 Access to abstract types and classes

The notions of abstract data and object types, as well as the associated principle of encapsulation, are supported in object-oriented database systems by the class concept. Consequently, abstraction which results from encapsulation must be respected and, in particular, queries must not be allowed to access the internal representation of instances of the class in question. In other words, query expressions may refer only to information which is declared as public.

Let us suppose that in Figure 1.8 a private (and therefore invisible) attribute DateofBirth has been defined for class Person, and that the age of a person can be determined with a specific method Age which uses the date of birth. Direct access to DateofBirth must not be permitted, but only the calculation of age, provided that method Age is public.

On the other hand, it should be possible to use public properties and methods in query expressions without any restrictions; in particular, names of methods should be allowed wherever names of attributes might occur.

Example 2.7

We want to search for all persons over the age of 50, supposing that method `Age` is public and determines the age of a person on the basis of the (private) attribute `DateofBirth`:

```
select *
from Person
where Age > 50
```

The age of everyone called Peter Smith is determined as follows:

```
select Age
from Person
where Name = 'Peter Smith'
```

If no distinction is made between attributes and parameterless methods, methods may replace attributes in path expressions; this allows sequences of method calls to be defined in one expression.

In some situations access to the internal structure of an object type may be desirable. BLOBs (Binary Large OBjects) are a typical example of this. They are used to display large data values (for example, audio or video data). However, if one wishes to access, say, the first 250 bytes of a picture object classified as a BLOB, the system must support an appropriate read operation, even if this requires access to the internal representation of the picture object.

2.2.6 Access to sets of objects

Query languages for object-oriented databases must support the manipulation of *sets* of objects, or, to use a more general term, *collections*, because in an object database a query result may consist not only of sets but also, for example, of lists (or bags, fields, and so on). In this context it is important to distinguish the two roles of a class: a class can be used as a type, thereby stating the structure and behaviour of objects, and as a set of such objects existing at some point of time: that is, its *extension*. Typically, class extensions have to be maintained explicitly by the programmer. In particular, different extensions of the same class can be maintained at the same time by assigning different names.

Example 2.8 ______________________________________

The following query determines all persons over the age of 50:

```
select Name
from Person
where Age > 50
```

In this example `Person` denotes both a class and its extension. If it is possible to name sets of objects, one could define `MyFamily` as a subset of the objects in `Person` and use this new set for further queries:

```
MyFamily :=
select p
from p in Person
where Name = MyName
```

```
select Name
from MyFamily
where Age > 50
```

Such a technique closely resembles the defintion and processing of *views* in relational databases.

In languages with an SQL-type syntax it should be possible to use in the from-clause any kind of collection that can be derived from the database: thus not only sets but also, for example, lists. Moreover, it must be possible to use named collections. The following example shows that such collections may arise implicitly as the value of an expression.

Example 2.9

The following query searches for the names of manufacturers of blue vehicles:

```
select Name
from (select Manufacturer from Vehicle
where Colour = 'blue')
```

The use of collections in queries can also be extended to where-clauses. This allows the formulation of selection conditions which test for membership within such a collection:

Example 2.10

Let us determine the number of companies whose presidents' annual salary exceeds 200 000 dollars:

```
select count(f)
from f in Company
where President in
       (select a
       from a in Employee
       where Salary > 200000)
```

Set-valued attributes or methods in path expressions are problematic when several such attributes or methods occur in the same path expression:

Example 2.11

We should like the following query to determine the salaries of employees working in companies' subsidiaries:

```
select f.Subsidiaries.Employees.Salary
from f in Company
```

The path expression is evaluated from left to right for each company `f` in the extension of class `Company`. First the set of subsidiaries of a given company `f` will be determined. To avoid a *type error*, we cannot apply attribute `Employees` to the result of `f.Subsidiaries`, because the latter is a set, whereas `Employees` is defined only for single subsidiaries. If we apply attribute `Employees` to every element of the set we shall obtain a set of sets, because for every subsidiary there exists a set of employees. If attribute `Salary` is then applied to every element of this set, again a *type error* will occur, because the salary is defined only for the elements of the individual sets.

One solution to intermediate sets in path expressions is to implicitly remove the set structure whenever single values are expected; to put it another way, sets whose elements are themselves sets are by default unnested. Following this strategy, `Employees` is not applied to a set of subsidiaries, but to each subsidiary in turn. The result is a set of sets; however, `Salary` is applied to each element, that is, each employee, such that for one given company `f` the final result is a set of salaries. Because the path expression has to be applied to each company, for the final result of the query a nesting of sets has again to be removed so that one set of salaries will ultimately be derived.

Another strategy would be to state the unnesting of sets explicitly by means of a *flatten* operator:

```
flatten(select flatten(flatten(f.Subsidiaries).Employees).Salary
        from f in Company)
```

In any case, defining the semantics of path expressions becomes a subtle matter when attributes or methods may be set-valued.

2.2.7 Implications of a class hierarchy

Because a subclass relationship represents a specialization between classes, which means that every object of the subclass is also an object of the superclass (IsA relationship), objects belonging to a subclass inherit those properties defined in its superclasses, unless the properties are overwritten by the subclass.

Example 2.12

The following query determines the qualifications of employees whose name is Peter Smith. Note that the attribute Name is not defined explicitly for class Employee but is inherited from its superclass Person:

```
select Qualifications
from Employee
where Name = 'Peter Smith'
```

The following query asks for employees aged over 50; it is assumed that a method `Age` as discussed in Example 2.7 is defined for `Person` and thus inherited by `Employee`:

```
select * from Employee where Age > 50
```

We like to use IsA relationships explicitly in query expressions.

Example 2.13

We first determine employees and everyone else over the age of 50:

```
select * from Person where Age > 50
```

In contrast to this, the following query selects only persons over 50 who are not employees:

```
select *
from p in Person
where Age > 50 and p not in Employee
```

Queries of this type may result in heterogeneous sets: that is, sets containing elements of different types. The first query in the last example defines objects of class Person as well as objects of class Employee. For the latter there are more attributes defined than for class Person, which may lead to *type errors*, if the result of the query is used for further processing. There are various ways to remedy this situation, for instance by stating that all objects in the resulting set are to carry all the attributes and that their values are to be null values if they would otherwise be undefined. Alternatively, it can be stated that objects of a heterogeneous set are to be represented only by their identities.

Ultimately, neither of these possibilities is satisfactory. If we choose the first, we are confronted with the problematic issue of null values, as known from relational databases. On the other hand, choosing the alternative may not deliver the intended query result and thus may necessitate further processing. The next example illustrates how further processing of a heterogeneous set can be achieved by using a case expression. In doing this, however, we go beyond select-from-where instructions.

Example 2.14

We want to determine both employees and other persons over 50 and then process the data depending on their class membership.

```
case (select *
     from Person
     where Age > 50)
if Employee then ...
else...
end case
```

2.2.8 Generating and modifying objects

A data manipulation language must allow *modification* operations to be carried out on instances of classes. We shall deal with the operations insert, modify and delete, and discuss some language aspects in connection with these operations.

To generate new objects of a certain class, we assume the existence of a generic method 'new' which, in principle, could be applied to every class.

Example 2.15

Inserting new objects into the class Address could be implemented as follows:

```
insert new(Street, Location)
values('Broadway', 'New York')
into Address
```

First, a new object with the attributes Street and Location is generated using 'new', and subsequently values are attached to these attributes and the object is inserted into the existing extension of the class Address.

In the simplest case only the atomic values of an object are affected by a modification operation:

Example 2.16

The following modification operation ensures that Ford will manufacture only red cars:

```
update Vehicle
set Colour = 'red'
where Manufacturer.Name = 'Ford'
```

Obviously, update operations become more complex as soon as objects that contain references to other objects are to be generated. It must be possible to supply these references in the form of values or to automatically assign them a default value, generally *nil*, which will later be replaced by the correct value. This is illustrated by the following example.

Example 2.17

Let us consider the insertion of a new vehicle for which an object `Manufacturer` already exists. The first instruction selects the identity of the manufacturer with the name of Ford in variable `m`; the second instruction then inserts a new vehicle made by that manufacturer:

```
m := select f
     from f in Company
     where Name = 'Ford'

insert new(Model, Manufacturer, Colour)
values ('Mondeo', m, 'red')
into Vehicle
```

The following closed expression achieves the same effect:

```
insert new(Model, Manufacturer, Colour)
values ('Mondeo', select f
                  from f in Company
                  where Name = 'Ford',
       'red')
into Vehicle
```

In principle, both expressions contain a *type error*, because the query defines a set (with one element) and the manufacturer of a vehicle is not set-valued. To avoid such situations, an appropriate *flatten*-operator should be available.

It is evident that insertion or modification operations must also be capable of generating complex objects such as lists. In other words, the modification operations of the data manipulation language must be able to use the constructors provided by the type system. And finally, modification operations may also have an effect on existing collections, as shown in Example 2.18:

Example 2.18

John Smith is to be employed in Ford's subsidiary in Chicago. In order to realize this modification the following procedure may be required:

(1) Insert `John Smith` into the class `Employee`.

(2) Select the object from the class `Subsidiary` of the company with the name Ford and the location 'Chicago' in its address.

(3) Insert the object `Employee` with name `John Smith` into the set `Employees` of this subsidiary.

2.2.9 Generating new classes, instances and operations

A data manipulation language of an object-oriented database must allow views to be defined on a given database state. Similarly to relational databases where relations are defined by query expressions, we can regard a set of values defined by a query expression as a view. In object-oriented databases, sets of object identities and extensions of classes can also be defined. Moreover, it should also be possible to define sets of objects *with* values. It must be possible to generate classes and their extensions either via the appropriate constructs of the definition language or *dynamically* by queries. Of course, it must be possible to use these classes in subsequent queries in the same way as the classes defined in the schema.

Example 2.19

Let us define a view in which person objects, their name values and vehicles manufactured by Ford are to be interrelated in such a way that for each tuple the name of the person and the name of the company's president correspond. For this purpose we introduce a new class K and assign to this class a set of tuples as described below:

```
create class K as
  select p, p.Name, f
  from p in Person, f in Vehicle
  where f.Manufacturer.Name = 'Ford'
        and p.Name = f.Manufacturer.President.Name
```

The result of this query is of the following type:

```
{[p: Person, Name: String, f: Vehicle ]}
```

Obviously, this type must be permissible for the object model in question.

Thus, the language used for defining views must not only be able to use the type constructors of the underlying object model but also contain the associated instance constructors. A distinction must be made depending on whether the outcome of the query is to consist of new objects or not. In accordance with relational views it is justifiable for views to introduce no new object identities, because they constitute derived data and objects: that is, view formation is *object-preserving*. But it is also tenable to define a view as *object-generating* in order to be able to realize, for example, inheritance on the objects in the view.

Note that in the previous example both variants would be justifiable. As an object-preserving feature, the identities of the persons could be used. However, this requires an operator to collect for each person all the corresponding vehicles into one set; in this case the type of the query would become {[p: Person, Name: String, Vehicles: {Vehicle}]}, where {Vehicle} is a set type with element type Vehicle. Thus the view would serve to expand the properties of the existing objects of class

Person dynamically. The definition of new objects requires the dynamic generation of object identities for objects that are to be included in the result of a query. There are at least two different ways to achieve this. One possibility is to generate new objects during the calculation of the query result: that is, to assign new identities to the elements of the resulting set. In our last example every generated instance of the resulting class of the type

```
{[ p: Person, Name: String, f: Vehicle ]}
```

would then be an independent object. A second possibility would be first to generate values which could later be converted into objects if required.

The dynamic generation of objects in queries leads to another problem, because it needs to be clarified how the new class is to be integrated into the already existing class hierarchy. Again, there are various approaches. On the one hand, it can be argued that results have no superclass, with the possible exception of a class Object which is the root of the given hierarchy; on the other hand, one can establish some rules which govern how the super- and subclasses of a view can be derived from the existing ones.

Finally, we should like to point out that the above-mentioned strategies for generating new classes and their instances can also be extended to cover operations, since generating new classes requires that class-specific behaviour can also be generated dynamically. The language used must allow for the introduction of new operations, for which there are at least two options: first, referencing existing implementations when new object types are generated, that is, resorting to operations previously formulated with a definition language; or, second, dynamically generating operations in queries as instances of a separate class Operation.

2.3 Navigating with path expressions

In object-oriented databases, if we want to process the overall structure of a compound object which is defined by aggregation, we have to follow references between objects. For this purpose typically a navigating technique based on path expressions is used. Now we are going to explain the relationship between path expressions and relational join operations and then, after having discussed some more examples, we shall outline some possible extensions of path expressions. We shall not go into formal details – this is postponed to Chapter 3.

Attributes and methods in object-oriented databases are formally scalar or set-valued functions. For example, an attribute which is defined for a class can be regarded as a function from the extension of the class to a set of values. Object-oriented databases take advantage of this functional relationship to develop a simpler syntax. We want to illustrate the relationship between join expressions and path expressions by an example based on the schema in Figure 1.8. Consider schemata for companies and vehicles defined as follows:

```
CREATE TABLE Company
     (CompanyID INT NOT NULL,
     Name VARCHAR NOT NULL,
     Street VARCHAR NOT NULL,
     Location VARCHAR NOT NULL,
     President INT NOT NULL,
     PRIMARY KEY (CompanyID),
     FOREIGN KEY (President)
     REFERENCE Employee (EmplNo)
     );

CREATE TABLE Vehicle
     (VehicleID INT NOT NULL,
     Model VARCHAR NOT NULL,
     Manufacturer INT NOT NULL,
     Colour VARCHAR NOT NULL,
     PRIMARY KEY (VehicleID)
     FOREIGN KEY (Manufacturer)
     REFERENCES Company (CompanyID)
     );
```

The president of the company manufacturing the car with number 93 can then be determined by the following SQL expression:

```
select c.President
from Vehicle v, Company c
where v.VehicleID = 93 and v.Manufacturer = c.CompanyID
```

This expression is evaluated in the following manner: first, all vehicles with number 93 are selected; then a join between these tuples and the company tuples is performed, joining tuples where the manufacturer of the vehicle is identical to the company under consideration; finally by projection all attributes of the tuples are deleted from the resulting set, except the attribute President.

In an object-oriented schema, however, two classes Vehicle and Company would have been defined (cf. Figure 1.8):

```
Company: [
      Name: String,
      Headoffice: Address,
      President: Employee ]

Vehicle: [
      Model: String,
      Manufacturer: Company,
      Colour: String ]
```

Let #93 be the object identity of the vehicle in question and assume that we can refer to this identity by variable v. Manufacturer, Headoffice and President are attributes whose values reference respective objects of classes Company, Address and

```
case (select *
     from Person
     where Age > 50)
if Employee then ...
else...
end case
```

2.2.8 Generating and modifying objects

A data manipulation language must allow *modification* operations to be carried out on instances of classes. We shall deal with the operations insert, modify and delete, and discuss some language aspects in connection with these operations.

To generate new objects of a certain class, we assume the existence of a generic method 'new' which, in principle, could be applied to every class.

Example 2.15

Inserting new objects into the class `Address` could be implemented as follows:

```
insert new(Street, Location)
values('Broadway', 'New York')
into Address
```

First, a new object with the attributes `Street` and `Location` is generated using 'new', and subsequently values are attached to these attributes and the object is inserted into the existing extension of the class `Address`.

In the simplest case only the atomic values of an object are affected by a modification operation:

Example 2.16

The following modification operation ensures that Ford will manufacture only red cars:

```
update Vehicle
set Colour = 'red'
where Manufacturer.Name = 'Ford'
```

Obviously, update operations become more complex as soon as objects that contain references to other objects are to be generated. It must be possible to supply these references in the form of values or to automatically assign them a default value, generally *nil*, which will later be replaced by the correct value. This is illustrated by the following example.

Example 2.17

Let us consider the insertion of a new vehicle for which an object `Manufacturer` already exists. The first instruction selects the identity of the manufacturer with the name of Ford in variable `m`; the second instruction then inserts a new vehicle made by that manufacturer:

```
m := select f
     from f in Company
     where Name = 'Ford'

insert new(Model, Manufacturer, Colour)
values ('Mondeo', m, 'red')
into Vehicle
```

The following closed expression achieves the same effect:

```
insert new(Model, Manufacturer, Colour)
values ('Mondeo', select f
                  from f in Company
                  where Name = 'Ford',
       'red')
into Vehicle
```

In principle, both expressions contain a *type error*, because the query defines a set (with one element) and the manufacturer of a vehicle is not set-valued. To avoid such situations, an appropriate *flatten*-operator should be available.

It is evident that insertion or modification operations must also be capable of generating complex objects such as lists. In other words, the modification operations of the data manipulation language must be able to use the constructors provided by the type system. And finally, modification operations may also have an effect on existing collections, as shown in Example 2.18:

Example 2.18

John Smith is to be employed in Ford's subsidiary in Chicago. In order to realize this modification the following procedure may be required:

(1) Insert `John Smith` into the class `Employee`.

(2) Select the object from the class `Subsidiary` of the company with the name Ford and the location 'Chicago' in its address.

(3) Insert the object `Employee` with name `John Smith` into the set `Employees` of this subsidiary.

2.2.9 Generating new classes, instances and operations

A data manipulation language of an object-oriented database must allow views to be defined on a given database state. Similarly to relational databases where relations are defined by query expressions, we can regard a set of values defined by a query expression as a view. In object-oriented databases, sets of object identities and extensions of classes can also be defined. Moreover, it should also be possible to define sets of objects *with* values. It must be possible to generate classes and their extensions either via the appropriate constructs of the definition language or *dynamically* by queries. Of course, it must be possible to use these classes in subsequent queries in the same way as the classes defined in the schema.

Example 2.19

Let us define a view in which person objects, their name values and vehicles manufactured by Ford are to be interrelated in such a way that for each tuple the name of the person and the name of the company's president correspond. For this purpose we introduce a new class K and assign to this class a set of tuples as described below:

```
create class K as
  select p, p.Name, f
  from p in Person, f in Vehicle
  where f.Manufacturer.Name = 'Ford'
        and p.Name = f.Manufacturer.President.Name
```

The result of this query is of the following type:

```
{[p: Person, Name: String, f: Vehicle ]}
```

Obviously, this type must be permissible for the object model in question.

Thus, the language used for defining views must not only be able to use the type constructors of the underlying object model but also contain the associated instance constructors. A distinction must be made depending on whether the outcome of the query is to consist of new objects or not. In accordance with relational views it is justifiable for views to introduce no new object identities, because they constitute derived data and objects: that is, view formation is *object-preserving*. But it is also tenable to define a view as *object-generating* in order to be able to realize, for example, inheritance on the objects in the view.

Note that in the previous example both variants would be justifiable. As an object-preserving feature, the identities of the persons could be used. However, this requires an operator to collect for each person all the corresponding vehicles into one set; in this case the type of the query would become {[p: Person, Name: String, Vehicles: {Vehicle}]}, where {Vehicle} is a set type with element type Vehicle. Thus the view would serve to expand the properties of the existing objects of class

Person dynamically. The definition of new objects requires the dynamic generation of object identities for objects that are to be included in the result of a query. There are at least two different ways to achieve this. One possibility is to generate new objects during the calculation of the query result: that is, to assign new identities to the elements of the resulting set. In our last example every generated instance of the resulting class of the type

```
{[ p: Person, Name: String, f: Vehicle ]}
```

would then be an independent object. A second possibility would be first to generate values which could later be converted into objects if required.

The dynamic generation of objects in queries leads to another problem, because it needs to be clarified how the new class is to be integrated into the already existing class hierarchy. Again, there are various approaches. On the one hand, it can be argued that results have no superclass, with the possible exception of a class Object which is the root of the given hierarchy; on the other hand, one can establish some rules which govern how the super- and subclasses of a view can be derived from the existing ones.

Finally, we should like to point out that the above-mentioned strategies for generating new classes and their instances can also be extended to cover operations, since generating new classes requires that class-specific behaviour can also be generated dynamically. The language used must allow for the introduction of new operations, for which there are at least two options: first, referencing existing implementations when new object types are generated, that is, resorting to operations previously formulated with a definition language; or, second, dynamically generating operations in queries as instances of a separate class Operation.

2.3 Navigating with path expressions

In object-oriented databases, if we want to process the overall structure of a compound object which is defined by aggregation, we have to follow references between objects. For this purpose typically a navigating technique based on path expressions is used. Now we are going to explain the relationship between path expressions and relational join operations and then, after having discussed some more examples, we shall outline some possible extensions of path expressions. We shall not go into formal details – this is postponed to Chapter 3.

Attributes and methods in object-oriented databases are formally scalar or set-valued functions. For example, an attribute which is defined for a class can be regarded as a function from the extension of the class to a set of values. Object-oriented databases take advantage of this functional relationship to develop a simpler syntax. We want to illustrate the relationship between join expressions and path expressions by an example based on the schema in Figure 1.8. Consider schemata for companies and vehicles defined as follows:

```
CREATE TABLE Company
     (CompanyID INT NOT NULL,
     Name VARCHAR NOT NULL,
     Street VARCHAR NOT NULL,
     Location VARCHAR NOT NULL,
     President INT NOT NULL,
     PRIMARY KEY (CompanyID),
     FOREIGN KEY (President)
     REFERENCE Employee (EmplNo)
     );

CREATE TABLE Vehicle
     (VehicleID INT NOT NULL,
     Model VARCHAR NOT NULL,
     Manufacturer INT NOT NULL,
     Colour VARCHAR NOT NULL,
     PRIMARY KEY (VehicleID)
     FOREIGN KEY (Manufacturer)
     REFERENCES Company (CompanyID)
     );
```

The president of the company manufacturing the car with number 93 can then be determined by the following SQL expression:

```
select c.President
from Vehicle v, Company c
where v.VehicleID = 93 and v.Manufacturer = c.CompanyID
```

This expression is evaluated in the following manner: first, all vehicles with number 93 are selected; then a join between these tuples and the company tuples is performed, joining tuples where the manufacturer of the vehicle is identical to the company under consideration; finally by projection all attributes of the tuples are deleted from the resulting set, except the attribute President.

In an object-oriented schema, however, two classes Vehicle and Company would have been defined (cf. Figure 1.8):

```
Company: [
      Name: String,
      Headoffice: Address,
      President: Employee ]

Vehicle: [
      Model: String,
      Manufacturer: Company,
      Colour: String ]
```

Let #93 be the object identity of the vehicle in question and assume that we can refer to this identity by variable v. Manufacturer, Headoffice and President are attributes whose values reference respective objects of classes Company, Address and

Employee. The attributes can be regarded as functions. If, for instance, function Manufacturer is applied to an object belonging to class Vehicle, it gives as the result the object identity of the respective company; accordingly, function President will give us for every company the object identity of its president. Manufacturer and President are functional in the sense that for every vehicle there is but one manufacturer and for every company there is but one president. If one takes into consideration that function Manufacturer gives precisely the object identity from which the president is to be determined, then the above SQL expression can be regarded as a composition of functions:

```
President(Manufacturer(v))
```

Note that this notation is only possible because the object with the identity #93 – in other words, the value of the variable v – belongs to class Vehicle and accordingly the result of the application of Manufacturer to #93 is the identity of an object of class Company. This ensures that the attributes Manufacturer and President can be applied in the desired manner. But also note that the above composition of functions references the employee who is president of the company, whereas the relational SQL expression supplies the employee number of the president. If the name of the president is of interest, the following adjustment becomes necessary:

```
Name(President(Manufacturer(v)))
```

Generally, object-oriented languages do not use expressions that have to be evaluated from the inside out, but rather expressions which allow evaluation from left to right. These expressions are called *path expressions*. For our example we obtain the path expression:

```
v.Manufacturer.President.Name
```

The dot-notation renders bracketing superfluous. In a path expression the variable which contains the identity of the object to be processed is positioned in front of the first dot. The name of the class to which the object belongs is often used as the variable name. In order to be able to distinguish between a class and a variable, variables are written here with a small initial letter. Moreover, in an SQL expression the variable can be omitted if only one class is contained in the from-clause and appears only once. We have already employed this short-hand notation in the examples above and, as a rule, will continue doing so.

To emphasize the usefulness of path expressions we present some more examples:

Example 2.20

The following query determines the names of the presidents of manufacturers producing blue vehicles.

```
select vehicle.Manufacturer.President.Name
from vehicle in Vehicle
where vehicle.Colour = 'blue'
```

As mentioned above, we do not need to include the variable `vehicle` in this case; instead we can write:

```
select Manufacturer.President.Name
from Vehicle
where Colour = 'blue'
```

The next query is to search for the names of presidents of companies with head offices in Detroit:

```
select President.Name
from Company
where Headoffice.Location = 'Detroit'
```

The following query is to search for vehicles whose manufacturer's name is the same as the name of the company's president:

```
select * from Vehicle
where Manufacturer.Name = Manufacturer.President.Name
```

And finally we show how the relational query in Section 1.4.1 can be expressed using path expressions. Remember that this query was presented to illustrate how awkward querying in relational databases might become. The task of the query is to find out whether an employee named Lacroix is employed in Ford's Ghent subsidiary. Using path expressions we can write:

```
select e
from e in Employee, c in Company, s in Subsidiary
where c.Name = 'Ford' and
      s in c.Subsidiaries and
      s.Office.Location = 'Ghent' and
      e in s.Employees and
      e.Name = 'Lacroix'
```

These examples illustrate that path expressions are indeed a compact and elegant technique for the tracing of relationships. Not only such rather aesthetic arguments, but also the aspect of efficiency advocates the use of path expressions. Applying references to objects in path expressions rather than using key values as in relational databases often enables more efficient query evaluation. Tracing relationships by means of calculating a join generally imposes a high system overhead, because keys have to be mapped to addresses by means of index structures. This may be avoided if references between objects are used instead. To a certain extent, tracing references corresponds to a *materialized* join in a relational database.

In principle, path expressions have to be *completely* specified. If one were to request the hp of an automobile, the following path expression would achieve the desired effect:

```
automobile.Drive.Engine.HP
```

However, if there is only *one* path in the schema graph connecting Automobile and Engine (cf. figure 1.8), then the path expression `automobile.HP` could be used as a short-hand notation.

A number of further generalizations of path expressions have been proposed, and we should like to present some thoughts on this topic. First, paths based on a subclass relationship can be followed in the reverse direction; thus in order to determine the hp of all the cars owned by a certain employee the following expression becomes possible:

```
employee.Fleet.Drive.Engine.HP
```

If we want to express explicitly that we are interested only in vehicles which are automobiles, we can insert variables into the path expression to produce this tie, as illustrated in the next example:

```
employee.Fleet[x].Drive.Engine.HP
```

In order to ensure that only vehicles of type Car are considered, for example, a where-clause may be included in the corresponding SQL expression:

```
select Fleet[x].Drive.Engine.HP
from Employee
where x in Automobile
```

Finally, as a further possible generalization, not only inheritance relationships but also aggregations may be followed in any direction. For example, a query determining the office location of a subsidiary where a particular employee works could be expressed as follows:

employee.Employees^{-1}.Office

Note that Employees is an attribute of class Subsidiary. With the notation Employees^{-1} we express that the relationship between subsidiaries and employees is to be traversed in reverse order for these employees. Therefore, the result of the above expression is the value of the attribute Office of those objects belonging to class Subsidiary and thus of objects belonging to class Address.

2.4 Inheritance

Now we come back to inheritance to look at some interesting aspects in more detail. We first recall the basic terminology we have introduced so far. Inheritance is based on a *hierarchy* of classes, which is implied by defining a class as a *specialization* or, using a different term, by an *IsA relationship*, of one or several other classes. The subordinate classes are called *subclasses* and the superordinate classes are referred to as *superclasses*. Subclasses *inherit* the structure and behaviour of their superclasses. The same attribute or method, but with a different type, respectively implementation, may be defined for different classes; names may thus be *overloaded*. If a subclass provides specific structure and behaviour, this will *override* inheritance. Sometimes certain values, so-called *default values*, or simply *defaults*, may also be inherited. On the other hand, *class attributes*, that is, attributes which are not applicable to individual objects but to the set of *all* objects, and class methods might *not* be inherited. Inheritance *conflicts* may arise if *multiple inheritance* takes place. If a message is sent to an object, then according to the rules of inheritance, a class to provide the attribute, respectively method implementation, will be selected. The selected class is called the *receiver class* of the message. The actual receiver class corresponding to an object and a message can be decided only at run time; *late binding* of attributes and method implementations to objects is therefore required.

If for two classes K_1, K_2 we have K_1 IsA K_2, we consider every object in class K_1 as belonging to class K_2 as well. From this it follows that an IsA relationship expresses a subset relationship between class extensions, implying that an object in general is an element of several class extensions. However, for each object there exists a unique class to which it is primarily assigned; this class is called the *base class* of the object. When we talk about *the* class of an object, we refer always to its base class. If there are one or several direct or indirect superclasses of a base class then – as a consequence of the subset relationships – every object of the base class is also an object of each of its superclasses. This, however, is to be understood in a *logical* sense as meaning that the object can be treated as an object of the superclass as well.

In this section we shall examine interesting inheritance issues in greater detail. Examples are based on the scenario in Figure 1.8, but we shall expand or vary it if necessary. The following discussions will be at a more informal level – a formal treatment is postponed to Chapter 3.

2.4.1 Reuse by inheritance, redefinition and conflicts

Let us consider the following definitions for the classes Person, Employee, Manager and Shareholder, including their IsA relationships (Figure 2.1):

Classes Employee and Shareholder inherit all the properties of class Person according to the IsA relationships stating that every object of classes Employee and Shareholder is also an object of class Person. Consequently, the attributes Name, Age, Domicile and Fleet of class Person are also applicable to classes Employee and Shareholder. This implies that each object belonging to the classes Employee or Shareholder has values for these attributes without having to define the attributes within those classes; in other words, we see *reusability of type information*.

Let us now define a *default value* for the employee's salary:

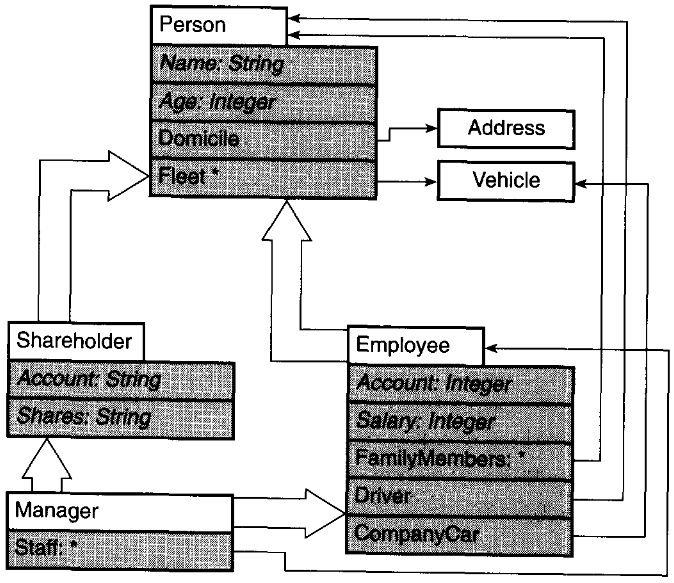


Figure 2.1 A nontrivial case of inheritance.

```
Employee IsA Person: [
... ,
Salary: integer (value: 1000) ]
```

This default has the effect that for each object of class Employee a salary of 1000 dollars is automatically assigned, unless a different salary for specific employees is explicitly defined. Here we see *reusability of values*.

When specific properties of subclasses have to be defined, a *redefinition* of inherited properties occurs. For example, we may define an additional attribute Driver for class Manager:

```
Manager IsA Employee, Shareholder: [
... ,
Driver: Employee ]
```

The redefinition should be valid for all objects belonging to class Manager. As Driver is also defined for superclass Employee, this property is *overwritten* by the redefinition.

Class Manager, unlike Employee and Shareholder, has two immediate superclasses, namely Employee and Shareholder. We have *multiple inheritance*. In principle, Manager inherits the attributes of all superclasses, including the indirect superclass Person. But because Employee and Shareholder already inherit the attributes of Person, it is sufficient to consider only the immediate superclasses. Two problems can be observed:

(1) For Employee and Shareholder attribute Account is defined with a different type; the name Account is *overloaded*. Which of the two possible versions of Account is to be inherited by Manager? Or should both versions be inherited? We are facing an *inheritance conflict.*

(2) The second problem is of a more indirect nature. Both Employee and Shareholder inherit the attributes from Person. If these attributes were to be inherited by the class Manager, the question arises whether they should still be considered properties of Person or whether, in principle, they should be considered as different properties of Employee and Shareholder. The latter case again causes an *inheritance conflict*, whereas the former, which seems to be appropriate in most situations, does not create any difficulties.

There are several approaches to the issue of inheritance conflicts. One possibility would be to assign priorities, which could be defined, for example, depending on the order in which IsA relationships appear in class definitions. We have written:

```
Manager IsA Employee, Shareholder
```

We can interpret this in such a way that if an inheritance conflict occurs the definitions of Employee have priority over the definitions of Shareholder. Another frequently encountered technique is based on *from-clauses* which allow us to define which attribute is to be inherited from which class. We could write the following:

```
Manager IsA Employee, Shareholder: [
      Staff: {Employee},
      Account inherited from Employee]
```

This notation gives us more flexibility in handling inheritance conflicts. These conflicts can be avoided by restricting the inheritance hierarchy. In our example we can achieve this by introducing an additional class EmplSha which would act as a superclass of Manager. Next we redefine Account for EmplSha and thus override the definitions of Employee and Shareholder. If Account is to be valid for Manager in the same way as it is defined for Employee, the following definitions are in order:

```
EmplSha IsA Employee, Shareholder: [
Account: Integer]

Manager IsA EmplSha: [
Staff: {Employee}]
```

In principle, inheritance of methods involves the same problems as inheritance of attributes. Let us look at class Employee, for example. The methods which can be applied to an object of this class are either directly attached to this class or are a result of inheritance. Let us further suppose that MoreSalary is a method intended to effect a rise in salary. If a message is sent to an object of class Employee to execute the method MoreSalary, Employee will be consulted first in order to determine the in a program or query where we expect to find an object of class K_2, for example as the value of a variable, an object of class K_1 is also permissible. Accordingly, an object of a superclass should be *substitutable* by objects of its subclasses. This requires that the types attached to classes K_1, K_2 must relate to each other accordingly; that is, it should be impossible for a type error to occur at run time. Evidently, objects of a subclass can replace objects of a superclass only when all the properties of the superclass are also defined for the objects of the subclass. Thus the type of the subclass must be a subtype of the type of the superclass in an appropriate sense. Using a more technical term, what we require is a form of *polymorphism* which says that variables may be bound by values of different type.

To each class we assign a *type*, which, in particular, defines the attributes applicable to the objects in the respective extensions and also the allowed values. Every attribute represents a structural property of the objects and therefore has itself a type attached to it. Let us consider the type assigned to class Person:

```
[ Name: string,
Age: integer,
Domicile: Address,
Fleet: {Vehicle} ]
```

The type assigned to Person is an example of a *tuple type*; this means that all objects of this class are represented by tuples and that the individual components of the tuples are values of the type assigned to the respective attributes. Name and Age have base types, whereas Domicile is of a *reference type* and Fleet is of a *set type*. A reference type is indicated by a class name; for every employee the address component is a reference to an object of class Address. The set type Fleet is defined over the *element type* Vehicle. Again, the element type is indicated by a class. Thus, for every employee the fleet component is a set of references to objects of class Vehicle. We can see that a type is assigned to a class in order to describe the structures of its objects, and that a class is itself a type for references to objects.

Now we are ready to say more precisely what we expect from a subtype in order to achieve substitutability. A type T' is a *subtype* of a type T and vice versa T is a *supertype* of T', if either $T = T'$ or the conditions stated below apply.

- If T is a class, then every subclass T' of T is a subtype of T. For example, Employee, Manager and Shareholder are all subtypes of Person, and Manager is also a subtype of Employee and Shareholder.
- If T is a tuple type, then T' is a subtype of T, if T' is also of tuple type and every attribute in T is defined for T' as well. Moreover, for corresponding attributes the T'-type must be a subtype of the T-type. For example, if T is the type of class Employee, then every type containing additional attributes is a subtype. Specifically, as a consequence of inheritance the type of Employee is a subtype of the type of Person and the type of Manager is a subtype of the types of Person and Employee. Furthermore, the type of Manager remains a subtype if attribute Driver is explicitly defined for Manager with type Employee, because class Employee is a subtype of class Person.

- If T is a set type, then T' is a subtype of T, if T' is also a set type and the element type of T' is a subtype of the element type of T. For example, set type {Employee} is a subtype of {Person}.

Example 2.21

Assume first the situation as depicted in Figure 2.1. By inheritance, all attributes of class `Employee` are also defined with the same type for class `Manager`. Therefore, the type of `Manager` is a subtype of the type of `Employee`. Let us illustrate substitutability using the following query, which determines all employees who have the same domicile as their drivers:

```
select a
from a in Employee
where a.Domicile.Location = a.Driver.Domicile.Location
```

Since every manager is also an employee and no redefinition of attributes occurs, all attributes of `Employee` are inherited by `Manager`. Wherever we expect an employee, a manager might occur. This means that we can bind variable a by an employee object as well as by a manager object. This substitutability enables us to use `Domicile` on the left-hand side and `Driver` on the right-hand side of the equation in the where-clause and to apply them either to an object of class `Manager` or to an object of class `Employee`, depending on the content of variable `a`.

Assume now that `Driver` is redefined in class `Manager` by type `Employee`. Does substitutability still hold? The answer is yes, because `Employee` is a subtype of `Person` and therefore `Domicile` will be defined for `a.Driver` independently of a being an employee or a manager.

This example shows that substitutability allows us to state queries in a concise form. Substitutability of objects and redefinition of methods requires *late binding*. We shall illustrate this in the next example.

Example 2.22

We should like to compute an indicator for the success of employees; however, for managers a different algorithm must be applied. We therefore assign two different implementations of a method `SuccessIndicator` to classes `Employee` and `Manager`. The signature

```
SuccessIndicator(Employee): Integer
```

is assigned to class `Employee`. We redefine the signature for class `Manager`:

```
SuccessIndicator(Manager): Integer
```

The following query determines all employees, and in particular all managers, whose SuccessIndicator is greater than 50:

```
select a
from a in Employee
where a.SuccessIndicator > 50
```

When processing this query different implementations of the method SuccessIndicator must be used for the objects of classes Employee and Manager. Therefore, it is not possible to provide a *static* binding of *one* implementation of SuccessIndicator to the variable a at compile time; instead, a *dynamic* binding at run time is required.

2.4.3 Safety

A language can be classified as *safe* when for each program or query, it is possible for the compiler to decide whether one of the error situations listed below might occur during execution:

- A type restriction defined in the schema is violated. For example, values of the wrong type are assigned to attributes, results of method calls differ from the defined result type, comparison operations are applied to values of a different type, or method calls have arguments of the wrong type.
- An undefined property of an object is referred to; for instance, a query needs an attribute which is not defined, or a method is called for which no signature is defined.

Safety is an important feature of programming languages in general, because it allows errors to be detected at compile time which otherwise might occur at some later time and cause an expensive correction to become necessary.

Example 2.23

The following query contains an obvious type violation, because the type of Fleet is different from the type of FamilyMember:

```
select a
from a in Employee
where FamilyMember = Fleet
```

Note that in this case the properties to be inherited must be defined first in order to check safety; the attribute FamilyMember is inherited from class Person by Employee.

Type violations of this kind will always occur, unless we are dealing with the trivial case where the database does not contain an object of class Employee. Therefore, the query must be rejected. The next query, however, contains only a potential type violation.

Example 2.24

Here the attribute `Account` is not defined for persons in general, but only for a subset of persons, namely `Employees`.

```
select Account
from Person
where Domicile in
(select Headoffice from Company)
```

When processing this query, a type violation will occur as soon as a person is found who is not an employee or a shareholder and whose domicile is identical to the head office of a company in the database.

We are now interested in the question of whether or not Account will be defined for the objects considered in the query. Unlike the query in Example 2.23, there exist interesting states of the database in which such type violations will not occur. If every person is an employee or a shareholder, or if only employees or shareholders have their domicile at the same location as the head office of a company, then only objects which are employees or shareholders, for which the attribute Account is defined, will be considered. It is evident, however, that this query must also be rejected if safety is to be guaranteed by the compiler, because information about states usually cannot be taken into account. In addition, Account is defined for Employee and Shareholder with different types, which is a further source of type violation.

Example 2.25

The next query illustrates (cf. Example 2.13) that a variable of class type, in this case `p` which is typed by class `Person`, can always be used for referencing objects of the subclass of its type, in this case `Employee` and `Manager`.

```
select *
from p in Person
where (p in Employee or p in Manager) and p.Age > 50
```

This query determines all persons who are employees or managers and are over 50. Since `p` is of type `Person`, all objects considered have the properties defined for persons. It is interesting to see that this form of query is much more compact than processing both subclasses separately, but in an almost identical manner.

Note that the reverse case, where objects of a superclass are referenced by a variable of the type of one of its subclasses, will in general result in a type violation. This is due to the fact that properties defined solely for a subclass might now be accessed for an object which belongs to the superclass and not to the subclass.

The following examples are concerned with safety in the case of redefined attributes and methods. We do not explicitly consider set types because analogous arguments apply.

Example 2.26

Let us consider the situation where attribute `Driver` is redefined for class `Manager`:

```
Employee IsA Person: [
... ,
Driver: Person,
... ]

Manager IsA Employee, Shareholder: [
... ,
Driver: Employee]
```

and the query:

```
select Domicile
from p in Person
where p in
(select Driver from Employee)
```

`p` is a variable of type `Person`. The inner select-expression may, however, supply a heterogeneous set containing objects of type `Person` and type `Employee`, because managers are also employees, and attribute `Driver` of type `Employee` applies to them. Yet, in spite of such heterogeneity we are not dealing with a type violation, because all employees are persons as well and thus all properties of `Person` – particularly `Domicile` – apply equally to `Employee`.

The next example shows that redefining methods may be problematic.

Example 2.27

We are interested in a method `Deputy`, which applies to classes `Employee` and `Manager`. This method allows us to identify employees and managers who may act as deputies for a given employee, respectively manager. Let us further assume that a different algorithm is required for managers than for normal employees. We shall consider the two following signatures for method `Deputy`:

```
Deputy(Employee, Employee): Boolean
Deputy(Manager, Manager): Boolean
```

The condition to be fulfilled by employees `a1` and `a2` to enable them to be deputies of each other are as follows (where `Denomination` is a new property of employees):

```
a1.Denomination = a2.Denomination and not (a1 == a2)
```

Additionally, for managers `m1` and `m2` we take into consideration their party membership (another new property of employees):

```
m1.Denomination = m2.Denomination and
m1.Party = m2.Party and not(m1 == m2)
```

Now, if we want to determine the deputies of all employees, the following query seems an obvious choice:

```
select a1, a2
from a1 in Employee, a2 in Employee
where a1.Deputy(a2)
```

However, this query is not safe, because if a1 references an employee who is also a manager, the implementation of `Deputy` attached to `Manager` will be executed. On the other hand, if a2 references an employee who is not a manager, an argument of the wrong type will be passed and thus an attempt will be made to access the non-defined attribute `Party`.

These examples show that redefining properties can be done to a limited degree only, otherwise safety can no longer be guaranteed. In both the preceding examples we took the intuitive approach to redefining attributes and signatures by choosing subtypes of the original types. With respect to attribute Driver, we redefined Person by Employee; with respect to method Deputy, we redefined Employee by Manager. In the first case, redefinition did not affect safety, but in the second case safety was violated. Obviously, the situation is *not* symmetrical for attributes and methods. This can be explained as follows (for the present, however, ignoring update operations and therefore restricting ourselves to queries):

(1) Let us examine two classes ClassA and ClassB and an attribute Attr, which is defined in both classes, and let us further assume that ClassB is a subclass of ClassA. Thus the attribute Attr is redefined in ClassB and we can write the following:

```
ClassA:
[ ...
Attr: classC,
...]
```

```
ClassB IsA ClassA:
[... ,
Attr: ClassD,
...]
```

Assuming that a is a variable of type ClassA, the following question is of interest: when is it permissible for variable a to reference objects of ClassB?

Assume variable a references an object of ClassB. If ClassD is a subclass of ClassC (as shown in Figure 2.2), then a.Attr will always supply an object which has at least the properties required by an object from ClassC. Therefore, redefinition cannot lead to a type violation in this case. If ClassD, however, is a superclass of ClassC, type violations generally will occur during the execution of a query referring to objects of ClassA. For example, the expression `a.Attr.Attr'` will lead to a type violation as soon as a references an object of class ClassB and Attr' is an attribute which is defined only for ClassC.

We may conclude that the type of a redefined attribute must be a subtype of the type of the original attribute.

(2) We shall now look at the redefinition of methods. For this purpose we shall consider two classes ClassA and ClassB and a method Meth for which both classes contain a signature. Assuming that ClassB is a subclass of ClassA we can write the following:

```
ClassB IsA ClassA
Meth(ClassA, Type11, ..., Type1k): TypeA
Meth(ClassB, Type21, ..., Type21): TypeB
```

Let us suppose that a is a variable of type ClassA. Again it is of interest to establish when it is permissible for variable a to reference objects from

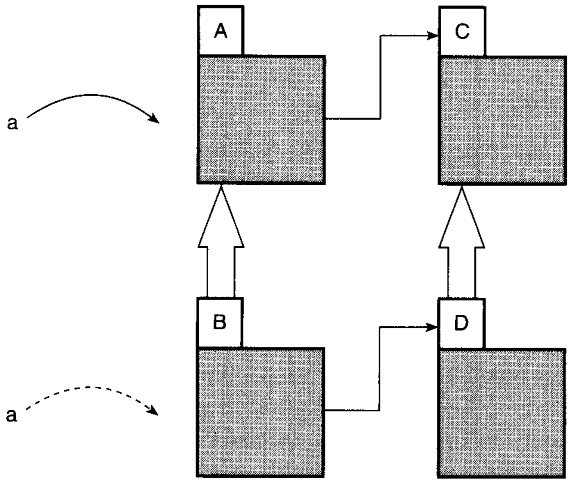


Figure 2.2 Safety and redefinition of attributes.

ClassB as well. As a first obvious restriction the number of arguments in both signatures must be equal: that is, k = l.

Let a reference an object belonging to ClassB. With respect to the result, we can apply the same arguments as we did under (1) and come to the conclusion that TypeB must be a subtype of TypeA.

Now the relationship between the argument types is of interest. Our example has shown that the argument types of the redefining signature cannot be strict subtypes. Since variable a is of type ClassA, one can at best guarantee the restrictions of the types of the arguments of the signature attached to ClassA. From this it follows that the types of the arguments of the redefining signature must be supertypes of the arguments of the redefined signature.

Consequently, the conditions for redefining methods somewhat oppose our intuition, because the signatures attached to the more specific class must have argument types which are more general than or equal to those of the signatures attached to the more general classes. Only then can safety be guaranteed by a compiler.

Finally, let us turn to update operations with respect to attributes. Here we have to learn that whenever updates are allowed, redefining the types of attributes cannot be permitted. To see this let us suppose that a is a variable of type ClassA, where ClassB is a subclass of ClassA. If a references an object of ClassB, it may happen that a value of the type of the respective attribute with regard to ClassA will be assigned to an attribute which is redefined in ClassB. The type of the redefining attribute then must be equal to, or a supertype of, the type of the redefined attribute. Considering the conclusions we reached when discussing queries, namely that the type of the redefining attribute must be a subtype of the type of the redefined attribute, it follows that redefining and redefined types must be identical. But this implies that whenever updates may occur attributes must not be redefined if safety is to be guaranteed.

2.4.4 Discussion

A class hierarchy combined with a mechanism for inheritance enables the reuse of properties. In particular, the possibility of reusing methods renders object-oriented databases attractive to software engineers. Yet, we have also seen that one important goal, namely (type) safety, is not always compatible with the notion of reusability. With respect to this matter we should like to discuss two further aspects.

First, the conflict between establishing inheritance and guaranteeing safety is a problem for object-oriented languages in general and not a problem that specifically affects object-oriented *databases*. Second, it is mostly in very particular situations that safety can no longer be guaranteed. It appears that in real situations this problem occurs only in instances of a special kind of recursion where the class whose method is under consideration itself acts as an argument type. In such cases one intuitively understands that class specialization requires specialization of the type of the argument as well; but it is precisely this procedure that jeopardizes safety.

On the other hand, it is hard to see how for methods such as the already mentioned MoreSalary with the signature

```
MoreSalary(Employee, Integer): Boolean
```

a specialization of the arguments can be required by the specialization of the class. It is scarcely conceivable that the argument type Integer should be specialized merely because Employee has been specialized to Manager.

Moreover, we have described inheritance as a *monotonic* mechanism where subclasses may override an inheritance but cannot actually refuse it. Therefore, all properties of a superclass are also properties of their subclasses. Such monotonicity is merely a natural consequence of the IsA relationships between the classes. However, there are also examples where monotonicity renders adequate modelling more difficult.

Example 2.28

We want to manage geometrical objects with the help of a database. Let us consider two classes `Rectangle` and `Square`, with squares being specializations of rectangles, since every square is by nature also a rectangle. For these classes the following definitions seem appropriate:

```
Rectangle: [
Centre: Point,
SideLength: Integer,
SecondSideLength: Integer]

Square: [
Centre: Point,
SideLength: Integer]
```

If we establish for square and rectangle the obvious IsA relationship `Square IsA Rectangle`, then inheritance is not in order, because squares have fewer properties than rectangles and not more. This is contrary to our previous assumptions, which were based on the concept of monotonicity of inheritance for subordinate classes.

If inheritance is to be realized, we must reject the intuition of an IsA relationship and write the following:

```
Square: [
Centre: Point,
SideLength, Integer]

Rectangle IsA Square: [
SecondSideLength: Integer]
```

This type of problem, too, is *not* inherent to object-oriented databases, but characteristic of object orientation in general. In order to achieve more flexibility when operating an inheritance mechanism, some object-oriented languages, such as *Smalltalk*, define the underlying class hierarchy separately from the IsA relationships. When working with databases such a procedure may be questioned, because an IsA relationship is an essential abstraction concept in the design phase of a database and the resulting abstractions should not be so readily abandoned in the later stages of software development.

Inheritance is not the only approach to achieving reuse of structure and behaviour; there is another, rather different technique, which is commonly known as *delegation*. As delegation is independent of a class hierarchy, it can be useful in situations where the desired effect of reuse could not be achieved by inheritance in an intuitive way. To demonstrate this we shall show by means of an example how multiple inheritance can be simulated by single inheritance. In fact, there exist object-oriented database systems, for example, Gemstone Smalltalk (cf. Section 4.4), which do not support multiple inheritance.

Let us look again at Figure 2.1 and assume that multiple inheritance is not supported. We thus have to find a representation based on single inheritance. Assume we define Manager to be only a subclass of Employee:

```
Manager IsA Employee: [
Staff: { Employee }]
```

To make it possible to query the shares of a manager we extend the above definition of Manager as follows. First we add a private attribute AsShareholder with type Shareholder. Whenever we create a Manager object we now have to create a Shareholder object as well; this shareholder object implements the role of the corresponding manager object with respect to its shareholder properties. Note that here we reuse all properties of class Shareholder for our purposes. To make the shares of a manager accessible, we next add a public method MyShares to Manager. To implement this method we make use of delegation: any call to MyShares will delegate this call to the corresponding AsShareholder object by executing the path expression `asShareholder.Shares`, for example. Taking these extensions into account we come up with the following definition of class Manager:

```
Manager IsA Employee: [
AsShareholder: Shareholder,
Staff: { Employee },
MyShares: Integer]
private: AsShareholder
```

2.5 Bibliographical notes

Requirements for and the properties of languages of object-oriented databases can be found in Bertino and Martino (1993), Kemper and Moerkotte (1994) and Loomis (1995), amongst others. In Section 2.2 we based our arguments on Kim (1990) and

Manola (1991). The use of path expressions in connection with object-oriented databases was first explored by Zaniolo (1983); examples of SQL variants using path expressions are Kifer et al. (1992), Kim (1990) and Bancilhon et al. (1992). Generalizations of path expressions, particularly abbreviating mechanisms, are discussed in Van den Bussche and Vossen (1993) and in Sciore (1994). Our discussion of (type) safety is based on Kemper and Moerkotte (1994). There is also much literature devoted to the development of object-oriented systems, for example Rumbaugh et al. (1991) and Booch (1991).

3 A formal framework for structure and behaviour

In this chapter we present a formal framework for object-oriented databases, providing more details on the concepts underlying the intuitive discussions in the previous chapter. In particular, we define the concept of *object base schema* and *object base*. We conclude the chapter with a formal treatment of path expressions applying the newly introduced formalism.

In order to make a clear distinction between the various aspects of object-oriented databases, an object base schema SC will be further divided into a *structure schema* SC_{struc} and a *behaviour schema* SC_{behav}. SC_{struc} defines the types and classes, whereas SC_{behav} is mainly responsible for attaching methods to classes. An object base $d(SC)$ then consists of class extensions and method implementations, according to the definitions and restrictions stated in SC_{struc} and SC_{behav}.

3.1 Modelling of structure

First we want to examine the modelling of the structural aspects of object-oriented databases, beginning with the definition of values, objects and types and then proceeding to classes, class extensions, class hierarchies and inheritance.

Let us assume that **A** is a set of *attribute names* and **O** is a set of *object identities*, where each object identity consists of the symbol # followed by a positive integer number. In order to simplify matters, we shall mostly use the short forms attribute and object instead of talking about attribute *name* or object *identity*. Let further nil $\in$ **O** be the *empty* reference. The set of allowed values then can be defined inductively in the following manner:

Definition 3.1

Let **D** be the set of values composed of the union of integer numbers, floats, strings, and boolean values.

(i) All elements in **D** are values. (*atomic values*)

(ii) All object identities in **O** are values. (*reference values*)

(iii) Let $n \geq 0$, and let $A_i \in$ **A** be distinct attributes and w_i values, $1 \leq i \leq n$, then all expressions are also values if they comply with the following structure:

(a) $[A_1\!: w_1,..., A_n : w_n]$; subexpressions of the form $A_i : w_i$, $1 \leq i \leq n$ are called components. (*tuple values*)

(b) $\{w_1,... w_n\}$ where w_i distinct, $1 \leq i \leq n$. (*set values*)

Let **W** be the set of all *complex values* defined in this way.

The order of the components in a tuple value is irrelevant. If necessary, we treat a tuple value as the set of its components. In particular, we consider two tuple values as being identical, if they possess identical components, but possibly in a different order. We shall admit tuple values of the form [ ] and denote them as *empty tuple* values. A set value of the form { } denotes an *empty set*, which we also write as usual; that is, $\varnothing$. In order to simplify matters when developing our formal setting, we shall disregard lists, bags, and so on as they can be dealt with similarly to sets. According to the definition given above, object identities are also values, which is necessary because references between objects are to be expressed.

In terms of structure we can now represent an object by means of an object identity and a value:

Definition 3.2

An *object* is a pair (o, w), where o is an object identity in **O** and w is a value in **W**.

Example 3.1

Let us look at some objects in the context of our running example (Figure 1.9). The object with the identity #1 represents information about vehicles and persons which are summarized in a tuple value. `Vehicles` denotes a set of reference values each referring to a concrete vehicle, and `Person` denotes a set of tuple values. The component `Fleet` of each of these values is itself a set value of references. Note how the use of reference values, that is, object identities, avoids redundancy. Since every vehicle is an object with a unique identity, we can restrict ourselves to the respective object identities.

```
(#1,[Vehicles:{#10, #11, #12},
  Persons: {[Name: 'Slim', Age:40, Domicile:#80, Fleet: {#10, #11}]}])

(#10, [Model: 'Golf', Manufacturer: #41, Colour: 'red'])
(#11, [Model '323', Manufacturer: #42, Colour: 'blue'])
(#12, [Model: 'R4', Manufacturer, #43, Colour: 'green'])

(#41, [Name: 'VW', Headquarters: #80,...])
(#42, [Name: 'Mazda', Headquarters: #81,...])
(#43, [Name: 'Renault', Headquarters: #82,...])
```

Making a distinction between an object's identity and its value has further consequences than merely making objects distinguishable independently of their values. Distinguishing between identity and value allows us to maintain several copies of the same objects: that is, objects with different identities but with identical values. Consequently, we have already established two different equality operators: '==' for *identity of objects*, that is, equality of object identities, and '=' for *equality of values* (cf. Section 2.2).

Sometimes it is not sufficient to make a distinction solely between identity and equality. Let us exemplify this by considering two tuple values $w_1 = [A: o_1]$, $w_2 = [A: o_2]$ where o_1 and o_2 are object identities and $o_1 \neq o_2$, and consequently $w_1 \neq w_2$. If the objects referenced by o_1 and o_2 have the same value, for example (o_1, w) and (o_2, w) respectively, then $w_1 \neq w_2$ applies, even though substituting o_1, o_2 by w would make both objects equal. One solution to this unsatisfactory situation is to distinguish between *shallow equality*, that is, equality of values without substituting references by the corresponding values, and *deep equality*, that is, equality only after recursive substitution of references by their values. It is easy to see that the following implications hold true: identity of objects implies shallow equality and shallow equality implies deep equality. Of course, the reverse implications do not generally apply.

Values of a similar structure are characterized by a common type and object identities of similar objects by a common class. Object identities are also values (reference values) and therefore a class is also a type. Later we shall assign a type to every class in order to define the permissible values for the objects of a class. Types and classes are therefore recursively related: a class may act as a type in order to type references among objects; on the other hand, a type may be attached to a class when the structure of its objects is to be defined.

Given a set of class names, we shall first define types and class hierarchies. According to the underlying hierarchy, types of classes are interrelated. This will give rise to a subtype ordering. After these definitions we are able to introduce a structure schema. Then we shall define extensions of classes and value sets of types. Having done all this we will finally be able to define an instance of a given structure schema. The formal framework so far will not reflect inheritance; this will be the final step.

Definition 3.3

Let **K** be a finite set of *class names*. The set of all types over **K** is defined inductively as follows:

(i) integer, float, string, and boolean are types. (*base types*)

(ii) Every class in **K** is a type. (*reference type*)

(iii) If $A_1, ..., A_n \in \mathbf{A}$ are distinct attributes and $T, T_1, ..., T_n$, $n \geq 0$ are types, then every expression which conforms to one of the following is also a type:

(a) $[A_1 : T_1, ..., A_n : T_n]$; subexpressions of the form $A_i : T_i$, $1 \leq i \leq n$ are called components. (*tuple type*)

(b) $\{T\}$ (*set type*)

Let **T**(**K**) be the set of all *complex types* defined in that way. Let $A : T$ be a component of a tuple type. If T is a set type, then the attribute A is said to be *set-valued*, otherwise A is said to be *scalar*.

Analogous to a tuple value, for a tuple type the order of the components is irrelevant; a tuple type can also be considered as the set of its components, if appropriate. A tuple type of the form [] is called an *empty tuple type*. Furthermore, we may write **T** instead of **T**(**K**) if this short-hand notation does not imply ambiguities.

Example 3.2

Here we give the type definitions corresponding to the values presented in Example 3.1. `Vehicle` and `Company` are classes. Later we shall see how extensions are attached to classes, which will then allow us to ascertain that the above values are indeed of the desired type.

```
[Vehicles: {Vehicle},
           Person: {[Name: String, Age: Integer,
           Domicile: Address,
           Fleet: { Vehicle}]}]

[Model : String, Manufacturer : Company, Colour: String]

[Name : String, Headquarters : Address,...]
```

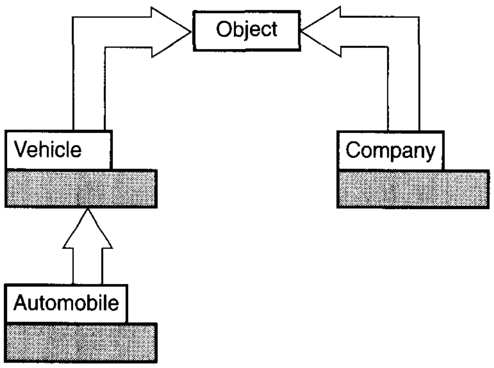


Figure 3.1 Graphic representation of a class hierarchy.

Classes are arranged in a hierarchy, which is based on the IsA relationships between the participating classes. If a class K is subordinate to a class K', then every object of K is also an object of K'. A class hierarchy of this kind is the basis for the inheritance of properties, and we have already seen how IsA relationships between classes imply corresponding *subtype* relationships between types of the classes involved. In order to introduce subtypes we need some preliminaries.

Let **K** be a set of class names and *object* be a specific class name in **K** such that all other classes are subclasses of class *object*.

Definition 3.4

A class hierarchy with respect to **K** is a partial order (that is, a reflexive, antisymmetrical and transitive relation) *IsA* on **K** such that *K IsA object* for every class $K \in \mathbf{K}$. If *K IsA K'*, then K is a *subclass* of K' and accordingly K' is a *superclass* of K.

Class hierarchies are often presented graphically (Figure 3.1).

Example 3.3

In addition to the hitherto used classes `Vehicle` and `Company`, let us consider a class `Automobile`, which is a subclass of `Vehicle`. We then obtain the graph shown in Figure 3.1 with the obvious directions of the edges.

A given class hierarchy *IsA* induces a *subtype order* $\leq$ in accordance with the definition presented below. Analogous to the role of the class *object* within a class hierarchy, *any* is a type of which every other type is a subtype.

Definition 3.5

Let us assume a set **K** of class names, a set **T** of types and a class hierarchy *IsA*. The *subtype order* $\leq$ on **T** is the smallest partial order which is closed with respect to the following rules:

(i) If K *IsA* K', then also $K \leq K'$ for $K, K' \in \mathbf{K}$.

(ii) Let $T = [A_1 : T_1, \ldots, A_n : T_n]$, $T' = [A'_1 : T'_1, \ldots, A'_m : T'_m]$ be two tuple types in $\mathbf{T}$. If $n \geq m$ and if for every component $A'_i : T'_i$, $1 \leq i \leq m$ of T' there exists a component $A_j : T_j$, $1 \leq j \leq n$ of T where $A'_i = A_j$ and $T_j \leq T'_i$, then $T \leq T'$ also holds true.

(iii) Let $T, T' \in \mathbf{T}$. If $T \leq T'$, then also $\{T\} \leq \{T'\}$.

(iv) $T \leq any$ for all $T \in \mathbf{T}$.

In (i) a subtype order for classes is defined which is consistent with the class hierarchy. The condition stipulated in (ii) states that two tuple types T and T' are included in the relation $T \leq T'$, if all attributes of T' (possibly in a different order) occur in T as well and types of attributes with the same name are subtypes of one another recursively; T may contain new attributes not occurring in T'. Note that according to this definition base types such as integer or string are incomparable with respect to '$\leq$'. The conditions stated in (iii) and (iv) require no further explanation.

Having introduced these terms we now can define a structure schema, which will assign types to classes according to the respective class hierarchy:

Definition 3.6

A *structure schema* has the following form:

$$\mathrm{SC}_{struc} = (\mathbf{K}, IsA, type)$$

$\mathbf{K}$ is a finite set of class names, *IsA* a class hierarchy and type: $\mathbf{K} \rightarrow \mathbf{T}$ is a mapping which assigns a type to every class. Structure schemas are *well formed* in such a way that *type* has the following property:

$$K\ IsA\ K' \Rightarrow type(K) \leq type(K')$$

We now want to define an object base with respect to its structural component $d(\mathrm{SC}_{struc})$, in a way which reflects SC_{struc}. An object base contains objects whose classes are ordered in a hierarchy and which have a possibly structured value. Let us begin by assigning objects to classes. The corresponding mapping will interpret *IsA* according to the intended semantics by a subset relationship; moreover, every object identity is attached to a selected class – its *base class*.

Definition 3.7

A *base extension* is a mapping *inst* which assigns pairwise disjoint finite sets of object identities to the class names in $\mathbf{K}$. If $o \in inst(K)$ for $K \in \mathbf{K}$, then K is called the *base class* of o.

An *extension* for a given mapping *inst* is a mapping *Inst* which assigns sets of object identities to the class names $K \in \mathbf{K}$ taking into account the class hierarchy:

$$Inst(K) = inst(K) \cup \{\, o \mid o \in inst(K'), K' \in \mathbf{K}, K'\ IsA\ K \}$$

Example 3.4

Let us restrict ourselves to the classes `Vehicle` and `Automobile`. By means of *inst* disjoint sets of object identities are attached to these classes. *Inst* then takes the class hierarchy into account; thus with regard to the class hierarchy represented in Figure 3.1 we have the following:

$inst(Object) = \emptyset$
$inst(\text{Vehicle}) = \{\#13\}$,
$inst(\text{Automobile}) = \{\#10, \#11, \#12\}$,
$Inst(Object) = \{\#10, \#11, \#12, \#13\}$,
$Inst(\text{Vehicle}) = \{\#10, \#11, \#12, \#13\}$,
$Inst(\text{Automobile}) = \{\#10, \#11, \#12\}$.

Now we can link types to value sets:

Definition 3.8

A mapping *dom* which attaches to every type a corresponding value set (domain) is defined as follows:

(i) *dom*(integer) is the set of all integer numbers (analogous for string, float, boolean);

(ii) $dom(K) = Inst(K)$ for every class $K \in \mathbf{K}$;

(iii) For a tuple type $T = [A_1 : T_1, ..., A_n : T_n]$, $n \geq 0$, we have

$$dom(T) := \{[A_1 : w_1, ..., A_n : w_n] \mid w_i \in dom(T_i), 1 \leq i \leq n\};$$

(iv) For a set type $\{T\}$ we have

$$dom(\{T\}) := \{\{w_1, ..., w_n\} \mid w_i \in dom(T) \text{ for } n \geq 0, 1 \leq i \leq n\}.$$

Finally, we can attach values to objects according to the types of their classes, which allows us to define the structural part of an object base. However, this definition still does not reflect inheritance.

Definition 3.9

Let $\text{SC}_{struc} = (\mathbf{K}, IsA, type)$ be a structure schema, $\mathbf{O}$ the set consisting of all object identities and $\mathbf{W}$ the set of all values. An *instance* (without inheritance) of SC_{struc} is given by:

$$d(\text{SC}_{struc}) = (inst, val).$$

Here *inst* is a base extension and val: $\mathbf{O} \rightarrow \mathbf{W}$ a mapping with the property:

$$o \in inst(K), K \in \mathbf{K} \Rightarrow val(o) \in dom(type(K)).$$

$$\begin{tikzcd} \text{Classes} & \text{Types} \\ \mathbf{K} \arrow[r, "type"'] \arrow[d, "inst"'] & \mathbf{T} \arrow[d, "dom"] \\ \mathbf{O} \arrow[r, "val"] & \mathbf{W} \\ \text{Object identities} & \text{Values} \end{tikzcd}$$

Figure 3.2 Formal object model (without inheritance).

The diagram in Figure 3.2 provides a summary of what has been discussed so far: classes have types and are instantiated; types define values whereas objects have values. In order to reflect an assignment of object identities to classes, the diagram in Figure 3.2 needs to commute.

So far we have used class hierarchies to define extensions of classes and to order types. However, the main purpose of a class hierarchy is to provide the basis for inheritance to achieve reusability. The next example serves to illustrate what is missing so far in our formal framework.

Example 3.5

Let us consider the classes `Employee` and `Person`. Every employee is a person. This IsA relationship implies that all properties of a person apply equally to an employee; therefore the definition given below of the respective types seems obvious. Unfortunately, it does not yet have the desired effect, because *type*(`Employee`) is not a subtype of *type*(`Person`).

```
Employee IsA Person

type(Person) = [Name: string,
               Age: integer,
               Domicile: Address
               Fleet: {Vehicle}]

type(Employee) = [Qualifications: {string},
                 Salary: integer,
                 FamilyMembers: {Person}]
```

In the above example the desired intent is to define attributes not only for Person, but also implicitly for Employee, according to the condition for a well-formed subtype ordering. If reusability of definitions is to be permitted, we must differentiate between the type *directly attached to a class* by means of the *type* mapping and the type *implied through the class hierarchy IsA* via inheritance of type information.

Analogous considerations apply to default values. Defaults (cf. Section 1.5.2) are values which are to apply to all objects in a respective class. A default inherited from a certain class applies to an object unless it is overwritten by a more specific value attached either to a subclass of the respective class or directly to the object.

Example 3.6

We want to express that all vehicles, and in particular vehicle #4, are green without having to specify this for every vehicle separately. We define a corresponding default value for class `Vehicle`; this is achieved in a formal manner by allowing us to apply function *val* to classes, for example to class `Vehicle` in this example.

```
type(Vehicle) = [ Model: string,
       Manufacturer: Company,
       Colour: string ]
val(Vehicle) = [ Colour: 'green']
val(#4) = [ Model: 'Golf',
       Manufacturer: #41]
#4 IsA Vehicle
```

Analogous to Example 3.1 the intent here is to ensure that every value defined as a default for class Vehicle is also defined for every individual vehicle, unless a different colour is assigned explicitly to a particular vehicle. Consequently, the vehicle with object identity #4 is green. Again we must differentiate between the value *directly attached to an object* and the value *implied through the class hierarchy IsA* via the inheritance of values.

In order to express inheritance in the terms described above, we need to expand our current formal framework. We can restrict ourselves to inheritance in connection with tuple types, because we need attributes to access type information and class values. The following technique acquires the components to be inherited from the next superclass relative to the class hierarchy for which the component in question is defined. A class K' is called the next class to class K with respect to a certain property; if K IsA K', the property is defined for K', and for every other class K'' for which the property is defined as well, the following holds: if K IsA K'' and K'' IsA K', then $K' = K''$. Note, that because K *IsA* K, for all classes K, every class is its own superclass or subclass.

To simplify matters we shall assume for the time being that there are no inheritance conflicts (Section 2.4.1); we shall later present a property for structure schemata guaranteeing that no conflicts will occur (Definition 3.13).

We expand the mapping *type* introduced earlier to a mapping *Type*, thus taking into account inheritance of attributes:

Definition 3.10

Let SC_{struc} = (**K**, *IsA*, *type*) be a structure schema, $K \in \mathbf{K}$ a class, $A \in \mathbf{A}$ an attribute and $T \in \mathbf{T}$ a type. *Type* : $\mathbf{K} \rightarrow \mathbf{T}$ is a mapping with the following property:

If *type*(K) is a tuple type, then $A : T$ is a component of *Type*(K) whenever there exists a superclass K' of K in which A is defined with type T: that is, $A : T$ is a component of *type*(K') and, in addition, K' is the next superclass to K in which A is defined.

In this expanded framework we have to adapt the property of well-formedness accordingly. A structure schema $SC_{struc} = (\mathbf{K}, IsA, type)$ is now *well formed* in the following way:

$$K\ IsA\ K' \Rightarrow Type(K) \leq Type(K').$$

Let us now consider the inheritance of defaults and to this end define $\mathbf{W}_{tup}$ to be the set of all tuple values in $\mathbf{W}$. We define a function $val_{st} : \mathbf{K} \rightarrow \mathbf{W}_{tup}$ which has the following property:

> If $K \in \mathbf{K}$ and $A : w$ is a component of $val_{st}(K)$, then $Type(K)$ contains a component of A. Further, if this component is $A : T$, then also $w \in dom(T)$.

In contrast to *val*, val_{st} expresses schema information. Accordingly, a structure schema SC_{struc} now has the form $SC_{struc} = (K, IsA, type, val_{st})$. Now we are ready to expand the mapping *val* to a mapping *Val*, such that inheritance of defaults is taken into account:

Definition 3.11

Let $SC_{struc} = (\mathbf{K}, IsA, type, val_{st})$ be a structure schema, $\mathbf{O}$ a set of object identities and $\mathbf{W}$ a set of values. $Val : \mathbf{O} \rightarrow \mathbf{W}$ is a mapping with the following property:

Let $o \in \mathbf{O}$, K the base class of o, $A \in \mathbf{A}$ an attribute name and $w \in dom(A)$. If $Type(K)$ is a tuple type having a component with respect to A, then $A : w$ is a component of $Val(o)$ if either $A : w$ is already a component of $val(o)$ or the following applies:

(i) $val(o)$ does not contain a component of A, and

(ii) there is a superclass K' of K, such that $A : w$ is a component of $val_{st}(K')$, and, in addition, K' is also the next superclass to K having a component of A.

Otherwise we have $Val(o) = val(o)$.

The arguments we have put forward are illustrated in Figure 3.3 and allow us to take into account the inheritance of attributes and defaults:

$$\begin{tikzcd}
\text{Classes} & \text{Types} \\
\mathbf{K} \arrow[r, "Type"] \arrow[d, "Inst"'] & \mathbf{T} \arrow[d, "dom"] \\
\mathbf{O} \arrow[r, "Val"] & \mathbf{W} \\
\text{Object identities} & \text{Values}
\end{tikzcd}$$

Figure 3.3 Formal object model (with inheritance).

Definition 3.12

Let $\mathrm{SC}_{struc} = (\mathbf{K}, IsA, type, val_{st})$ be a structure schema. An *instance* of SC_{struc} has the following form:

$$d(\mathrm{SC}_{struc}) = (inst, val),$$

where *inst* is a base extension and the following holds:

$$o \in inst(K), K \in \mathbf{K} \Rightarrow Val(o) \in dom(Type(K)).$$

One should be aware of how inheritance is realized here: inheritance implied by SC_{struc} and $d(\mathrm{SC}_{struc})$ is made explicit by means of the expansion of *type* and *val* to *Type* and *Val* respectively. Obviously, this is in line with our intention to reuse definitions and values attached to superclasses without having to define them redundantly in subclasses.

As promised earlier we conclude this section by giving a definition for structure schemata without inheritance conflicts.

Definition 3.13

Let $\mathrm{SC}_{struc} = (\mathbf{K}, IsA, type, val_{st})$ be a structure schema. SC_{struc} is called *free from inheritance conflicts* if for all triples of classes $K, K', K'' \in \mathbf{K}$ with K'' IsA K, K'' IsA K' the following applies:

(i) If $type(K)$ and $type(K')$ contain a component of A but $type(K'')$ does not, then there exists a class K''' with $type(K''')$ containing a component A; moreover, K''' IsA K, K''' IsA K' and K'' IsA K'''.

(ii) val_{st} is treated in an analogous way.

Example 3.7

Let us assume that a class K has two superclasses which are not ordered by IsA and let us further assume that for both superclasses there is defined an attribute A, but with different types. Without additional precautions it is not clear which of the two definitions of A is to be valid for K:

$$type(K) = [], K\ IsA\ K', K\ IsA\ K'', type(K') = [A : T'], type(K'') = [A : T'']$$

We can solve this inheritance conflict by introducing an additional class $\underline{K}$ which redefines attribute A with the desired type. This explicit overwriting removes the ambiguities. Note that the new class hierarchy meets the condition of being free of inheritance conflicts:

$\underline{K}\ IsA\ K'$, $\underline{K}\ IsA\ K''$, $K\ IsA\ \underline{K}$,
$type(K) = []$, $type(K') = [A : T']$, $type(K'') = [A : T'']$, $type(\underline{K}) = [A : T]$

3.2 Modelling of behaviour

The behaviour of an object is given by the methods that can be applied to it. Therefore, modelling the behaviour involves, on the one hand, determining which methods are applicable to which objects, and, on the other, defining what effects these methods have. We capture these two aspects in the following way:

- Methods are assigned to classes. The applicability of a method to an object follows from the inheritance rules, analogous to the inheritance of attributes and defaults.
- We attach signatures to methods which define the types of the input parameters and the result. We do not consider how methods are implemented. This procedure is analogous to that used within abstract data types, where the externally visible interfaces are defined by means of signatures separated from the externally invisible implementations of the operations.

Let us assume that **M** is a finite set of *method names*. An expression of the form

$$(M : K \times T_1 \times ... \times T_k \rightarrow \mathrm{T}),$$

where $k \geq 0$, $M \in \mathbf{M}$ is a method, $K \in \mathbf{K}$ is a class, T_i, $T \in \mathbf{T}$ are types, $1 \leq i \leq k$, is called a *signature*, which is associated with class K with respect to method M. Types T_i, $1 \leq i \leq k$, state the expected type of the input arguments and type T states the type of the result. Having clarified signatures we can continue:

Definition 3.14

A *behaviour schema* is an expression

$$\mathrm{SC}_{behav} = (\mathbf{K}, IsA, \mathbf{S}),$$

where **K** is a finite set of classes and **S** a finite set of signatures with the following property:

If $(M : K \times T_1 \times ... \times T_k \rightarrow T)$, $(M : K \times T'_1 \times ... \times T'_l \rightarrow T') \in \mathbf{S}$ where $k, l \geq 0$, then $k = l$, $T = T'$ and $T_i = T'_i$ for $1 \leq i \leq k$.

If T is a set type, then M is said to be *set-valued*, otherwise M is said to be *scalar*.

A behaviour schema which is defined in the above manner allows us to attach signatures of methods with the same names to different classes; however, not more than one signature per method for each class is allowed.

Analogous to well-formed structure schemas we shall now introduce a condition which is known as *contravariance* of the method arguments and *covariance* of the method results. We have encountered this requirement in an informal way already in Section 2.4.3; we shall use it again later to study safety of path expressions.

Definition 3.15

A behaviour schema $\mathbf{SC}_{behav}$ = (**K**, *IsA*, **S**) is said to be *well formed* if the following holds:

$$(M : K \times T_1 \times ... \times T_k \rightarrow T), (M : K' \times T'_1 \times ... \times T'_l \rightarrow T') \in \mathbf{S}, K' \, IsA \, K, k,l \geq 0$$
$$\Rightarrow k = l, T' \leq T, T_j \leq T'_i, 1 \leq i \leq k.$$

Signatures define the allowed method calls. Let $S = (M : K \times T_1 \times ... \times T_k \rightarrow T)$. Method M can be applied to all objects of class K which inherit the signature of M that is attached to K. A call of M is called *allowed* if it is in accordance with S:

$M(o, w_1, ..., w_k)$, respectively $o.M(w_1, ..., w_k)$

where $o \in Inst(K)$ and $w_i \in dom(T_i)$, $1 \leq i \leq k$.

Whenever for some w_i it is true that $w_i \in dom(T'_i)$, $T'_i \leq T_i$, we shall assume that only the T_i part of w_i will be considered for the method call (see Definition 3.5). This actually means that methods may also be called with values which are of a subtype of the respective argument type stated in the signature.

If an object o inherits a signature of a method M which is attached to a class K, then K is called the *receiver class*. The selection of a receiver class is known as *method resolution*. This leads to the following:

Definition 3.16

Let $\mathbf{SC}_{behav}$ = (**K**, *IsA*, **S**) be a behaviour schema. An *instance* to $\mathbf{SC}_{behav}$ is expressed as

$d(\mathbf{SC}_{behav}) = (inst, impl, res)$,

where *inst* is a base extension and *impl* and *res* respectively are mappings of the following form:

(i) *impl* is a mapping which attaches a partial function I (the 'implementation') over the respective value sets to signatures $S \in \mathbf{S}$. If $S = (M : K \times T_1 \times ... \times T_k \rightarrow T)$, then the following holds:

$I : dom(K) \times dom(T_1) \times ... \times dom(T_k) \rightarrow dom(T)$.

(ii) *res* : (**M**,**K**) $\rightarrow$ **S** is a partial mapping which attaches a signature $S \in \mathbf{S}$ of the form $S = (M : K' \times T_1 \times ... \times T_k \rightarrow T)$ to a method call $M(o, w_1, ..., w_k)$. Moreover, let K be the base class of object o, then K *IsA* K' applies. If no such signature exists, then *res* is not defined for the respective method call.

Example 3.8

Let us consider classes `Employee` and `Person`. `Employee` has in part the following structure:

```
type(Employee) = [ . . . ,
       Salary: integer,
       FamilyMembers: {Person}]
```

In order to increase the salary, for example, a method with the following signature could be used:

```
(MoreSalary : Employee × integer → integer).
```

If a member is added to the family of an employee, a method with the signature

```
(Offspring : Employee × Person → {Person})
```

can be used to reflect the new situation.

It should be noted that according to the definition given above, the implementation of a method is expressed by means of a partial function. Proposals as to how to realize such functions in the form of programs will be presented in the second part of this book, when we examine languages used with concrete examples of object-oriented databases.

The selection of a receiver class according to a method call depends either on the base class of the object alone, or, in addition, on the argument types. We can distinguish between the following strategies.

Let us suppose that $M(o, w_1, ..., w_k)$ is a method call and K is the base class of o. The receiver class is that class K' for which either

(1) a signature to M in the form $(M : K' \times T_1 \times ... \times T_k \to T)$ is defined and which is also the next class to K with respect to *IsA* for which this holds. Thus the selection depends only on the base class of the object (*single dispatch*).

or

(2) a signature to M in the form $(M : K' \times T_1 \times ... \times T_k \to T)$ is defined, where $w_i \in dom(T_i)$, $1 \le i \le k$, and which is also the next class to K with respect to *IsA* for which this holds. Thus the selection depends not only on the base class of the object but also on the types of the arguments (*multiple dispatch*).

The first strategy has the advantage that it is easier to implement; unlike the second strategy, however, it may happen that the correct typing of the arguments is not guaranteed with respect to the selected implementation, even if a suitable implementation defined for a superclass exists.

Example 3.9

Let us examine classes `Employee` and `Manager` from Example 2.27. The signatures of the method for Deputy are expressed in the following manner:

```
(Deputy  :  Employee × Employee → boolean)
(Deputy  :  Manager × Manager  → boolean)
```

Let us assume that we want to send a message to a manager concerning `Deputy`. If we apply the first strategy, where the selection of the implementation depends only on the receiver object, then the implementation attached to class `Manager` will be selected. In situations where the second parameter is not a manager we can expect a type error at run time. Using the second strategy, however, would mean that in these situations implementations of the class `Employee` would be selected and a type error would be avoided because every manager is also an employee.

Unless explicitly stated otherwise, we shall assume that the strategy of single selection is applied, which is realized accordingly by means of the mapping *res*.

The following definition summarizes what has been said in this chapter; the reader is also referred to the illustration in Figure 3.4:

Definition 3.17

Let SC_{struc} = (**K**, *IsA*, *type*, val_{st}) be a structure schema and SC_{behav} = (**K**, *IsA*, S) a behaviour schema with instances $d(\mathrm{SC}_{struc})$ = (*inst*, *val*) and $d(\mathrm{SC}_{behav})$ = (*inst*, *impl*, *res*). Then an *object base schema* SC is given by:

SC = (**K**, *IsA*, *type*, val_{st} , S);

and an *object base* by:

d(SC) = (*inst*, *val*, *impl*, *res*).

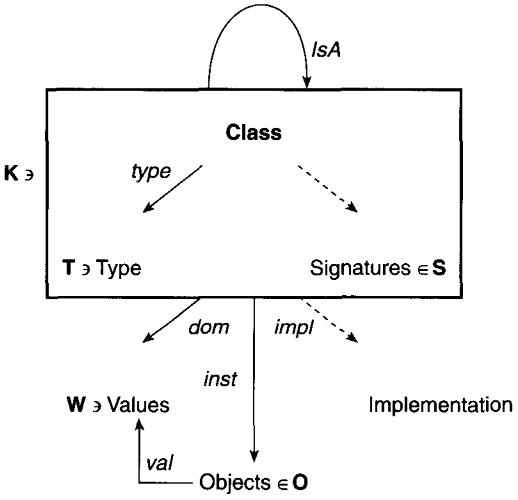


Figure 3.4 Summary of the formal model.

An object base schema **SC** integrates the important aspects we have examined with regard to the structure and behaviour of object-oriented databases: a hierarchy of typed classes with (possibly) attached defaults and a set of method signatures. An object base *d*(**SC**) represents a possible state: that is, it contains a set of objects with values, an abstract implementation for each signature and a strategy to realize inheritance according to the definitions expressed in **SC**. In the next section we shall exemplify the application of these definitions by means of path expressions.

3.3 Formal treatment of path expressions

There are two reasons to look at path expressions in detail. First, path expressions are a distinctive feature of object-oriented languages. Despite their elegance they require a nontrivial formal treatment; the latter is mainly due to the complex data structures, particularly sets, which do not exist in relational languages. Second, analysing path expressions allows us to put our formal framework into practice and thus to demonstrate its relevance. We restrict ourselves to the basic structure of path expressions, first dealing with expressions consisting of scalar attributes and methods and then looking at the more general case, namely set-valued attributes and methods.

3.3.1 Scalar path expressions

Let **O** be a set of object identities, **V** a set of variables, **W** a set of values, **T** a set of types and **A**, **K**, **M** pairwise disjoint sets of names for attributes, classes and methods respectively. Let us further refer to an object base schema **SC** = (**K**, *IsA*, *type*, val_{st}, S) and an associated object base *d*(**SC**) = (*inst*, *val*, *impl*, *res*). We may then inductively define scalar path expressions as follows:

Definition 3.18

(P1) Each object identity o and each variable v is a *scalar* path expression.

(P2) If $p_0, p_1, ..., p_k$ are scalar path expressions and M is a scalar attribute, $k = 0$, or a scalar method, $k \geq 0$, then

$$p_0.M(p_1..., p_k)$$

is also a *scalar* path expression, where $p_1, ..., p_k$ are the arguments. In the case $k = 0$ we write instead of $p_0.M()$ the shorter $p_0.M$.

Let **P** be the set of all (for the time being, scalar) path expressions.

It should be noted that in a path expression $p_0.M(p_1, ..., p_k)$ the expression p_0 supplies the object identity to which the method M is applied; and, if $k \geq 1$ applies, the expressions $p_1, ..., p_k$ supply the required parameter values.

Example 3.10

Examples of scalar path expressions are:

```
#41
employee
employee.Domicile
automobile.Drive.Engine.HP
employee.Deputy(company.President)
```

Here, `employee`, `automobile` and `company` are variables.

Methods are characterized by signatures which define the required types of the parameter values as well as the result type. It is necessary to ensure that methods are called only for objects for which there exists a signature. For reasons of static type checking (cf. Section 2.4.3), each path expression needs to be assigned a type in such a way that the value defined by the path expression is exactly of this type or at least a subtype thereof.

For this purpose we need to clarify what we understand by the type *Type* of a path expression. At first glance there do not seem to be any difficulties in determining the type of a path expression p with the general form $p_0.M(p_1, ..., p_k)$, as only the signature for M needs to be consulted in order to select the result type of this signature as the type for p. However, the question arises of which signature to select if there are several signatures whose result types may even be incomparable with respect to the subtype order $\leq$. In order to make the right selection in accordance with inheritance, we obviously need to know the type of the path expression p_0. Therefore, we shall follow an inductive approach to determine the type of a path expression, thereby applying a simple form of *type inference*.

First, we extend the domain of the already introduced mapping *Type*, that is, the set of class names **K**, by the set of path expressions **P** and thus also by the set of object identities **O** and the set of variables **V**. We can assign a type to a path expression p, provided that its structure conforms to certain criteria of well-formedness. Unless these criteria are fulfilled, the type of p is not defined.

Before looking at the general case, we shall deal with some special cases of path expressions: that is, path expressions which are given by an object identity or a variable. First, the *type* of an object identity o is its base class: that is, $Type(o) = K$, provided that $o \in inst(K)$. With respect to variables we assume that the type of each variable is given in advance. For example, if a path expression is used within an SQL expression, the type could have been defined within the from-clause. In particular, for the following we always can assume $Type(v) \in \mathbf{K}$.

Definition 3.19

Let $p = p_0.M(p_1, ..., p_k)$ be a path expression in **P** and let $Type(p_0) \in \mathbf{K}$. Then the type of p is defined as follows:

(T1) Let M be an attribute, that is, $k = 0$. If there exists a component of the form $M : T$ of $Type(Type(p_0))$, then $Type(p) = T$. Otherwise, $Type(p)$ is not defined.

(T2) Let M be a method. If there exists a superclass K of $Type(p_0)$ such that K is the next superclass with a signature S of the form $(M : K \times T_1 \times \ldots \times T_k \rightarrow T)$, then $Type(p) = T$, provided that $Type(p_i) \leq T_i$, $1 \leq i \leq k$ applies. Otherwise $Type(p)$ is not defined.

Example 3.11

Here we show the types of the path expressions presented in our previous example:

```
Type(#41) = Company
Type(employee) = Employee
Type(employee.Domicile) = Address
Type(automobile.Drive.Engine.HP) = integer
Type(employee.Deputy(company.President)) = boolean
```

We assume that the types of the variables `employee` and `automobile` are the classes `Employee` and `Automobile`, respectively. For the last expression we further assume the signature:

```
(Deputy : Employee × Employee → boolean)
```

The following is an example of a path expression for which there is no type defined:

```
vehicle.Manufacturer.HP
```

This is because attribute `HP` is not defined in the class referenced by `Manufacturer`. A further example of this kind is:

```
company.Headquarters.Place.Address
```

A path expression is called *typed*, if a type is defined for it. A typed path expression has a structure which is *well formed* in the following manner:

Let $p = p_0.M(p_1, \ldots, p_k)$ be a typed path expression.

- The type of p_0, that is, $Type(p_0)$, is a class; therefore a path expression p at least represents a possible successive application of attributes or methods.
- The attribute M or a signature to the method M is indeed defined for $Type(p_0)$.
- Provided that M is a method, then, with respect to the selected signature, the arguments $p_1, \ldots, p_k$ are of the type defined therein or of a subtype thereof.

Having clarified the syntax and the type of scalar path expressions we shall now turn our attention to their semantics; that is, we shall be able to indicate what value is defined by a path expression, respectively when it is not defined. Subsequently, we shall demonstrate that the type of this value is the type of the path expression or a subtype thereof.

We shall restrict ourselves to expressions without variables and thus require that variables must be bound according to their type before the path expression is evaluated. When evaluating an SQL expression, these bindings may, for instance, be supplied by the from-clause. Let $\mathbf{P}' \subseteq \mathbf{P}$ be the set of variable-free, typed path expressions and let $sem : \mathbf{P}' \rightarrow \mathbf{W}$ be a partial mapping. Since methods are partial functions, *sem* cannot be defined for all vectors of arguments of method calls. We shall define the mapping *sem* inductively and in this way obtain an obvious strategy for the evaluation of path expressions. The semantics *sem* of a path expression p is defined as follows:

Definition 3.20

Let p be an object identity; then $sem(p) = p$. Let $p = p_0.M(p_1, ..., p_k)$ be a scalar, variable-free, typed path expression and assume that $sem(p_i)$ is defined, $0 \leq i \leq k$.

(S1) Let M be an attribute and therefore $k = 0$. Because p is typed, $Val(p_0)$ contains a component of the form $M : w$. Then

$$sem(p) = w$$

(S2) Let M be a method. Since p is typed, there is an implementation of M in the form $I = impl(res(M,Type(p_0)))$. If $I(sem(p_0), sem(p_1), ..., sem(p_k))$ is defined, then

$$sem(p) = I(sem(p_0), sem(p_1), ..., sem(p_k))$$

Otherwise, $sem(p)$ is not defined.

Example 3.12

Based on the class extensions shown in Figure 1.9 the following values are obtained:

```
sem(#41) = #41
sem(#68.Domicile) = #82
sem(#10.Drive.Engine.HP) = 80
```

Let us now consider a variable-free, typed, scalar path expression p; let T be the type of p. Having defined what we mean by the type of a path expression and its semantics, the interesting question then is whether we can guarantee, for every object base, that the value defined by the semantics is indeed of type T, respectively, of a subtype of T. We shall demonstrate that the answer is positive; in other words, we are going to prove the safety of path expressions. The crucial point here is that the selection of an attribute or a signature of a method to be applied to an object takes place through the inheritance mechanism, which itself depends on the base class of the object in question. The actual selection is, therefore, not visible from the syntactic structure of a path expression, but depends on the concrete objects being referenced during the evaluation of the path expression.

To demonstrate the problem of safety in more detail let us consider a path expression of the form $p = p_0.M_1.M_2$, where M_1 and M_2 are assumed to be methods. The type of p is given by the result type of method M_2 as stated in the signature of M_2 under consideration; this signature itself depends on the result type of the signature considered with respect to M_1. Let us assume that T_1 is the class which appears to be that type. If a subclass T'_1 of T_1 exists for which there is defined a different signature to M_2, then the implementation of M_2 which is actually selected for evaluation of p may differ from the one which has been assumed for determining the type of p. Indeed, the types of the value defined by a path expression and of the path expression itself may differ! Well-formed structure and behaviour schemata, however, guarantee that these two types are ordered by $\leq$, that is, $T'_1 \leq T_1$ applies, as is in fact desired. Let us look at this in more detail.

Proposition 3.1

Let $p = p_0.M(p_1, ..., p_k)$ be a variable-free, scalar, typed path expression, where $Type(p) = T$. Assume $sem(p)$ is defined. Then $sem(p) \in dom(T^*)$, where $T^* \leq T$.

The proof of the proposition is a simple induction on the number of occurrences of attributes and methods appearing in a path expression p. For brevity we restrict ourselves to methods; attributes can be treated similarly to methods without arguments.

Assume first that in p there is only one occurrence of a method: that is, all p_i, $0 \leq i \leq k$ are object identities. Then, because the type of p depends on the base class of p_0, the signature of method M which is selected during evaluation of p and the signature being considered for the definition of *Type* with respect to p coincide. Therefore, we have $sem(p) \in dom(T)$.

Assume now that the proposition holds for a number of method occurrences $m' \geq 1$. We have to show that the proposition still holds if $m = m' + 1$. Let p be a path expression with m occurrences of methods and let $Type(p) = T$. We can construct p out of some p' for which the number of occurrences is m'. Let $p' = p'_0.M'(..., p'_i, ...)$ and $Type(p') = T'$; there are two cases:

- $p = p'_0.M'(..., p'_i, ...).M(p_1, ..., p_k)$. Because of our assumptions we have $sem(p') = o'$, where $Type(o') \leq Type(p')$. According to method resolution and the contravariance and covariance of methods we can conclude $sem(p)$ is defined and moreover $sem(p) \in dom(T^*)$, where $T^* \leq T$, as desired.
- $p = p'_0.M'(..., p'_i.M(p_1, ..., p_k) ...)$. Because of our assumptions we have $sem(p'_i) = o'_i$, where $\text{Type}(o'_i) \leq Type(p'_i)$. According to method resolution and the contravariance and covariance of methods we can conclude that $sem(p'_i.M(p_1, ..., p_k))$ is defined and moreover that $sem(p'_i.M(p_1, ..., p_k)) \in$

$dom(T_i)$, where $T_i \leq T_i'$ and T'_i is the argument type of the corresponding signature with respect to M'. Therefore $sem(p) \in dom(T^*)$, where $T^* \leq T$, as desired.

3.3.2 Set-valued path expressions

We now want to include set-valued path expressions in our discussion.

Definition 3.21

Let $p = p_0.M(p_1, ..., p_k)$, $k \geq 0$. p is a *set-valued* path expression, provided one of the two following conditions applies:

(P3) p_0 is a path expression and M a set-valued attribute or a set-valued method.

(P4) p_0 is a set-valued path expression and M an attribute or a method.

If $k = 0$, that is, no arguments are given, we can write simply $p_0.M$. Now **P** is the set of all (scalar and set-valued) path expressions.

Example 3.13

The following examples illustrate various possibilities of forming set-valued path expressions:

```
person.Fleet
person.Fleet.Manufacturer
person.Fleet.Manufacturer.Subsidiaries
person.Fleet.Manufacturer.Subsidiaries.Manager
person.Fleet.Manufacturer.Subsidiaries.Manager.Employees
```

The type of a set-valued path expression $p = p_0.M(p_1, ..., p_k)$ can be determined analogously to scalar path expressions. It should be taken into account, however, that the type of an attribute or the result type of a method may be a set. If $p = p_0.M(p_1, ..., p_k)$ is a path expression, it is not sufficient to require only $Type(p_0) \in \mathbf{K}$ (cf. Definition 3.19); in a set-valued path expression either $Type(p_0) \in \mathbf{K}$ or $Type(p_0) = \{K\}$, $K \in \mathbf{K}$ must apply. In the latter case the attribute or the method M has to be applied to the individual elements of the set defined by the path expression (which corresponds to the effect of a flatten operator). To give the semantics of set-valued path expressions, in an analogous way to scalar path expressions, we shall consider variable-free expressions only.

Definition 3.22

Let $p = p_0.M(p_1, ..., p_k)$ be a typed, set-valued, variable-free path expression and let us further assume that $sem(p_i)$ is defined, $0 \leq i \leq k$.

(S3) Let M be an attribute and therefore $k = 0$.

- Let p_0 be scalar and M set-valued:
  $W' = \{w \mid p' = sem(p_0), Val(p') \text{ contains a component } M : W, w \in W\}$
- Let p_0 be set-valued and M scalar:
  $W' = \{w \mid p' \in sem(p_0), Val(p') \text{ contains a component } M : w\}$
- Let p_0 be set-valued and M set-valued:
  $W' = \{w \mid p' \in sem(p_0), Val(p') \text{ contains a component } M : W, w \in W\}$

Define $sem(p) = W'$.

(S4) Let M be a method and assume $sem(p_i)$ is defined, $0 \leq i \leq k$. Let $p'_i = sem(p_i)$, if p_i is scalar, respectively $p'_i \in sem(p_i)$, provided p_i is set-valued, $0 \leq i \leq k$. Let further $I = impl(res(M, Type(p'_0)))$. Let at least one of p_0 and M be set-valued; then we have

$$W' = \{w \mid w \in I\,(p'_0, p'_1, \ldots, p'_k)\}$$

If M is scalar and $W' = \emptyset$ holds, then $sem(p)$ is not defined. Otherwise define $sem(p) = W'$.

Example 3.14

Let us look at the path expressions in our previous example, assuming a situation as shown in Figure 1.9.

```
sem(#60.Fleet) = { #10, #11 }
sem(#60.Fleet.Manufacturer) = { #41, #42 }
sem(#60.Fleet.Manufacturer.Subsidiaries) = { #50, #51, #52 }
sem(#60.Fleet.Manufacturer.Subsidiaries.Manager)
= { #60, #65, #66 }
sem(#60.Fleet.Manufacturer.Subsidiaries.Manager.Employees)
= { #65, #66, #68,... }
```

Types of set-valued path expressions and their semantics are related in an analogous way to scalar path expressions (see Proposition 3.1). However, if the last method to be applied in a set-valued path expression is scalar, we must take into account the fact that we obtain a result in the form of a set. This does not necessarily have to be considered a type violation. In accordance with the approach used for relational systems we can interpret a set of this kind as a set consisting of possible answers. The following two path expressions may further illustrate this situation, referring to Figure 1.9:

`#60.Fleet.Manufacturer` yields the answers: #41, #42,
`#60.Fleet.Manufacturer.Subsidiaries.Manager` yields the answers: #60, #65, #66.

3.4 Bibliographical notes

The formal framework for an object-oriented data model which we have introduced in this chapter is based on Kanellakis et al. (1992); the reader is also referred to Abiteboul et al. (1995). New findings regarding deep equality have been published by Abiteboul and Van den Bussche (1995). A different, logic-oriented approach to formalizing object-oriented databases has been studied by Kifer et al. (1995) and will be considered in Chapter 7 of this book. Finally, the formalization of path expressions is based on Kifer et al. (1992) and Frohn et al. (1994).

Part II

Languages

4 Case studies

In this chapter we substantiate our observations on object-oriented database systems by describing four commercially available systems:

- Illustra from Informix Software, Inc.,
- O_2 from O_2 Technology,
- GemStone from GemStone Systems, Inc., and
- ObjectStore from Object Design, Inc.

Following the topic of this book, we confine ourselves to the underlying models and languages. We do not claim that our selection of systems is complete, but they show the multitude of possibilities from which you can choose when taking design decisions for object-oriented database systems:

- Illustra is a development of Postgres, which itself has its roots in Ingres, one of the first relational database systems. Following this tradition, Illustra expands relational concepts by the concepts characteristic of object orientation.
- O_2 is a database system in the classical sense, because it possesses independent languages for the definition and manipulation of databases. The language concepts of relational databases are mapped onto a specific object model and expanded accordingly.
- GemStone extends the object-oriented programming language Smalltalk and aims to make a database system out of Smalltalk.
- Similarly, ObjectStore extends the object-oriented programming language C++ by adding persistence.

Hence, while the first two systems can be considered to be rooted in database systems, the other two can be seen as originating from programming languages. The selection made here therefore reflects at least the two different directions of development that an object-oriented database system can take:

- a database system equipped with an object model and an appropriate language, or
- a high-level (and object-oriented) programming language extended with database functionality.

Within the database approach we can distinguish a more recent branch called *object-relational* systems, which still emphasize the relational view while extending it carefully with object-oriented concepts. We present one example from each tradition in more detail: Illustra for an object-relational system, O_2 for a system with an emphasis on making a database system object oriented, and GemStone for a system starting from an object-oriented programming language. The design decisions of these systems with respect to the architecture and the persistence model become clearer when we discuss the alternative approaches to these two aspects and compare them.

4.1 Architecture and persistence model of object-oriented databases

Object-oriented databases try to cover the same functionality as relational databases. As a consequence, the properties characteristic of object-oriented languages and those characteristic of databases must be integrated. Different approaches are used for the architecture and the persistence model in order to achieve this integration.

These differences also reflect the starting point of the development towards an object-oriented database.

4.1.1 Architecture

Today's database systems are typically based on a *client–server architecture*: a software architecture to realize functionally distributed systems in which the components are realized as independent processes which communicate with each other in a predefined form (via *request–reply pairs*).This fundamental principle is illustrated in Figure 4.1: one process, the *client*, sends a *request* to another process, the *server*, which is known to be able to process the request. The server reacts to the request in the desired way and then sends a *reply* back to the client.

The distribution of tasks between client and server in relational database systems typically is different from the distribution of tasks in object-oriented database systems. In relational database systems client and server communicate by the client's sending an SQL expression to the server; the server then returns the result of the expression in the form of a set of tuples (Figure 4.2). Therefore, the main part of the work in replying to a request is done by the server (which is also referred to as a *query server* or an *SQL server*); the client is responsible for the management of *cursors*, if it cannot process the set of answers as a whole, in order to be able to view the individual tuples in the desired sequence.

The reasons for this distribution of tasks are obvious: a database system aims to store data as free of redundancy as possible under central control in order to ensure integrity. The distribution of data to clients would cause a communication effort leading to reduced overall efficiency. A further argument in favour of the server's processing an SQL expression is the large effort necessary to achieve efficiency. The query optimization component is one of the most complicated parts of a database system. A client can therefore process a request efficiently only if it takes on a large part of the database functionality; normally a client, especially if it is PC-based, cannot do that.

To respond to the specific application demands made on object-oriented databases, they offer alternatives to the query-server-based architecture in which either individual pages or entire objects are selected as a transfer unit for communication; the resulting architecture is called a *page server* or an *object server* respectively (see Figures 4.3 and 4.4).

Object-oriented databases must offer database support for application areas in which relational databases have proved to be insufficient. These application areas (see Section 1.1) are characterized by complex objects among other things. Such an

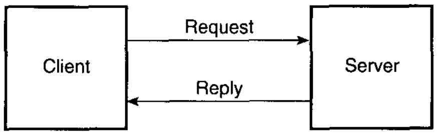


Figure 4.1 Client–server concept.

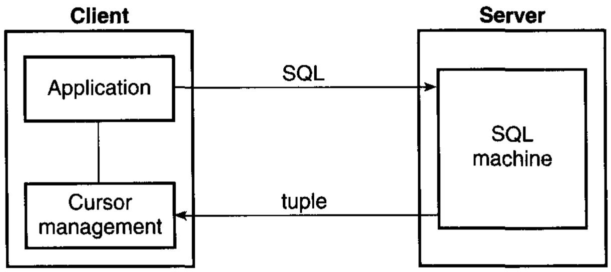


Figure 4.2 SQL-server architecture of a relational system.

object structure, in which relationships between objects are expressed by direct references, suggests so-called *pointer chasing*, for example by means of path expressions, where a client, depending on the processing stage, requests from the server the objects it needs next. In this approach it is not necessary to process joins, because the relationships between the objects are materialized in the form of direct references. Using this approach, the sending of an SQL expression to the server is unnecessary and thus object-oriented database systems can indeed be considerably more efficient than relational databases in these application areas.

If traditional database applications are also to be efficiently supported, navigational and associative access, for instance expressed in SQL, must be supported equally well. Depending on where the emphasis is placed in a concrete object-oriented database system, the architecture will be characterized either by a query server or by a page or object server; hybrid architectures comprising both a query server and a page/object server are also possible.

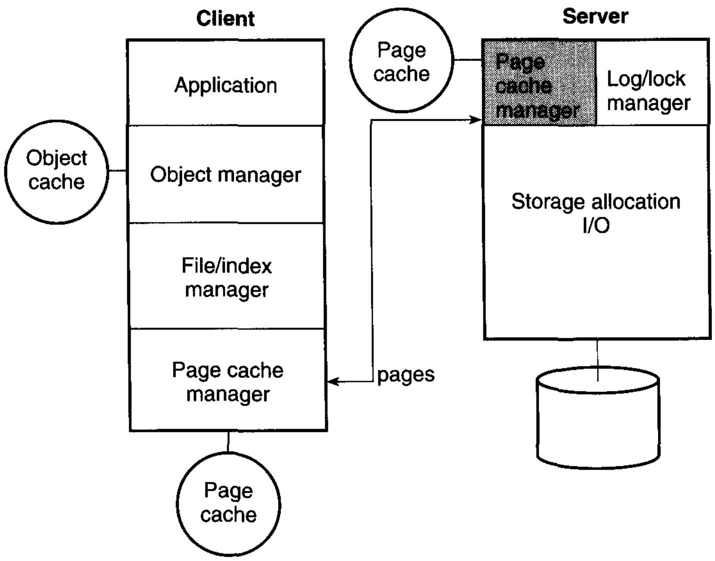


Figure 4.3 Page-server architecture.

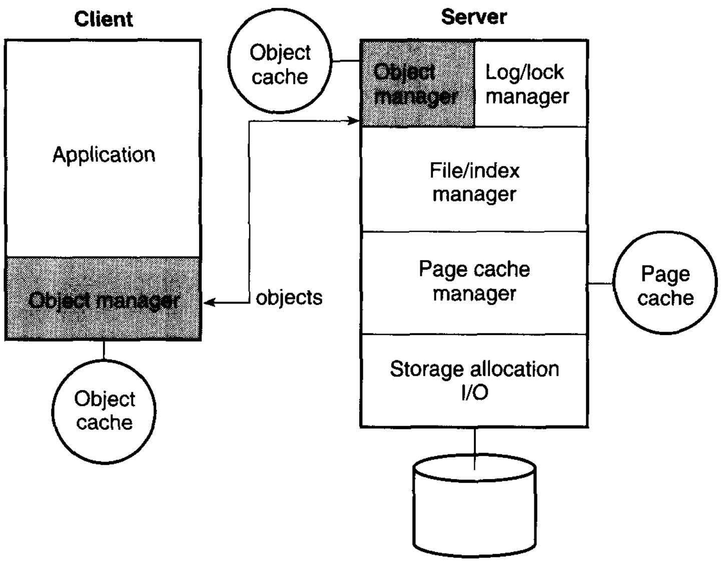


Figure 4.4 Object-server architecture.

The choice of server architecture has implications for the way methods are executed. Objects encapsulate structure and behaviour; within a query- or object-server architecture methods can be executed either at the server or at the server and the clients, while within a page-server architecture they can usually only run at the client. Executing them at the server has the advantage that large data transfers to the clients can possibly be avoided, and that business rules or integrity constraints can be controlled centrally. In principle, applications should not know where methods are stored or where they are executed. If the methods are part of the programming environment of the clients, this might have efficiency advantages, because no execution system need exist at the server. On the other hand, object definitions are then in part distributed over clients and servers, which may cause difficulties in guaranteeing consistency.

4.1.2 Persistence model

Languages of object-oriented databases try to avoid the *impedance mismatch* which can be observed when calling relational query expressions from host languages such as embedded SQL. We have so far studied only the aspect of how object-oriented databases attempt to remedy this situation via application-dependent data types. A seamless integration of programming languages and databases must, however, also find an answer to the question of how transient and persistent objects are to be treated. Ideally, persistence should be characterized as follows:

- It should be *orthogonal* to the type concept, so that in principle an object of an arbitrary type can be persistent.

- *It should be transparent.* The programmer should see no difference in processing with respect to persistent and transient objects.
- It should be *independent* of the storage medium so that no explicit read or write operations are necessary.

The *persistence model* of a language can therefore be regarded as the way in which these aspects are realized in the language.

The most obvious approach to defining persistent objects is to use specific *persistent classes*. This approach corresponds to the traditional approach of a database schema. Generally, this has the consequence that the programmer must move persistent objects explicitly into transient areas and vice versa. A second option is to *instantiate* objects *persistently* without the objects' having to belong to specific persistent classes. Problems may arise when persistent objects possess references to transient objects, thus putting referential integrity in potential danger. One way of tackling these problems is to maintain so-called *inverse* relationships, which facilitate the test for referential integrity if necessary.

A third option is based on the *reachability* of persistent objects. This means that an object is automatically persistent as soon as it is referenced from some persistent object. Similar techniques to those known from network systems are used here: in the simplest case, an object becomes persistent when it is given (in addition to its identity) a user-defined name; named objects are then persistent in their entirety. Moreover, navigation can now start at a named object, called an *entry point*, from which additional (persistent) objects can be reached via references if required.

4.2 Illustra

Of the systems considered in this text, Illustra is the one closest to a traditional database system. The presentation of Illustra is made more difficult by the frequent changes of name of its manufacturer – from Miro to Montage and, most recently, from Illustra to Informix. The 'constant' mother of these products is Postgres; the philosophy of the query language of Postgres is, however, still based on Quel, while Illustra follows the standardization efforts for SQL3 (see Chapter 5).

Illustra essentially combines a relational view of data with central features of object orientation. This motivates the term 'object-relational DBMS' (or ORDBMS for short), which has only recently emerged. Illustra supports the four main features which can be attributed to object-relational database systems:

- support for base type extensions in an SQL context,
- support for complex objects in an SQL context,
- support for inheritance in an SQL environment, and
- support for a production rule system.

In greater detail, the system provides for complex data and object-oriented concepts in an SQL setting through unique record identifiers, user-defined types and corresponding operators as well as functions and access methods, complex objects, inheritance of data and functions, polymorphism, overloading, dynamic extensibility, ad hoc queries and active rules to ensure data integrity. The object model of Illustra extends SQL2 by user-defined types, hierarchies of tables, multiple inheritance, object identity and overloading of functions; it has been ensured that these extensions will conform to the expected SQL3 standard. In this section, we survey the architecture of the system, briefly discuss its object model, and give examples of queries.

4.2.1 Architecture

The architecture of Illustra is shown in Figure 4.5. It is a client–server architecture where the server has the function of a query server. On the user's side there is an SQL parser, expected to be compliant with SQL3. The query system is supported by a rule system for integrity purposes. Queries written in SQL are subject to optimization before they are processed. A function manager takes care of user-defined functions attached to user-defined types. Finally, access methods to physical data structures are provided, and a storage manager handles the database.

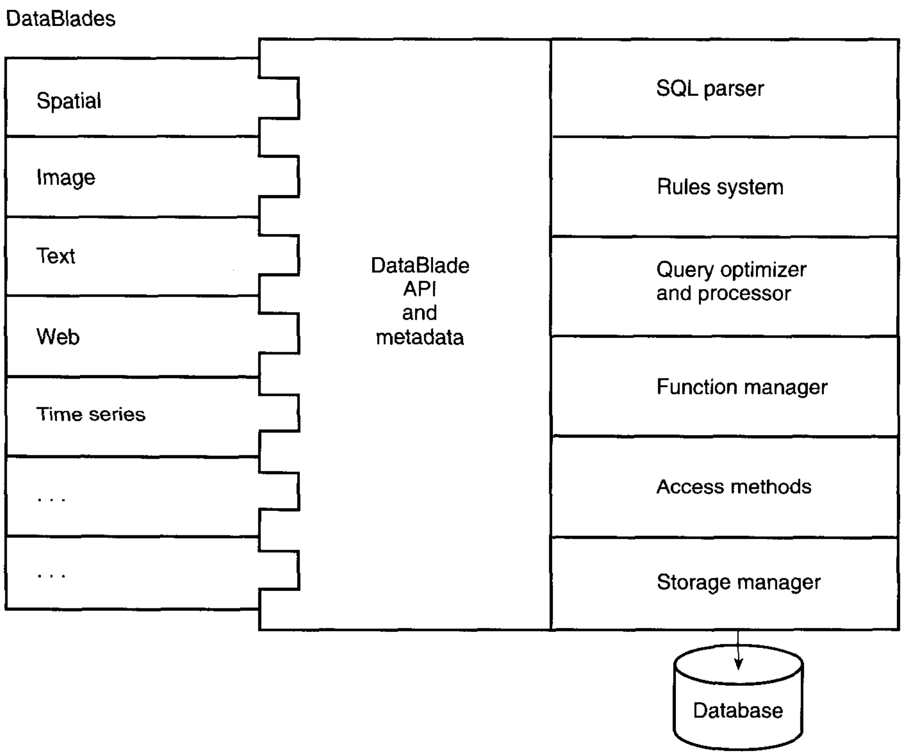


Figure 4.5 Illustra architecture.

All these components interact with the data dictionary, holding metadata, and reside on the server, while user interface and application code run on clients. User-defined functions can be specified to run on either a client or the server; if a function is run on a client, it is handled by a component called the Dispatcher. According to the query-server architecture model, clients communicate with the server by sending SQL expressions. The processing of these expressions may result in calling more user-defined methods, which are themselves part of the database, from the server. This means that Illustra can store 'complete' objects in such a manner that state and behaviour are part of the database.

4.2.2 Type and table declarations

In this subsection we look at type and table declarations in the SQL-based language of Illustra, show how inheritance is realized, and give examples of how to use functions bound to types.

Example 4.1

Referring back to Figure 1.8, the following are valid declarations:

```
create type Address_t (Street varchar(40),
                       Location varchar(30));

create type Auto_t (Model varchar(20),
                    Manufacturer varchar(25),
                    Colour char(5));

create type Person_t (Name varchar(30),
                      Age int,
                      Domicile Address_t,
                      Fleet setof(Auto_t));

create table Person of type Person_t;
```

The above definitions first create three types describing the relevant features of an address, an automobile, and a person, respectively. Notice that type Person_t makes use of the other two types. Finally, a table named Person is created whose tuples are of type Person_t. The base data types used in these definitions are the standard ones from SQL; however, as can be seen in type Person_t, the set constructor (`setof`) can also be used.

Another constructor, not shown in the declarations of Example 4.1, is the *reference* type constructor, which essentially creates a pointer to an element of another type.

Example 4.2

If attribute `Fleet` of type `Person_t` was declared as

```
Fleet setof(ref(Auto_t))
```

this would mean that, in a relation of that type, attribute `Fleet` has as its value a set of pointers to values of type `Auto_t`.

Example 4.1 can be continued to illustrate inheritance.

Example 4.3

The following type declarations introduce several subtypes of type `Person_t`:

```
create type Employee_t (Qualifications setof(varchar(20)),
                        Salary int,
                        FamilyMembers setof(ref(Person_t)))

 under Person_t;

create type Student_t (Major varchar(20)),
                       GPA float)

 under Person_t;

create type StudentEmployee_t (Percent float)
 under Employee_t, Student_t;
```

As can be expected, each subtype inherits all attributes from every supertype. Ambiguities due to multiple inheritance are avoided since the system will disallow a type declaration inheriting incompatible attributes from distinct supertypes.

The subtypes constructed in Example 4.3 cannot store data, so corresponding tables are needed as well. To this end, declarations like the ones shown next are needed.

Example 4.4

The following table definitions introduce the relevant data structures for storing tuples over the types introduced earlier:

```
create table Employee of type Employee_t under Person;

create table Student of type Student_t under Person;

create table StudentEmployee of type StudentEmployee_t
 under Employee, Student;
```

4.2.3 Function inheritance

Inheritance applies not only to attributes, but also to functions in Illustra. In order to illustrate this feature, consider a function determining whether an employee makes more money than, say, employee Susan:

```
create function BetterSal (Employee_t)
returns Boolean as
return $1.Salary > (select Salary from Employee where Name = 'Susan');
```

Given the type and table hierarchies defined above, this function, although defined for type Employee_t, would be inherited by type StudentEmployee_t and is hence applicable to instances of both Employee_t and StudentEmployee_t. The following query therefore determines the names of all employees *or* student employees who make more money than Susan:

```
select e.Name
from Employee e
where BetterSal(e);
```

If the query is to be evaluated on employees *only* (that is, excluding student workers), the appropriate formulation would be

```
select e.Name
from only (Employee) e
where BetterSal(e);
```

Function names may be overloaded. For example, the following declaration introduces a second function named BetterSal, now applicable to instances of type StudentEmployee_t only:

```
create function BetterSal (StudentEmployee_t)
returns Boolean as
return $1.Salary >
  (select Salary from StudentEmployee where Name = 'Peter');
```

This function returns 'true' for student employees making more money than Peter. Since student employees are specific employees, the first query shown above would now apply one version of BetterSal to employees, the other to student employees. In other words, multiple function definitions would be used in the evaluation of the same query, which is a typical example of *polymorphism.*

In the presence of multiple inheritance, function definitions could be inherited from multiple supertypes. For example, if BetterSal was defined above for type Student_t instead of type StudentEmployee_t, it would not be applicable to instances of the latter type, and the system would issue a run-time error.

4.2.4 Rules

Another interesting feature of Illustra is its rule system, which can be used to protect the integrity of a database. The general form of a rule is

on event **do** action

with the meaning that whenever a specified event occurs, the corresponding action is taken. Both events and actions can here be updates or queries, so that there are four basic forms of rules in Illustra. Examples of each form are given next.

Update-update rule:

The following rule watches for an update to Peter's salary and propagates it to Susan:

```
create rule UU as
on update to Employee.Salary
   where current.Name = 'Peter'
do update Employee
   set Salary = new.Salary where Name = 'Susan';
```

Query-update rule:

The following rule watches for a retrieval of Peter's salary; if that happens an insertion is made into an audit table indicating who posed the query, what salary was shown, and at what time the query occurred:

```
create rule QU as
on select to Employee.Salary
   where current.Name = 'Peter'
do insert into Audit values
   (user, current.Salary, current_timestamp);
```

Update-query rule:

In the following example, the user is notified by way of a specific query called an *alerter* that some update has occurred:

```
create alerter PeterSal (mechanism = 'callback');

create rule UQ as
on update to Employee.Salary
   where current.Name = 'Peter'
do alert PeterSal;
```

The second statement defines a rule that watches for an update to Peter's salary and notifies alerter PeterSal if such an update occurs. The alerter, created in the first statement, indicates that the Illustra server will signal to the client from which the update was received that this update has occurred. Other forms of notifications are also allowed.

Query-query rule:

The first rule shown above essentially made sure that Peter and Susan will have identical salaries from the next update to Peter's salary onwards. The same effect could be achieved by the following rule:

```
create rule QQ as
on select to Employee.Salary
   where current.Name = 'Susan'
do instead
   select Salary from Employee where Name = 'Peter'
```

This rule will modify any query asking for Susan's salary in such a way that Peter's salary is shown instead; thus, the two salaries will always look identical.

4.2.5 DataBlades

A novel part of Illustra is the so-called *DataBlades*, which have the task of simplifying the development of object-oriented applications, and which essentially extend the set of data types the system can understand and process. A DataBlade is a user-defined type, or a collection of such types, together with the functions applicable to it and specific customized access techniques. Therefore, a DataBlade is more than an application-specific class, because it additionally provides optimized (internal) access techniques. As indicated in Figure 4.5, the first available DataBlades support text, time, spatial data, image processing, and World Wide Web applications (that is, WWW-based access to an Illustra database). The Spatial DataBlade, for example, provides R-trees as a specialized access method, whereas the Text DataBlade contains a specific access method for full-text search. Actually, there are distinct spatial blades for two-dimensional as well as for three-dimensional data. A time series is a set of data organized by time of occurrence, for example data representing the movement of a share or a bond price. The Web blade provides capabilities to create and maintain Web pages for the Internet. From a database perspective, Web pages are complex objects that can be stored, manipulated, or generated from other data. Due to the recent explosive growth of the Internet, the automatic creation and management of Web pages is likely to become a large area of database application.

Clearly, these DataBlades are only a starting point, as users are encouraged to create their own specialized DataBlades in order to facilitate an object-oriented approach in standardized and efficient software development. The basic DataBlade supplied with the system is the *Foundation* blade comprising more than 40 traditional data types (including integer, character, etc.) and over 200 functions on these types.

4.3 O_2

The O_2 system has been marketed by O_2 Technology since 1991. O_2 is rooted in the tradition of database languages. Thus, O_2 is familiar with the notion of a database schema in which the classes and their hierarchy are defined. Furthermore, structured values can be directly applied without the need for nested object structures. O_2 also comprises an object-oriented version of SQL. O_2 Technology is involved in the standardization bodies of ODMG (see Chapter 5).

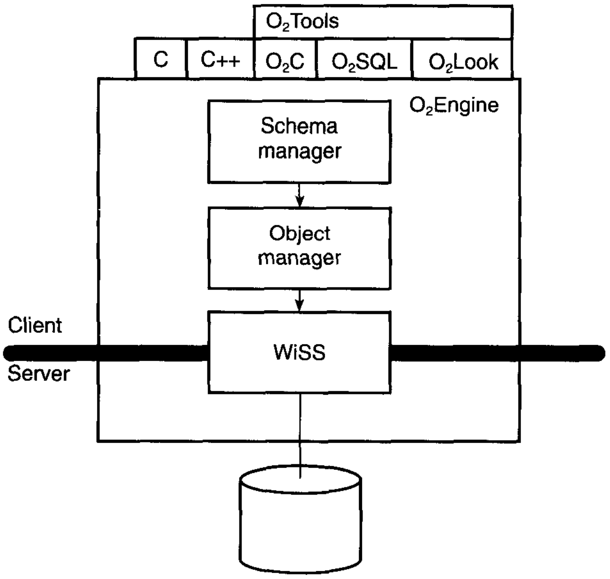


Figure 4.6 O_2 architecture.

O_2 has a client–server architecture which contains the following components (see Figure 4.6):

- the core, called O_2Engine, a language-independent object base system which supports the object model of O_2 and its schema administration;
- O_2Look, a graphic tool for the creation, presentation and editing of database objects;
- O_2SQL, an SQL-like query language, which we shall discuss in the following sections;
- O_2Tools, a programming environment with graphic browser, editors, debugger and schema documentation;
- interfaces for C and C++;
- O_2C, a 4GL programming language which represents a superset of C.

The O_2Engine consists of three levels. The Schema Manager forms the top level, which is responsible for the creation, use, modification and deletion of classes, methods and global names; furthermore, its tasks include the realization of inheritance and checking the consistency of the schema. The Object Manager forms the middle level and organizes the manipulation of objects, including the sending of messages (method calls). Moreover, it is responsible for internal aspects like persistence, index structures and clustering. The lowest level is made up of a development of the Wisconsin Storage System (WiSS), which here assumes the role of an intelligent disk server.

The client–server architecture of the O_2Engine is a page server; only the lowest level of the O_2Engine is on the server. This separation is possible because modern workstations have sufficient processing capacity to take over a considerable part of the functionality of the system. According to this separation the server does not recognize objects. This means that no queries or methods can be carried out directly at the server. In a previous version O_2 was based on an object-server architecture; however, this approach was abandoned due to excessively high implementation demands. Figure 4.6 gives an overview of the architecture.

Below we shall discuss the data model and languages related to it in more detail. The main aspects of the data model have been described in Chapter 3. The characteristic element of O_2 is that it treats objects *and* values as independent constructs. A value is of a type which may be defined recursively by means of atomic types and type constructors and is therefore possibly structured. An object has an identity, a value, a behaviour which is defined by the applicable methods, and belongs to a class. Object identities can be used in the form of attribute values as references to other objects.

4.3.1 The declaration language

The primary elements of the declaration language of O_2 have, in their 'pure' form, already been introduced in Chapter 3. In order to emphasize the existing correspondences, we choose largely the same order for the presentation of the individual constructs.

Values may be *atomic* or *structured*. Atomic values may assume one of the following forms:

- integers, for example, 1, 35, –54371;
- reals, for example, 3.1415; –26.5E10;
- individual characters, for example, 'A', 'T', always included in single quotes;
- character strings, for example, "This is a sentence", always included in double quotes;
- the boolean values true or false;
- bit strings, different from character strings due to not being limited to characters.

Moreover, O_2 knows about reference values, although they should only appear in connection with class names. Structured values could be tuple, set or list values, with sets differentiating between 'unique sets' (sets in the ordinary sense) or 'sets' (sets which could contain multiple copies of the same element). Such values are now formed by prefixing them with the respective keywords. Examples of structured values, formed by using the constructors tuple, set, unique set or list, are the following:

```
tuple (Name: "John Smith", Age: 32)
tuple (Name: tuple (Name: "John", FamilyName: "Smith"),
       Age: 32)
set (1, 2, 3, 5, 7, 11)
set (1, 1, 2, 3, 5, 7, 11, 7, 2)
unique set (1, 5, 11, 13)
list (1, 2, 7)
```

A *type* is the description of a possibly complexly structured domain which is formed using:

- the atomic types integer, real, char, string, boolean and bits;
- the type constructors tuple, unique set, set and list, and
- class names.

Class names can be either system-defined or user-defined. The system-defined class names are given either by O_2 itself or by the toolbox O_2Kit (for example, Object, Date, Bitmap or Money). If the name of a user-defined class (for example, Person) is used in a type definition, an instance of this type references an object of the relevant class. If the class name in the type definition carries the keyword type as a prefix, an instance of this type contains a value (but not an object) with the same structure as the type of the relevant class.

For the purpose of reuse, types can be given a name; this is demonstrated in the following example:

```
create type Text: list (string)
```

Type definitions are mainly used in attributes or in the definition of the structure of a class. The general form of a class definition is the following:

```
class <Classname> [ <Options> ]
  type <Type-Specification>
  method <Method-Specifications>
end
```

Details will be given below. In principle, the type clause assumes the function that the mapping *type* had in Chapter 3. Correspondingly, the keyword *method* introduces a signature description. Apart from class definitions, type definitions in O_2 are used in the definition of methods in order to define the types of the parameters and the return value.

Subtypes in O_2 are defined as known from Chapter 3. First, this means that new attributes can be defined for a subtype. Moreover, the types of attributes of the supertype can themselves be specialized by a subtype.

Example 4.5

Examples of subtype relationships are:

```
tuple( Name: string,
        Address: tuple( Street: string, Location: string),
        Telephone: string)
```

is a subtype of

```
tuple( Name: string,
        Address: tuple( Street: string, Location: string))
```

Moreover,

```
list( tuple( Name: string, Age: integer))
is a subtype of
         list( tuple( Name: string))
```

Note that, regarding the formation of subtypes, lists are treated like sets.

As usual, an *object* in O_2 has an identity which is generated by the system and not visible to the exterior, a value which must be taken from a domain described by an admissible type, and a behaviour which is described by methods bound to the object. On the language side, the symbol * takes into account the differentiation between an object and its value: if x designates a variable to which an object identity can be assigned, *x denotes the value of the relevant object (the symbol * *decapsulates* the relevant object).

We should like to mention the persistence model of O_2, which is based on the following principles:

- Objects and values can be *named.*
- Each named object or each named value is persistent.
- Objects which are referenced by persistent objects or values are also persistent.

For example, the command

```
create name Persons : set(Person);
```

creates a name Persons which is initialized with the empty set and can then take up objects of type Person; the set named Persons is thereby defined as persistent.

The distinction between objects and values is also relevant to the comparison of objects, which is necessary, for example, in selections. O_2 makes the differentiations which are common for object-oriented systems (compare Section 3.1):

- Two objects are called *shallow equal* if all their attribute values at the outermost nesting level are equal. This implies that with each object-valued attribute the same object must be referenced.

- Two objects are called *deep equal* if their values are equal across all nesting levels. When a test to establish deep equal values is carried out, object references in attribute values are replaced by the value of the referenced object and compared; this is iterated until all references are resolved. Deeply equal objects cannot therefore be distinguished according to their (possibly complex) values alone.

Example 4.6

Consider the following tuple values of objects:

```
val(#1) = [A: "xyz", B: #2, C: #3]
val(#4) = [A: "xyz", B: #2, C: #3]
val(#2) = [D:  "abc", E: "efg"]
val(#3) = [F: "hij", G: "klm"]
val(#5) = [D: "abc", E: "efg"]
val(#6) = [F: "hij", G: "klm"]
val(#7) = [A: "xyz", B: #5, C: #6]
```

Objects #1 and #4 are shallow equal, because values and references are equal in their attributes, whereas objects #1 and #7 are deep equal, because all values are also equal in the referenced objects. Note that #1 and #7 are not shallow equal, because they are different in the references which appear as values of attributes B and C.

The most general structuring unit in O_2 is the schema, an application-oriented collection of class, type, application, object, value and name definitions. For each database an individual schema must be created and named with the 'create schema' command. An instance or database of a schema is called 'base' and also has to be explicitly declared and named. Thus a base is a collection of objects and values whose structure and behaviour conform to the definitions contained in the associated schema. Class definitions and the definitions of named objects and values can be used in different schemas via the commands 'export schema' or 'import schema'. Furthermore, it is possible to use objects or values from different databases within one application via the command 'import base'.

The definition of a *class* comprises a name (which generally starts with a capital letter), a type specification and a list of methods; additionally, the position of the class in the existing class hierarchy has to be determined. It should be noted that O_2 operates with only *one* predefined class, the class Object; new classes therefore become subclasses either of Object or of already defined classes. With regard to methods, a class definition often has only a signature specification of the form

<Methodname> [(<List-of-Arguments>)] [: < Resulttype>]

which is implemented separately.

The methods associated with a class and the attributes of the type of a class are treated equally by the system if the type at the outermost nesting level is a tuple type and the methods need no parameters. Correspondingly, the attributes and methods are collectively called the *properties* of a class. All properties of a class can be declared as private or public, or rather as read or write. We shall not use these options in the following.

The rules for redefining methods are different from the contravariance/covariance rule in Chapter 3. This means that O_2 gives up the aim of (type) safety; the compiler therefore cannot guarantee that no type errors will occur at run time. The rule applicable in O_2 demands that, in a redefined method, parameter and result types are subtypes of the respective types of the signature of the supertype. Because of this, additional flexibility with modelling is achieved in many cases.

Example 4.7

Some classes of our running example shown in Figure 1.8 can be described with the definition language of O_2 as follows:

```
class Person
 type tuple( Name: string,
             Age: integer,
             Domicile: Address,
             Fleet: set(Vehicle) )
 method adult: boolean
end;
```

Note that in this example class `Person` automatically becomes a subclass of the root `Object` of the class hierarchy.

```
class Employee inherit Person
 type tuple( Qualifications: set(string),
             Salary: integer,
             FamilyMembers: set(Person) )
 method isFamMembOf(p1: Person);
 method delete
end;
```

The implementation of the first of these methods is shown below; the implementation of the second will be discussed in the following section.

```
class Vehicle
 type tuple( Model: string,
             Manufacturer: Company,
             Colour: string )
end;

class OttoEngine
```

```
 type tuple( HP: integer,
             cc: integer )
end;

class VehicleDrive
 type tuple( Engine: OttoEngine,
             Gearing: string )
end;

class Automobile inherit Vehicle
 type tuple( Drive: VehicleDrive,
             Carbody: string )
end;
```

An implementation of method adult associated with class `Person` can be given as follows:

```
method body adult: boolean in class Person
      { if (self->Age >= 18) return(true);
                  else return(false);
      };
```

As usual in object-oriented programming languages, self is a method yielding the identity of the called object. Method `adult` is inherited by subclass `Employee` and can therefore be used, for example, in the following implementation of method `isFamMembOf`:

```
method body isFamMembOf(p1: Person) in class Employee
 {
  printf(" %s is ", self->Name);
  if (self in p1->FamilyMembers) {
  if (self->adult) printf(" an adult ");
     else printf(" a minor "); }
  else printf(" no ");
 printf(" family member of %s.\n ",
          p1->Name);
};
```

The following, bigger example shows a detailed specification and subsequent implementation of methods using the language O_2C, which is based on C.

Example 4.8

We examine the definition of a class DATE to incorporate date values (or rather: date objects). The class definition should have the following form:

```
class DATE
 type string;
 method year: integer, month: integer, day: integer,
        to_string: string,
        valid: boolean,
        compare (date2:DATE) : integer
end
```

These method names can then be associated with implementations in the following way:

```
method body year: integer in class DATE
 {
  int day,month,year;
  sscanf (*self,"%d.%d.%d",&day,&month,&year),
  return year;
 };

method body month: integer in class DATE
 { ... };

method body day: integer in class DATE
 { ...};

method body to_string: string in class DATE
 { return *self; };

method body valid: boolean in class DATE
 {
  char rest[100];
  int day,month,year;
  if (sscanf (*self,"%d.%d.%d%s",
      &day,&month,&year,rest) != 3)
      return 0;
  if ((month < 1) || (month > 12)
      || (day < 1) || (day > 31)
      || year < 1981)
    return 0;
  if (((month == 4) || (month == 6)
      || (month == 9) || (month == 11))
      && (day > 30))
    return 0;
  if (month == 2)
    return (day <= 28) ||
      ((day == 29) && (year%4 == 0
      && year%100 != 0 || year%400 == 0));
    return 1;
  };

method body compare (date2:DATE): integer in class DATE
 {
  if ((self->get_year) != (date2->get_year))
```

```
      return (date2->get_year) - (self->get_year);
    if ((self->get_month) != (date2->get_year);
        return (date2->get_month) - (self->get_month);
        return (date2->get_day) - (self->get_day);
    };
```

4.3.2 The query language

We shall now discuss the essential features of the query language of O$_2$. First of all, it has to be noted that O$_2$ makes a clear distinction between a *query language* for ad hoc queries to a database (O$_2$SQL), a *database programming language* (O$_2$C), and *programming language interfaces* via which access to O$_2$ databases is possible from other languages like C++.

In principle, the query language of O$_2$ is an extended SQL with, amongst others, the following properties:

- *Path expressions* for navigation in complexly structured objects are supported, provided the references encountered are single-valued.
- It is possible to call user-defined methods in queries.
- User-defined collections of objects can be formed.
- Generic methods are available, for example for copying (copy and deep_copy) or testing for equality (equal, deep_equal etc.).

Example 4.9

As a first query example to the car sales database, consider the query 'show the name of the company in which Meier is president'. Note that, according to the persistence rules of O$_2$, class `Company` cannot be queried directly; instead a (persistent) name, say `Companies`, needs to be declared first:

```
create name Companies set(Company);
```

For the following query, we assume that the set `Companies` of `Company` objects is not empty anymore.

```
select x.Name
from x in Companies
where x.President.Name = 'Meier'
```

Here, `x` is a variable for iterating through the current extension of class `Company`. For each company object which fulfils the given selection condition the value of the attribute `Name` is returned. The above requires that an extension for class `Company` has been introduced via a name declaration (see below); in the case discussed here, the class definition and its extension are assumed to have the same name.

The query language of O_2 is based on a series of design decisions, the most important of which are listed here:

- The result of a query evaluation may be an object or a value. The value may be constructed from the values present in the database; an object as a result, however, is always one which does exist in the database. In other words, the language cannot create new objects.
- Although the language has a syntax based on SQL's, it is of a functional nature, which means that queries can be assembled from basic functions by using composition and iteration.
- The language is – as opposed to programming language interfaces – typed dynamically, which means that type checking occurs only at run time. Moreover, it allows free access to values contained in objects, which means that it does not observe the encapsulation principle.

With the following examples we should like to illustrate some aspects of the language: for example the possibility of accessing all nesting levels of a complex structure and the possibility of dynamically constructing sets, lists, tuples or nested structures.

Example 4.10

We consider the query 'show the names of all employees over 50 working in (any) subsidiaries in Boston'. This query can be phrased as follows:

```
select x.Name
from x in
 flatten(select y.Employees
        from y in
          (select s
           from c in Companies, s in c.Subsidiaries
           where s.Office. Location = "Boston"))
where x.Age > 50
```

Note that this query fomulation exploits the persistence concept of O_2: so far we have only introduced Companies as a persistent name (see Example 4.9). Everything referenced from an object in set Companies is thus persistent also.

The example also demonstrates that O_2SQL does not allow path expressions in their full generality; in particular it is not possible to write the where-clause of the innermost select as 'c. Subsidiaries.Office.Location', since this would be set-valued which is not allowed in O_2.

A simpler formulation of the same query, avoiding one level of nesting, is obtained if a name for objects in class Subsidiary is introduced first:

```
create name Subs set(Subsidiary);
```

Now the query can be written as follows:

```
select x.Name
from x in flatten
  (select y.Employees
  from y in Subs
  where y.Office.Location = "Boston")
where x.Age > 50
```

Note that the set of subsidiaries in Boston is searched by the variable y; the inner select expression extracts the set of employees from each subsidiary object and hence produces a set of sets. The flatten operator eliminates one level of set nesting and thus produces a set of employees (a similar comment applies to the first formulation of the query shown above). The structure encountered by the query is therefore manipulated. Finally, the names of those over 50 are displayed.

An even simpler formulation of the same query is the following:

```
select x.Name
from y in Subs, x in y.Employee
where y.Office.Location = "Boston" and x.Age > 50
```

Example 4.11

The following example shows the possibility of dynamically constructing tuple structures. We consider the query 'show the qualifications of presidents of companies in Boston who earn more than 100 000, together with the company names'. To answer this query we assume that the named query `BostonCompanies` has already been defined as follows:

```
define BostonCompanies as
  select x from x in Companies
  where x.Headoffice.Location = "Boston"
```

The actual query can then be phrased as follows:

```
select tuple( CoName: b.Name,
              PrQual: b.President.Qualifications)
from b in BostonCompanies
where b.President.Salary > 100000
```

Thus, a tuple structure with the two attributes `CoName` (company name) and `PrQual` (president qualifications) is formed as output. The values of these are obtained by accessing the components of those Boston companies which satisfy the selection condition.

Example 4.12

The following queries illustrate the possibility of forming nestings:

(1) 'Show the names of the companies from Boston':

```
select co.Name
from co in (select c
            from c in Companies
            where c.Headoffice.Location = "Boston")
```

(2) 'Show the names of companies which have another company's subsidiary in the same location as their head office':

```
select co.Name
from co in Companies
where co.Headoffice.Location in
 (select d.Office.Location
 from c in Companies, d in c.Subsidiaries
             where c = co)
```

An equivalent formulation of the same query is the following:

```
select co.Name
from co in Companies, d in co.Subsidiaries
where co.Headoffice.Location = d.Office.Location
```

(3) 'Show the names and addresses of the head office and subsidiaries of those companies which have subsidiaries in the same location':

```
select tuple (CoName: c.Name,
             CoAddress: c.Headoffice,
             SubsAddresses: select d.Office
                             from d in c. Subsidiaries
                             where d.Office.Location in
                                (select o.Office.Location
                                from o in c.Subsidiaries
                                where d != o))
from c in Companies
where count (select d.Office
            from d in c.Subsidiaries
            where d.Office.Location in
                  (select o.Office.Location
                   from o in c.Subsidiaries
                   where d != o)) > 0
```

As a longer example, we now consider the task of deleting an employee. Recall that in Example 4.7 we have already noted the signature of a corresponding method in the declaration of class Employee; now an implementation is to be allocated to it. The following steps need to be undertaken:

(1) The respective employee is removed from class Employee.
(2) All references to it are also removed.
(3) The employee is newly inserted into class Person.

First, we complete our class declarations of the running example (compare Example 4.7) as follows:

```
class Company
  type tuple( Name: string,
              Headoffice: Address,
              Subsidiaries: set(Subsidiary),
              President: Employee )
  method public Empl_own_Veh: set(Employee)
end;

create name Companies: set(Company);

class Subsidiary
  type tuple( Name: string,
              Office: Address,
              Manager: Employee,
              Employees: set(Employee) )
end;


create name Persons: set(Person);
```

The implementation of method delete can then be carried out as follows; it assumes that objects have been associated with name Companies, and that Employees is the name of a set of objects of type Employee:

```
method body delete in class Employee {

o2 Company co
o2 Subsidiary su;
o2 Person p;
o2 Employee empl;
o2 unique set(Employee) set_empl;

(*delete references starting from Company/Subsidiary, if necessary*)

for ( co in Companies )
  { if ( co->President != nil )
    { if ( self == co->President ) co->President = nil; }
          for (su in co->Subsidiaries )
        { if ( su->Manager != nil )
             { if ( self == su->Manager )
                   su->Manager = nil; }
              set_empl = set_empl + su->Employees;
        for ( empl in su->Employees )
             { if ( self == empl )
                   su->Employees -= unique set(self); }
        }
     }
p = new Person;
p->Name = self->Name;
p->Age = self->Age;
p->Domicile = self->Domicile;
p->Fleet = self->Fleet;

(* code for determining the set set_empl of those
employees who have 'self' as a familymember goes here *)

for ( empl in set_empl )
      { if ( self in empl->FamilyMember )
           { empl->FamilyMember -= unique set(self);
             empl->FamilyMember += unique set(p);
  }
}
Employees -= unique set(self);
Persons += unique set(p);
};


end;
```

Finally it should be noted that the query language of O_2 fulfils most of the general requirements of object-oriented database languages, which were listed in Chapter 2. However, the expressive power of the language is limited, because the integration of the database language and the programming language is achieved only within the framework of a higher language such as O_2C.

4.4 GemStone

The GemStone system was developed by the Servio Logic Development Corporation, which later changed its name to GemStone Systems Inc., and has been marketed since 1987 (as the first database system of its kind). First, we give a brief system overview and then analyse the principal aspects of working with the GemStone Smalltalk language (previously called OPAL).

4.4.1 System overview

GemStone combines the concepts of an object-oriented language, in this case Smalltalk, with the functionality of a database system; its main characteristics are:

- It supports large collections of possibly large objects; an individual object can be simple or complex and as a byte string occupy up to 1 GB of storage space.
- It supports user-defined types and behaviour, object identity and inheritance; methods can be dynamically created and changed.

- It has a uniform language for data definition and manipulation; this language, called GemStone Smalltalk, is a derivative of Smalltalk-80. Programs are themselves objects in the system.
- In addition to GemStone Smalltalk, a series of high-level languages such as other Smalltalk languages, C or C++ can be used for writing application programs; furthermore, GemStone can be coupled with SQL databases via gateways.
- It provides a multi-user environment with protection and authorization mechanisms, index and cluster management for complex objects, and supports replication.

We should like to note at this point that GemStone closely follows Smalltalk and shares the Smalltalk philosophy, namely that *everything* in the system is an object. This means in particular that there is no distinction between types and classes.

GemStone has a client–server architecture in which we can distinguish different processes:

- *Gem server processes*: These can execute methods and evaluate queries. A Gem server also contains cache memory for objects and for pages; each client process is associated with a Gem server.
- *A Stone monitor process*: It allocates new object identities (called object-oriented pointers, OOPs for short), coordinates transactions and manages error situations. The Gems and the Stone are connected to each other via interprocess communication.
- *Page server processes*: These allow a Gem server to access files on a page level; the database, which is managed by a page server, can spread over several network nodes.
- *Applications*: They form the actual clients and are all associated with a Gem server.

The Stone uses an object table for mapping OOPs to physical addresses; this table can comprise up to 2^{31} entries, which means that a database can contain up to $2^{31} \approx 2 \times 10^9$ objects. An object can be stored separately from its subobjects, but the OOPs for the values of the respective instance variables are always stored together, that is, are clustered.

The Gems can be seen as virtual machines on which a compiler or an interpreter for GemStone Smalltalk is mapped. According to Figure 4.1, the Stone together with the processes corresponding to these machines (Gem processes) forms the server.

The interface programs, with which the Gem processes communicate, use libraries of functions or methods of the programming language being used (for example, the GemStone C Interface (GCI) contains a library of C functions). These functions can be called from application programs to execute GemStone Smalltalk code (for example, as a reply to a query by the user), to send messages to GemStone objects, or to have direct access to the system kernel.

To support the user during the development, execution and debugging of Smalltalk programs, GemStone has a Smalltalk programming environment, which is a collection of special interface programs for editing programs, browsing a database or establishing classes and methods.

The following sections give an introduction to GemStone Smalltalk, which is a high-level programming language which also has database functionality so that applications can be written using only this language.

4.4.2 Introduction to GemStone Smalltalk

First, we describe the basic principles of the functionality of GemStone Smalltalk. Objects with structure and behaviour as well as classes with instances are concepts of the language. As already mentioned, there is no differentiation between classes and types; both notions can be used synonymously. Methods and structures common to all objects of one class are held in a *class-defining object* (CDO) such that the definition of a new class can be considered as an object; all instances of a class contain a reference to 'their' CDO. Moreover, each object is an instance in *exactly one* class.

Objects may have an internal structure which is described via instance variables or attributes.

Example 4.13

Figure 4.7 shows an object of a class *Employee*. This object has the three instance variables *Name*, *Domicile* and *Salary*. *Domicile* references the object of the class *Address*. Objects of the class *Address* have the two attributes *Street* and *Location*.

It should be noted that not all objects have attributes; in particular, certain base types like SmallInteger or Character have no further internal structure. Objects of this class also have no identity of their own.

```
Employee
  Name
    String
      Ray Ross
  Domicile
    Address
      Street
        String
          Alameda
      Location
        String
          Gresham
  Salary
    SmallInteger
      45578
```

Figure 4.7 Employee objects with three instance variables.

A *message expression* in GemStone Smalltalk is built out of an identifier (or an expression), which represents a *receiver object* and a *message*. A message is built out of *selectors*, which specify the message to be sent, and *arguments*; these can themselves be written as message expressions.

GemStone Smalltalk, as a Smalltalk derivative, knows three kinds of messages:

- *Unary messages*: These do not have arguments; the selector is a single identifier, for example

  ```
  7 negated
  ```

- *Binary message expressions*: There is a single selector consisting of one or two special characters and one argument, as for example:

  ```
  8 * 4
  ```

  Binary messages are also used for comparisons, for example

  ```
  4 < 5
  ```

  which returns 'true'. Analogously,

  ```
  myObject = yourObject
  ```

  returns 'true' if both objects are of the same value, whereas

  ```
  myObject == yourObject
  ```

  returns 'true' if both objects are identical (have the same OOP).

- *Keyword messages*: These have a receiver, and the selector is given by up to 15 pairs of the form 'keyword argument' with each keyword ending with ':', for example,

  ```
  7 rem: 3
  ```

  or

  ```
  arrayOfStrings at: (2+1) put: 'Curly'
  ```

Moreover, *message cascades* can be formed if a series of messages are to be sent to the same object; for example,

```
arrayOfComposers add: 'Mozart';
arrayOfComposers add: 'Beethoven'.
```

has the same effect as

```
arrayOfComposers add: 'Mozart'; add: 'Beethoven'.
```

4.4.3 Structure definition in GemStone Smalltalk

Similarly to Smalltalk, GemStone Smalltalk offers an extensive hierarchy of predefined classes (Kernel classes), which are shown in Figure 4.8. Let's look at some of these in more detail:

```
Object
     Association
       SymbolAssociation
     Behaviour
       Class
       Metaclass
     Boolean
     Collection
       SequenceableCollection
          Array
            InvariantArray
            Repository
          String
            InvariantString
              Symbol
       Bag
         Set
           Dictionary
             SymbolDictionary
               LanguageDictionary
           SymbolSet
           UserProfileSet
     CompiledMethod
     Magnitude
       Character
       DateTime
       Number
          Float
          Fraction
          Integer
            LargeNegativeInteger
            LargePositiveInteger
            SmallInteger
     MethodContext
       Block
          SelectionBlock
     Segment
     Stream
       PositionableStream
          ReadStream
          WriteStream
     System
     UndefinedObject
     UserProfile
```

Figure 4.8 Hierarchy of Kernel Classes.

- *Object, Set*: These classes do not have attributes, they only have methods.
- *Association*: The attributes of this class are key and value; instances are pairs of associated objects.
- *Fraction*: The attributes of this class are numerator and denominator.
- *Magnitude*: This class uses methods for linearly ordered objects, as for example Integers; it also uses comparison operators such as '<' or '>=' and conversion functions like asInteger or asLowercase.
- *Boolean*: This class has only the two instances true and false and as methods the boolean functions such as or, xor or not; no subclasses of this class can be defined.

Each class in the hierarchy inherits structure and behaviour from its superclass. Multiple inheritance is not possible; if required, it has to be simulated by single inheritance (cf. section 2.4.4). When declaring a new class, a 'suitable' position in the hierarchy, where the new class will be located, has to be specified; the default position is the root *object* of the hierarchy.

To introduce a new subclass of an existing (kernel or user-defined) class the message subclass has to be sent to this class. As the result of this message the class defines a new subclass of itself. This message is a keyword message in which, among others, the following keywords can be defined:

```
subclass:              astring
instVarNames:          anArrayOfStrings
classVars:             anArrayOfClassVars
inDictionary:          aDictionary
constraints:           aConstraints
instancesInvariant:    invarBoolean
isModifiable:          modifyBoolean
```

In this message instVarNames: may have up to 255 arguments, which are expressed in the following form:

```
#( 'string1' 'string2' ...)
```

Each of these strings represents the name of one *instance variable*. Extending Smalltalk, attributes may be typed; the type is given as a *constraint* as part of the argument of the constraint keyword, as shown in the following example for the attribute Name of class *Employee*:

```
constraints: #[ #[ #Name, String ]]
```

We should comment on the syntactical usage of the #-symbol. # is used as an array delimiter. The argument to constraints is an array whose elements are arrays of pairs of elements. The first of these elements is always a symbol that represents the name of the corresponding instance variable. Note that instance variable names are used as strings when defined as part of the argument to instVarNames; however, when it is intended to refer to the variable itself it must be preceded by #.

Class variables indicated in classVars contain values which can be accessed by all objects of the respective class at run time. A class Employees could, for example, have an instance variable Profession and a class variable Average salary, with the latter being continuously updated by an appropriate method.

In inDictionary it could be stated in which directory this class is to be installed; for example,

```
inDictionary: UserGlobals
```

makes the relevant class part of a global area.

By using keyword isModifiable we can state whether class definitions can be modified or not. If the argument is the boolean true, then values for instVarNames or constraints can be changed afterwards.

Finally, we can specify in a class declaration whether or not the instances of this class may be changed; this is done by stating the booleans true or false in instancesInvariant.

Example 4.14

As a first example of using GemStone Smalltalk we refer to the relational schema `Employees`, where each employee is described by the attributes `Name`, `Department` and `Salary`. In GemStone Smalltalk we may define it as follows:

(1) Declare a class for *tuple objects* whose instances are individual tuples with the attributes mentioned.

(2) Declare a *set object* whose elements are the already declared tuple objects.

Since the given class hierarchy does not know the class tuple, a new subclass of `Object` is declared in the first step:

```
Object subclass: 'Employee'
instVarNames:    #('Name' 'Department' 'Salary')
classVars:       #()
inDictionary:    UserGlobals
constraints:     #[  #[ #Name, String ],
                     #[ #Department, String ],
                     #[ #Salary, SmallInteger ]]
instancesInvariant: false
isModifiable:    false.
```

As a next step a subclass of the class set is declared for sets of employee tuples as follows:

```
Set subclass: 'Employees'
instVarNames:  #()
classVars:     #()
inDictionary:  UserGlobals
constraints:   Employee
instancesInvariant:  false
isModifiable:   false.
```

Instances of class `Employees` are sets. These sets do not have instance variables because no argument is stated to `instVarNames`; however, the elements of the sets have instance variables, namely instances of class `Employee`, as can be concluded from the keyword constraints. If we would like to allow arbitrary elements in the sets we could state class object as an argument of keyword constraints or even omit it.

Example 4.15

To give a more complex example we refer to Figure 1.8. In the following declarations we shall omit clauses when their argument has not changed or is empty. We start defining `VehicleDrive` to consist of an `OttoEngine`:

```
Object subclass: 'OttoEngine'
instVarNames:  #( 'HP' 'CC')
constraints: #[ #[ #HP, SmallInteger ],
                #[ #CC, SmallInteger ] ].

Object subclass: 'VehicleDrive'
instVarNames:  #( 'Engine' 'Drive' )
constraints: #[ #[ #Engine, OttoEngine ] ,
                #[ #Drive, String ] ].
```

The following class `Vehicle` references *inter alia* the class `Company`, which is not yet defined:

```
Object subclass: 'Vehicle'
instVarNames:  #( 'Model' 'Manufacturer' 'Colour' )
constraints: #[ #[ #Model, String ],
                #[ #Manufacturer, Company ],
                #[ #Colour, String ] ].
```

Now, class `Automobile` can be defined as a subclass of `Vehicle`:

```
Vehicle subclass: 'Automobile'
instVarNames:  #( 'Drive' 'Carbody' )
constraints: #[ #[ #Drive, VehicleDrive],
                #[ #Carbody, String ] ].
```

```
Object subclass: 'Address'
instVarNames:  #( 'Street' 'Location' )
constraints: #[ #[ #Street, String ],
                #[ #Location, String ] ].
```

To prepare the definition of class `Person`, which should have a set-valued attribute `Fleet`, a set of vehicles is defined:

```
Set subclass: 'Vehicles'
instVarNames:  #( )
constraints: Vehicle.
```

Class `Person` can now be defined as follows:

```
Object subclass: 'Person'
instVarNames:   #( 'Name' 'Age' 'Domicile' 'Fleet' )
constraints: #[ #[ #Name, String ],
                #[ #Age, SmallInteger ],
                #[ #Domicile, Address ],
                #[ #Fleet, Vehicles ] ].
```

The result of the above declarations is shown in Figure 4.9.

```
Object
    OttoEngine
    VehicleDrive
    Vehicle
      Automobile
    Address
    Collection
      Bag
        Set
          Vehicles
    Person
```

Figure 4.9 Extract from the class hierarchy of the Vehicle example, including the declarations.

4.4.4 Method generation in GemStone Smalltalk

After the declaration of structure is finished, methods may be defined, to generate instances, to initialize classes or simply to use classes and objects.

Two kinds of methods can be distinguished (as in Smalltalk): *class methods* and *instance methods*. The former are understood only by classes, the latter only by instances. Most classes understand the message *new* to create a new instance of a class, as for example in

```
Vehicle new.
```

In general, the definition of a method always contains a *message pattern*, with which the method can be activated, followed by a *message body*, which contains the selector and (optionally) formal parameters, (optionally) temporary variables, one or more statements and a return statement. To give an example let us look at method new in more detail.

Example 4.16

Sending new to a class creates a new object of this class where all attributes of this object are initialized with *nil.* If values different from nil are to be assigned we have to introduce a class method, as in the following example with respect to class OttoEngine:

```
classmethod: OttoEngine
makeEngine
     ^ (self new) HP: 136;
                  CC: 1998
%
```

The first line instructs the programming environment to understand the subsequent text as a method to be compiled and installed in the respective class; % is the command delimiter. The second line is the *message pattern*, which can then be used for instantiation of the class as follows:

```
OttoEngine     makeEngine.
```

The third line starts with a *hat* (^), which denotes the return value of the method; this is specified more precisely by the items following ^. In the example, this is (self new). *self* is a special variable to which all methods have access and which represents the receiver. (self new) sends the message new to the receiver, which in our case is class OttoEngine, and a new instance is created. The remaining parts of the line assign values to the corresponding attributes with the help of keyword messages. The return value of the method is therefore a new instance of OttoEngine with the indicated values. If the following message is then sent to OttoEngine,

```
(OttoEngine makeEngine) HP
```

a new instance, as explained above, is created, from which the value 136 of the attribute HP is returned; to make this possible the following instance method must have been defined for OttoEngine.

```
method: OttoEngine
HP          (* message pattern *)
    ^ HP    (* return statement *)
%
```

Obviously, this kind of initialization of objects is awkward, because the method has no formal parameters through which it would be reusable for different calls. This problem is solved by defining a method for instance creation whose arguments specify the values to be assigned to attributes.

Example 4.17

To continue the preceding example, such a method is obtained as follows:

```
classmethod: OttoEngine
newHP: aNumber newCC: anotherNumber
| tempEngine |
tempEngine := self new.
tempEngine HP: aNumber; CC: anotherNumber;
^ tempEngine
%
```

Whenever this method is executed, a new object is created. First, the temporary variable with the name `tempEngine`, which is defined in line 3, is instantiated by calling new. A keyword message with the relevant values is then sent to the resulting object; finally, the new object is delivered. To create a new Otto engine, a method call of the following kind is now sufficient:

```
OttoEngine newHP: 136 newCC: 1998
```

4.4.5 Data manipulation

Once the structure of a database and the methods for its initialization are defined, methods to manipulate objects can be introduced. The following example shows some elementary methods of that kind:

Example 4.18

The following methods are instance methods which are applicable to objects of class `Employee`:

```
Object subclass: 'Employee'
instVarNames:    #('Name' 'Department' 'Salary')
classVars:       #()
inDictionary:    UserGlobals
constraints: #[  #[ #Name, String ],
                 #[ #Department, String ],
                 #[ #Salary, SmallInteger ]]
instancesInvariant: false
isModifiable:    false.
```

We assume that there already exist instances of this class.

```
method: Employee
Name
     ^ Name     (* supplies the name of the receiver *)
%
```

```
method: Employee
Department
     ^Department  (* supplies the department of the receiver *)
%

method: Employee
Salary
     ^Salary (* supplies the salary of the receiver *)
%

method: Employee
Name: aString   (* message pattern *)
Name := aString (* allocates a new value to the name of the receiver)
%

method: Employee
Department: aString
Department := aString
%

method: Employee
Salary: aSmallInt
Salary := aSamllInt
%
```

The first three messages (of type accessing) are trivial, but are necessary for accessing the attribute values of an employee object. We shall need them later. The last two messages (of type updating) can be used to fulfil (at least) two functions: update of objects (here especially, transfer of an employee from one department to another, or for salary adjustment) and as an alternative means of assigning values to newly created objects.

Among the Kernel classes there is a class behaviour which provides a method

```
compileAccessingMethodsFor: anArrayOfSymbols
```

which automatically creates accessing and updating methods. For each element stated in anArrayOfSymbols, the method creates methods to access instance variables and to assign values. The following is an example in which the abovementioned six methods would be created:

```
Employee compileAccessingMethodsFor: #(#Name #Department #Salary)
```

Example 4.19

The following (class) method creates a new instance of the class `Employee`:

```
classmethod: Employee
newName: aName
| tempEmpl |
tempEmpl := self new.
emplEmpl Name: aName
^ emplEmpl
%
```

The (instance) methods of the last example (`Department` and `Salary`) can be used to assign values to the attributes `Department` and `Salary`, which are initialized with *nil.*

Database objects cannot be explicitly deleted in GemStone. Instead, the approach used here can be compared to the approach used in network database systems: an object is deleted by means of destroying its *reachability*. It will, however, remain in memory until the system releases the memory it occupies for reuse by means of *garbage collection*. Objects become unavailable by assigning nil to all referencing attribute values.

Example 4.20

The class Set inherits a method `remove` from its superclass `Bag`. Let `E` denote an instance of class `Employees`, which is a subclass of `Set`; `E` is a set of employees. Let `Victim` denote an object in `E` which is to be deleted. The following message performs the desired task:

```
E remove: Victim
```

If `E` was the only place where `Victim` was referenced, it has now become unavailable and thus its storage will be released.

The query language in GemStone Smalltalk can be regarded as a combination of methods designed for the various subclasses of Collection. We shall look at the methods do, select, reject and detect, but there are several other methods that could be used.

Example 4.21

We want to produce a list of employees. First we provide a method which returns the attribute values as one string:

```
method: Employee
asString
^ (self Name) + ' ' +
         (self Department) + ' ' +
         (self Salary asString)
%
```

The symbol + denotes concatenation of strings. Note that the value of `Salary` is still to be converted from `SmallInteger` to `String`. The following method prints a set of employee objects as a table, with one object per line:

```
method: Employees
asTable
| aString |
aString := String new.
self do: [ :n | aString := aString + n asString.
         aString := aString add: Character lf ].
^ aString
%
```

First, a new instance of the class string is created and assigned to the temporary variable `aString`. Then the `do` method is sent to `self`; the argument is enclosed in square brackets and is called a *block*. The effect of `do` is to iterate over the elements of the receiver and evaluate the block for each individual element. In the example above, the variable `n` ranges over the elements in the receiver set in the following way. The current value of object `aString` is concatenated with the string that results from sending method `asString` to the current object binding `n`. Then the result of this step is concatenated with a linefeed character to obtain the desired tabular form.

The previous example shows how we can iterate over the elements of a collection object; we shall now concentrate on how to extract subsets and elements of sets. To simplify method code Gemstone Smalltalk provides path expressions, which are an extension to ordinary Smalltalk.

Example 4.22

The following code *selects* the set of all employees who work in the research department; it is assumed that an instance of `Employees` with the name `Empl` already exists:

```
| researchEmpl |
researchEmpl := Empl select: [ :anEmpl
                        | anEmpl.Department = 'Research' ].
researchEmpl asTable.
```

The select method is inherited from `Collection`. It evaluates a block (like the do method) on the elements of the receiver, which serve as arguments in sequential order. The values for which the block evaluation is true are collected in an object of the same type as the receiver, and this object is finally delivered. In the above example, select is sent to `Empl`; the desired employees are computed by accessing the values of the attribute `Department` of each object in `Empl` in a path expression. A table is delivered, created by the method `asTable` as shown in the last example.

Example 4.23

The following method computes all employees of the research department who earn more than 50 000:

```
| researchEmpl50 |
researchEmpl50 := Empl select: [ :anEmpl |
                                 (anEmpl.Department = 'Research')
                                 & (anEmpl.Salary > 50000) ].
researchEmpl50 asTable.
```

Here '&' denotes logical *And*; '|' would denote logical *Or* and '~' *Not*.

If we are interested in the set of all objects for which a given selection block evaluates to false, the method *reject* can be used instead of *select*:

Example 4.24

The following query will compute the set of all employees who do *not* work in the research department:

```
|nonResearchEmpl |
nonResearchEmpl := Empl reject: [ :anEmpl
                           | anEmpl.Department = 'Research' ].
nonResearchEmpl asTable.
```

Finally let us look at method *detect*, which enables us to extract a certain element out of a set:

Example 4.25

The following expression defines an employee named Smith, if such an object exists. If there is more than one such object, the first one found will be returned:

```
Empl detect: [ :anEmpl | anEmpl.Name = 'Smith' ].
```

If no such object exists, an error message is returned and the interpreter stops; this can be avoided by adding an *exception block*, for example:

```
Empl detect: [ :anEmpl | anEmpl.Name = 'Smith' ]
              ifNone: [ nil ].
```

So far we have considered only path expressions referring to scalar attributes. Set-valued attributes may also occur inside a path expression, for example:

```
aPers.Fleet.*.Model = '2CV'
```

In this expression, for each given object aPers, the model of each vehicle in the set aPers.Fleet is considered to see whether its model is '2CV'. Note that * can be considered as a *flatten operator*.

In order to speed up processing of a query, *indices* which use the values of attributes as keys can be used. Indices can be defined for instances of subclasses of Collection. Two kinds of indices are provided: an *identity index* to support queries concerning the identity of objects, and an *equality index*, which supports queries concerning equality.

Example 4.26

The following message sent to `Empl` creates an identity index for the attribute `Name`:

```
Empl createIdentityIndexOn: 'Name'
```

whereas

```
Empl createIdentityIndexOn: 'Address.Location'
```

creates an index on the location part of the address for objects in `Empl`.

We can take advantage of indices when processing a select method. To indicate syntactically that an index should be used, the square brackets are replaced by curly ones. For example, to use a possibly existing index on attribute Department we can write:

```
researchEmpl := Empl select: { :anEmpl
                             | anEmpl.Department = 'Research' }.
```

Finally, we should like to demonstrate how to sort objects for instances of class Bag.

Example 4.27

In the following the method `sortAscending` expects as arguments an array of path expressions; sorting is done with respect to the first path and then with respect to the second one:

```
| returnArray tempString |
tempString := String new.
returnArray := Empl sortAscending:
               #( 'Name' 'Department' ).
returnArray do: [ :n | tempString add: (n Name);
               add: ' '; add: (n Department);
               add: Character lf ].
^ tempString.
```

First, employees are sorted according to the values of `Name`; employees with the same name are sorted according to the values of `Department`. A list of the sorted objects is then delivered. Method `add` is defined for class `SequenceableCollection` (see Figure 4.8) and is used here to concatenate strings.

If we want to sort names in ascending order, but departments in descending order, we could write:

```
returnArray := Empl sortWith:
                        #( 'Name' 'Ascending'
                        'Department' 'Descending').
```

4.5 ObjectStore

ObjectStore is another attempt to extend an object-oriented programming language to embrace the functionality of a database system. ObjectStore is based on C++, because this language is considered to be of the greatest importance in the application areas envisaged for ObjectStore. The data model on which ObjectStore is based is essentially the object model of C++. The persistence of an object is seen orthogonally to the type of the object. Thus objects of any C++ type can be either transient or persistent; that is, there can be transient and persistent objects for one and the same type within a program.

ObjectStore has a page-server architecture, which means that the processing of objects takes place exclusively at the clients. Referencing of objects is organized as in C++. Thus, in principle, no additional effort is necessary for the processing of persistent objects as compared with transient (ordinary C++) objects. But this is true only if an object which is persistently stored in the database has already been moved into the main memory of the client. The client always assumes that this is the case. If this assumption is incorrect – and at the beginning of the processing this is almost always the case – a page fault will be raised which results in a request to the object needed: that is, the relevant page of the server.

We are especially interested in the extensions to the object model of C++ made by ObjectStore. Among them is an extension of the given class library by a class for *collections*; subclasses of this class are classes for sets, multisets (bags), lists and arrays. In queries such collections can be accessed associatively. Another interesting extension is *bi-directional relationships* between objects to support referential integrity.

Example 4.28

The following class definitions demonstrate the use of bi-directional relationships (`inverse_member`) and of sets (`os_Set`).

```
class Employee
{
```

```
public:
  string Name;
  int Salary;
  Subsidiary * EmplSubsidiary
   inverse_member Subsidiary::Employees;
};
```

EmplSubsidiary is an attribute of reference type of class Subsidiary, which states for each employee the subsidiary in which this employee works. EmplSubsidiary is defined as an inverse attribute; the complementary part is the multi-valued attribute Employees representing the relationship between Subsidiary and Employee. It should be noted that Employees is here of set type {Employee}, as can be seen in the following definition of the class Subsidiary (see also Figure 1.8):

```
class Subsidiary
{
public:
  char* Name;
  Address Location;
  Employee* Manager;
  os_Set<Employee*> Employees
   inverse_member Employee:: EmplSubsidiary;
  void Add_Employee (Employee *e)
   {Employees -> insert(e);}

  int Works_Here (Employee *e)
   {return Employees -> contains(e);}
};
```

Correspondingly, in Subsidiary, the attribute Employees is inversely defined to the relationship between Employee and Subsidiary represented by the attribute EmplSubsidiary.

The method Add_Employee, applied to a concrete subsidiary, adds a reference to a new employee to the set of employees in the subsidiary, and the method Works_Here tests whether a reference to an employee, which has been passed on as a parameter, belongs to the set of references of employees of the subsidiary. Note that these two methods use the special methods insert and contains, which are defined in the class for collections.

Inverse relationships support referential integrity. If, for example, an employee related to a Subsidiary is deleted, the reference to it is automatically removed from the set Employees. Another interesting effect is that, when a reference to an employee is inserted into a set of Employees of a subsidiary (see above), the attribute EmplSubsidiary automatically receives the reference to the relevant subsidiary as a value.

Associative access to objects of a collection type is achieved by using boolean C++ expressions. The result of such a query expression is the subset of the collection for which the boolean expression evaluates to true.

Example 4.29

In the following example we first define a set of employees and then the subset of overpaid employees by selecting those from the set of all employees who satisfy the stated condition.

```
os_Set<Employee*> AllEmployees;

os_Set<Employee*> &
  Overpaid = AllEmployees [: Salary >= 100000 :];
```

The boolean expression `[: Salary >= 100000 :]` is evaluated with regard to each object of the set `AllEmployees`. To achieve this, ObjectStore maintains a range variable which is basically implicit and which is assigned the references to the individual objects one after the other. If explicit reference to this variable is required, *this* may be used. We can also write the boolean expression as

```
[: this -> Salary >= 100000 :].
```

Example 4.30

It is possible to iterate over the individual elements of a collection, as demonstrated by the following program segment to increase salaries by 10 per cent.

```
Subsidiary* d;
...
foreach (Employee* e, d -> Employees)
  e->Salary *= 1.1;
```

foreach is an ObjectStore construct for iterating through the elements of a collection. `e` is a variable over the elements of the collection; these are precisely the references to the objects of the type `Employee` which form the employees of the subsidiary `d`. Note that `d -> Employees` and `e -> Salary` are path expressions in a slightly different notation: if we replace the arrow by a dot, we obtain the notation we have used so far.

A path expression in general is formed by several -> expressions in sequence. The expression

```
a -> EmplSubsidiary -> Name
```

determines the name of the subsidiary of the employee associated with a. Path expressions also result implicitly from using nested queries.

Example 4.31

We wish to determine, from a particular set of employees, those who are employed in a subsidiary with a manager called 'Primus'.

```
os_Set <Employee*> TheseEmployees;

TheseEmployees[:
     EmplSubsidiary[:
           Manager -> Name == 'Primus' :] :]
```

The path expression contained in this query has the following form:

```
TheseEmployees -> EmplSubsidiary -> Manager -> Name
```

But note the difference: whereas this path expression determines for each employee of the set `TheseEmployees` the name of the manager of its subsidiary, the nested query shown above defines a subset of the set `TheseEmployees`.

In nested queries it is sometimes necessary to refer to the object of an outer level. The following expression determines those employees who are managers of their subsidiaries.

```
TheseEmployees[: d = this,
      EmplSubsidiary[:
           Manager == d :] :]
```

'this' is the range variable connected with the outer query. In order to be able to refer to the range variable of the outer query from the inner query, this variable must be given an alias name, here d, in order to distinguish it from the range variable of the inner query, which also has the name 'this'.

In ObjectStore conventional join operations can be expressed, in which relationships between objects, defined via attribute values, are established. Be aware that there is actually no real need for such operations in ObjectStore applications, because join operations are expressed by attributes of reference type. Consider, for example, the path expressions given above. The individual attributes each have as their value a reference to their related object; the desired join has thus already been materialized.

Example 4.32

The following expression joins projects and employees, with the join being defined via the attributes `ProjectNo` and `Works`. We assume that both attributes are of type int.

```
Project [: Employee [: ProjectNo == Works &&
                            Name == 'Fred' :] :]
```

4.6 Bibliographical notes

Overviews of commercially available object-oriented database systems and of research and development projects in this area can be found in Kemper and Moerkotte (1994); numerous references to early relevant literature can also be found in the bibliography of Vossen (1993). The client– server architecture which is currently popular in database systems originated from process communication in operating systems; for details, see Tanenbaum (1995). The section on Illustra is based on Manola (1994) and on Stonebraker (1996). As was mentioned in Section 4.2, Illustra is a representative of the emerging species of object-relational database systems; another example of that class of systems is the common server version of IBM's DB2, described in detail in Chamberlin (1996). More detailed information about O_2 can be found, in particular, in the original publications on this system, for example in Bancilhon et al. (1988), Lecluse et al. (1988), Lecluse and Richard (1989), Velez et al. (1989) or Deux et al. (1990); Bancilhon et al. (1992) is a compendium of these original publications, which were published during the development of this system. For further information on GemStone we refer the reader to Ullman (1988), Butterworth et al. (1991), Maier et al. (1986), Penney and Stein (1987) or Purdy et al. (1987). For ObjectStore we refer to Lamb et al. (1991), Orenstein et al. (1992) and Soloviev (1992).

Up-to-date information on the systems we have discussed in this chapter can be found on the World Wide Web; the respective URLs are:

For Illustra:	http://www.informix.com
For O_2:	http://www.o2tech.fr
For GemStone:	http://www.gemstone.com
For ObjectStore:	http://www.odi.com

5 Standardization

In this chapter we look at the efforts being made to achieve standardization in the field of object-oriented *databases*. It has been argued that the lack of accepted standards is one of the reasons why these systems fail to play a more dominant role in the marketplace. Therefore, the development of standards has become a priority, and even more so when it was proved that they had a positive impact on products as soon as they had been introduced to object-oriented *systems*.

We examine three different standards in this chapter. First, we describe SQL3, a development of the relational language standard SQL which will reflect object-oriented properties for the first time. SQL3 may be considered as moving towards object-relational systems of the type described in the preceding chapter. Next, we discuss the standardization proposal ODMG-93, which has been put forward by the Object Database Management Group (ODMG). This group is a consortium of vendors whose main objective is to develop standards for object-oriented databases. We highlight essential aspects of their most recent proposals, which are expected to be incorporated in commercially available products in the near future.

Before dealing with the ODMG we turn our attention to the 'superordinated' activities of the Object Management Group (OMG), which is concerned with standardization of architectures in object-oriented *systems* and so far has been more successful than the ODMG. In particular, we take a brief look at some essential aspects of the Object Management Architecture (OMA) and Common Object Request Broker Architecture

(CORBA) specifications. A reason for us to emphasize the work of the OMG is that their standards are stable and have already been integrated in commercially available products, whereas OMDG standards are still under discussion. Furthermore, we want to point out the different character of the two groups and their standardization work: while ODMG strives to establish a *portability* standard, OMG has always intended to introduce an *interoperability* standard. In view of the increasing penetration of the software market by object orientation in general and object-oriented products in particular, both in database systems and in other types of system, and in view of the current paradigm shift in software development (where configurable *components* derived from reference models are starting to replace large monolithic systems), these aspects will be of greatest importance in the future.

5.1 SQL3

The language standard SQL (the accepted abbreviation for Standard Query Language) has been used for relational databases for more than ten years and is now supported by most developers of relational (and other) database systems. In 1992 SQL Version 2 (known as SQL92 or SQL2) was adopted, and work on a follow-up version was begun at about that time as well. Version 3 of the SQL standard (SQL3 for short) is to reflect the increasing use of object-oriented concepts in relational databases in the future. Therefore, we describe the essential features of SQL3 next. The reader should bear in mind, however, that the SQL3 standard had not yet been adopted at the time of writing this book (Spring 1997) and that some information given in this section may no longer apply when the standard is finally adopted. At the moment, it is expected that the standard will be completed in 1999.

5.1.1 Extensions of SQL2

In this section we assume that the reader is familiar with the basics of SQL. (In Section 1.3 of this book, the basic features of SQL concerning table declarations, update operations and queries are explained.) Compared with SQL2, the new features of SQL3 include not only obvious further developments and extensions of existing concepts, but also some completely new concepts. Some developments belonging to the former category are listed below:

- The *trigger concept* provides extended possibilities for monitoring integrity. Triggers are used in database systems, for example, to check automatically whether key or foreign-key conditions are satisfied after update operations have been carried out. If necessary, warnings are issued or corrections are automatically performed.

- Improved security measures based on the Grant and Revoke commands. These SQL commands enable allocation or withdrawal of access rights to tables or views.
- Extended query possibilities, for example by using *recursion*, additional predicates (for example, 'for all', 'for some') or extended join operations.
- A restricted possibility to define complex structured values by means of the Row data type, which renders an attribute tuple-valued (see next example).
- New, predefined data types, in particular enumeration types, Boolean values and so-called Large Objects (LOBs) for holding large storage objects. These objects can be either binary (Binary LOBs or BLOBs) or of type Character (Character LOBs or CLOBs).

Example 5.1

We shall exemplify the application of the `Row` data type by means of the following declaration of a table for persons:

```
create table Person
 (Name varchar(40),
 Age int,
 Address row(Street   varchar(30),
             Location varchar(20),
             Telephone row(Areacode char(5),
                           Number char(7))));
```

The attribute `Address` here is tuple-valued and consists of the three attributes `Street`, `Location` and `Telephone`; `Telephone` is itself tuple-valued.

5.1.2 Value and object types

The second category mentioned above comprises concepts which are not available in SQL2. They include the aspects listed below, which are also known collectively as the Major Object-Oriented SQL Extensions (MOOSE):

- Extension of the current SQL type system to include *abstract data types* (ADTs). We can distinguish between *value types*, which act as a generalization of the SQL2 domain concept, and *object types* supporting objects with a value-independent identity.
- In an ADT structure declaration the *type constructors* `LIST`, `SET` and `MULTISET` can be employed.
- *Specialization* of types such that each subtype has exactly one (direct or indirect) 'maximal' supertype which itself has no other supertype. This method of constructing subtypes applies equally to value and object types.
- Table hierarchies as a form of specialization for relations: that is, a table can be defined as a *subtable* of one or more other tables (see also the next subsection).

- User-defined *functions* as part of an ADT. Functions may be realized either in SQL or as external functions; type checking is essential. Overloading of function names, however, is permitted.
- Method and function *calls* inside SELECT-expressions.

We shall illustrate the SQL approach to abstract data types with the help of some examples. As mentioned already, *user-defined data types* (or UDTs for short) are divided into value and object types and have the following essential properties:

The values (instances) of value types behave like values of the predefined types of this language: that is, they 'exist permanently' and need not be explicitly created or destroyed. On the other hand, instances of an object type behave like objects in other object-oriented languages: that is, they have a well-defined lifetime, must be created explicitly and can cease to exist. Unlike value types, object types have a distinct identity, which is independent of the value, and therefore they can be referenced from multiple locations and hence shared by other objects.

As the following example illustrates, complex objects without identity can be constructed with the help of value types.

Example 5.2

In our running example the following declarations create a table Automobile whose attribute Drive takes as value an instance of value type VehicleDrive; the latter type has the attribute Engine which takes as value an instance of value type OttoEngine:

```
create value type OttoEngine
  (HP int,
  CC int);

create value type VehicleDrive
  (Engine OttoEngine,
  Gearing varchar(12));

create table Automobile
  (Drive VehicleDrive,
  Carbody varchar(20));
```

Concerning the use of value types, it should be noted that for each attribute of such a type a system-internal *observer* and a *mutator* function are created. Formally, these are available externally at the interface of the (encapsulated) ADT for querying and modifying a current value. A *constructor* function, which is also generated internally, serves to create or initialize values. In addition to these constantly available functions, functions specific to value types or to object types can be introduced. Furthermore, with the help of the options Public and Private it can be determined whether attributes and functions should be visible from the outside. Finally, it is possible to create subtypes of value types, and to do this across multiple levels. Every subtype inherits the attributes and functions of each supertype; multiple inheritance is allowed.

Next, we shall look at object types and in particular the following example:

Example 5.3

In our running example object types may be used in the following manner:

```
create object type Person
       (Name varchar(30),
       Age int,
       Domicile Address);
```

`Address` here designates the following value type:

```
create value type Address
        (Street varchar(25),
        City varchar(25));
```

The next declaration introduces a subtype of object type `Person`:

```
create object type Employee under Person
       (Qualification varchar(30),
       Salary decimal(7,2),
       FamilyMembers set(Person));
```

Finally, the following declaration uses these types within a table:

```
create table Company
       (Name varchar(30),
       Headoffice Address,
       Subsidiaries set(varchar(30)),
       President Employee);
```

These examples indicate the complexity that future relational database structures may have. One should be aware, however, that even SQL3 contains redundancies which could lead to conflicts between developers and users of database products, because, like SQL2, the standard defines the *syntax* of a language but not its semantics.

The object identity of instances of object types can even be made visible. This can be achieved by declaring the corresponding type with the option `WITH OID VISIBLE`.

5.1.3 Subtables

We continue our discussion of SQL3 by describing the possibility, mentioned earlier, of declaring tables as *subtables* of already existing ones. In many respects the concept of tables goes further in SQL3 than it does in SQL2:

- As opposed to the relational data model, tables in SQL3 can be more than just sets of tuples. A table can now be a *multiset* or a *list* of tuples. For this purpose one of the options `SET`, `MULTISET` or `LIST` must be included in the declaration of a table, where the second of these is the default setting. Note that this novel feature merely makes explicit what has been a reality in most commercial relational systems for a long time, namely that tables are multisets of tuples (allowing duplicate entries), which are even stored in a specific sequence (thus being ordered and de facto constituting a list).
- Every tuple within a table can be equipped with a unique *row identifier*, which may be used either implicitly for identification or explicitly, for example as a foreign-key value. Row identifiers are specific data types with system-wide unique values. For each base table a row identifier can be declared explicitly by using the option `WITH IDENTITY`. Any such table then has an additional attribute with the name `IDENTITY`, which will not be visible in the result of Select *-queries. On the other hand, this attribute can be accessed implicitly, for example to create aggregations.

Example 5.4

Unlike Example 5.3, the following declares `Person` as a table with the new attribute `Partner`:

```
create table Person with Identity
       (Name varchar(30),
       Age int,
       Domicile Address,
       Partner Person Identity);
```

`Identity` is now another attribute of the table `Person`, and the value of this attribute for every tuple is a unique value of the implicitly defined type `Person Identity`. This additional type can be used just like an ordinary type, in this case as the value of the attribute `Partner`.

In particular, the notion of a row identifier is used for the creation and maintenance of subtables, which are also new in SQL3. A table may be declared as a subtable of one or more other tables, as illustrated in the next example:

Example 5.5

The following declares a subtable of the table declared in the previous example:

```
create table Employee under Person
       (Qualification varchar(10),
       Salary int,
       FamilyMembers set(Person Identity));
```

The table Employee inherits all attributes from the table Person and additionally defines others. Instance-wise, each tuple in the table Employee must have a corresponding tuple in the table Person.

The above-mentioned row identifiers ensure tuple correspondence between tables and subtables as mentioned in the previous example. Every base table that has an associated sub- or supertable is implicitly marked with a row identifier: that is, with an attribute of the type Identity. In the previous example the table Person had an implicitly defined attribute of type Person Identity, and the table Employee an attribute of the type Employee Identity, where the latter type is a subtype of the first type.

When using subtables and row identifiers it is important that the update operations are defined in such a way that the tuples of a subtable lattice are (and remain) consistent with each other. The following rules are under discussion for SQL3:

- If a tuple is inserted into a table which is a subtable of another table, then a tuple with the same row identifier needs to be inserted into the supertable. Since the tuple of the subtable has more attribute values than needed for the supertable, the tuple of the supertable results from the new tuple by a projection.
- If a tuple in a table which has subtables is modified, all inherited attribute values in the subtable are modified accordingly.
- If a tuple in a table which is a subtable is modified, the corresponding tuples in the supertables will be modified as well.
- If a tuple is deleted from a table which is part of a subtable lattice, all corresponding tuples will be deleted as well.

We finally list some examples of SQL3 queries.

Example 5.6

The following queries refer to the tables declared in the last two examples:

(1) Show the names of all persons living in Boston:

```
select Name
from Person
where Domicile.City = 'Boston'
```

Note here the use of dot notation (Domicile.City) to access the components of an ADT.

(2) Show those employees whose family members include a person named 'Peter Smith':

```
select *
from Employee a
```

```
where 'Peter Smith' in
      (select b.Name
      from (a.FamilyMembers) b)
```

This query illustrates access to elements of a set-valued attribute (in this case `FamilyMembers`).

As a concluding remark, we mention that tables in SQL3 correspond to classes in a pure object-oriented model. This is because SQL3 has extensions that make the language look object-based. Moreover, SQL3 is intended to become a full programming language and hence will comprise control structures as well.

5.2 OMG standards

The development of SQL as a language standard for relational databases was begun only 15 years after this model had been initially presented. As a consequence, it became difficult to push SQL as a standard for commercially available systems in the late 1980s, even though manufacturers recognized the positive implications of such a standard. For object-oriented databases the premises for developing standards are basically different. On the one hand, there are numerous ways to specify an object model for databases; on the other, the importance of object orientation is not restricted to databases. However, although standardization has begun much earlier than in the case of relational databases, it has not progressed as fast as would have been desirable.

Today, object-oriented database systems are expected to run on computers with object-oriented operating systems and to interoperate with object-oriented programming languages. Furthermore, in the not too distant future nearly all computing systems will be distributed systems based on a client–server architecture as described in Chapter 4 of this book. In view of this growing importance of object-oriented 'technology' in the near future, experts have already begun to draw up plans for standardization. In particular the Object Management Group (OMG), a huge industry consortium of more than 650 hardware and software manufacturers, is working towards establishing standards. A first important goal of the OMG was to 'define' or specify an *object request broker* (ORB), which can be understood as an interface between the hardware and software components of different manufacturers. The ORB became a central component of a specification called the Common Object Request Broker Architecture (CORBA), which describes the architecture of heterogeneous and interoperable systems at the interface and service levels. We take a closer look at these efforts in this section.

The OMG is mainly active in two areas: object orientation and distributed systems. Both fields are regarded as essential for the future development of information processing. Experts envisage that objects representing encapsulated software components will be interacting in heterogeneous hard- and software environments, exchanging and offering services, and passing messages in order to execute programs, perform calculations, access databases and so on. In order to realize this vision, two significant goals must be reached:

- *interoperability*, that is, the ability of different pieces of software to collaborate in a heterogeneous and distributed hardware environment;
- adequate *integration* of newly developed and existing legacy systems.

5.2.1 Distributed object management

At present, Distributed Object Management (DOM) is regarded as a highly promising concept to achieve the twofold goal of interoperability and integration, because it supports these features in heterogeneous, locally autonomous and distributed systems in two ways:

- The data structures and the functionality of such systems are represented as encapsulated objects, which communicate with each other by sending messages to well-defined interfaces.
- Transparent access to (server) objects is made possible for (client) applications (that is, without knowing the exact location, the internal representation or the access language used).

Distributed object management combines concepts of object-oriented and distributed system models, application integration environments and object-oriented databases, thus presenting all resources available in a network to the user as a collection of generally accessible objects which can be combined appropriately for specific applications. Ideally the following features are supported:

- the capability to integrate existing and separately developed collections of objects or data into a heterogeneous database system;
- the capability to integrate different (traditional as well as novel) types of data (for example, conventionally formatted data, audio data, images, and so on);
- the possibility of applying traditional database techniques (such as query processing, query optimization, transaction synchronization and recovery) to the integrated collection of objects;
- the capability to integrate resources at every desired level of granularity (for example, objects representing an entire DBMS as opposed to objects representing squares or employees);
- the capability to launch and control the 'execution' of combinations of objects (at any network location), if necessary by means of moving objects to a desired location;
- the capability to support collaboration between intelligent components.

Obviously, the client–server concept can be exploited in this context, in that objects as encapsulated units with well-defined interfaces can assume the role of both clients and servers (service providers). A Distributed Object Management System (DOMS) mainly comprises the following elements:

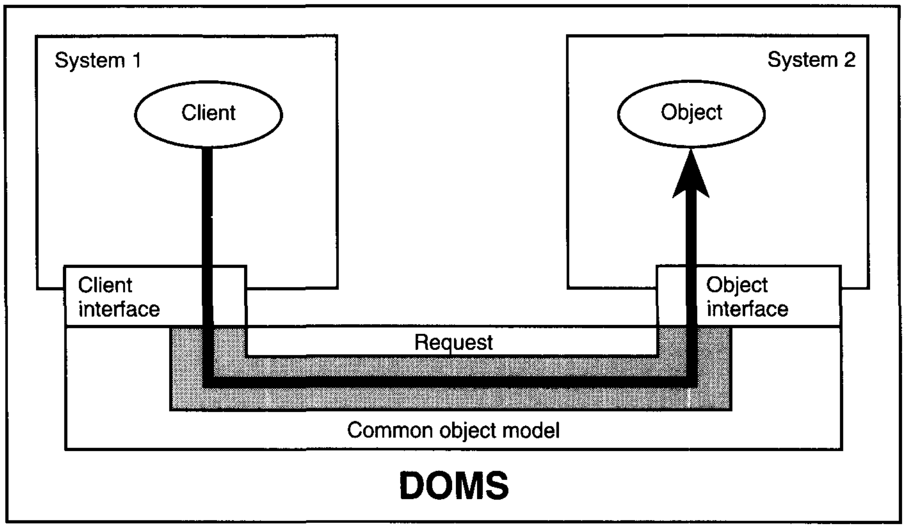


Figure 5.1 The principle of distributed object management.

- an arbitrary number of distributed nodes which run application programs, database systems or simple objects; these are the available resources;
- a collection of clients, which can also be application programs, software tools or simple objects, and which issue requests that are served by the resources.

Based on a common object model, distributed object managers mediate between clients and resources. This principle is graphically illustrated in Figure 5.1. Yet, the complexity of such a system leads experts to abandon the principle that every object knows what service can be accessed in another object or which type of functionality is to be expected. Instead, a *mediating component* (often called a *middleware* layer) is added to the ordinary client–server architecture; service providers inform this component about their available functionality. Clients can turn to this mediator if they want to learn which object in the system can supply a required service. Basically, the mediator manages a directory from which service requests can be answered directly or indirectly (that is, by turning to a second mediator). In principle, there are two types of mediators:

- We refer to a mediator as a *trader* if it is strictly restricted to its function as a mediator: that is, establishing a connection between a client and a server. This is illustrated in Figure 5.2, where, following established terminology, the client is denoted as the *importer* and the server as the *exporter* (of a service).
- We refer to a mediator as a *broker* if it passes on a service request to a server and later passes back the result to the client. When the broker which has been addressed cannot provide the requested service, other brokers may be consulted.

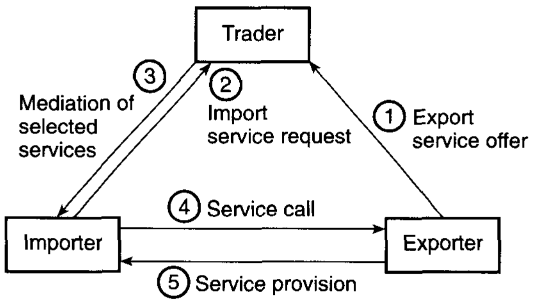


Figure 5.2 The trader principle.

OMG favours the concept of service brokering and we shall therefore consider it in greater detail in the following subsections.

5.2.2 Object Management Architecture

As outlined above, the OMG can be regarded as a consortium, which was founded with a view to standardizing DOM architectures and services. The OMG recommendations are based on an object model with the usual properties (which have been amply discussed in previous chapters). This model is used to support the integration of distributed applications. In principle, it should be possible to compose such applications in a modular fashion, with the individual modules or components calling each other via well-defined interfaces.

The general framework of the OMG activities is set by the Object Management Architecture (OMA), as summarized in Figure 5.3. This architecture defines a reference model which identifies and characterizes the components, inter-

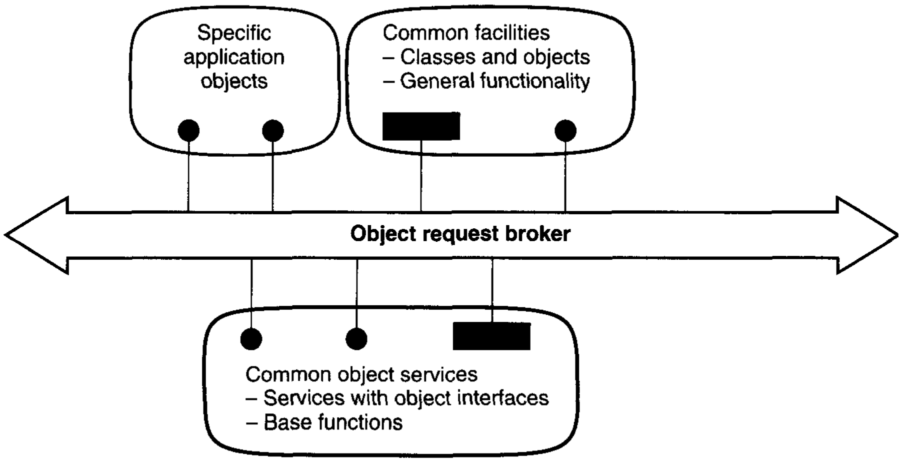


Figure 5.3 Object Management Architecture (OMA)

faces and protocols which together form a distributed object architecture. The OMA consists of four essential components:

- *Application objects*: The OMA actually aims at applications which are interoperable, portable and reusable. Therefore, application objects are specific to the individual end-user applications, and represent business objects or application programs operating on such objects.
- *Object request broker* (ORB): This is the central component which mediates between the distributed objects, passing on method calls to the appropriate target objects and returning the results back to the caller. The architecture and function of the broker, which is sometimes also called the CORBA *object bus*, are laid down in the CORBA specifications (see next subsection).
- *Common object services*: These services support communication between distributed objects, and essentially define the system-level object frameworks which extend the broker. They include basic functionality such as security, treatment of events and persistence of objects, discussed in more detail below.
- *Common facilities*: These form a collection of objects for general purposes (for example, error handling or printing) required by many applications. The common facilities are divided into horizontal and vertical facilities and can be used directly by business objects.

We now take a closer look at the CORBA object services, which are system-level services with interfaces specified in the Interface Definition Language (IDL), a language specifically designed for the description of object interfaces. These services can be used to create components, name them and introduce them to the environment; in particular, they allow the handling of objects that would normally be restricted to appearing in a database and being managed by a DBMS. At the time of writing (Spring 1997), the OMG has defined the following 13 object services:

- *Life Cycle Service*: This service defines operations for creating, copying, moving, and deleting components on the object bus.
- *Persistence Service*: This service provides a uniform interface for storing objects persistently on a variety of storage servers (such as object-oriented databases, relational database or file systems).
- *Naming Service*: This service allows objects to locate other objects by name (instead of by identity); objects can be bound to existing network directories or naming contexts (such as OSF DCE or Sun NIS).
- *Event Service*: Objects can register or unregister their interest in specific events through this service. It defines an object called an *event channel*, which collects and distributes events among objects.
- *Concurrency Control Service*: This service provides a lock manager which can obtain locks on behalf of transactions or threads.

- *Transaction Service*: Recoverable objects which use flat or nested transactions for concurrent operations are provided with a two-phase commit protocol through this service and can hence enjoy coordinated termination.
- *Relationship Service*: This service provides a way to create dynamic associations or links between objects which otherwise do not know each other. In addition, it provides mechanisms for traversing such links to group the associated objects. One application of this service is the enforcement of referential integrity.
- *Externalization Service*: This service provides a standard way of getting data into an object or out of an object using a stream-like mechanism.
- *Query Service*: This service provides query operations for objects in a similar way to object-oriented databases. In particular, it is supposed to be a superset of both SQL3 (see above) and OQL, the Object Query Language under standardization by the ODMG (see below).
- *Licensing Service*: This service provides operations for metering the use of objects to ensure fair compensation for their usage; it supports various ways of charging (per session, per node, per site and so on).
- *Properties Service*: Properties are named values which can be associated with objects or components through the operations provided by this service. In particular, properties can be associated dynamically with an object's state.
- *Time Service*: This service provides interfaces for time synchronization in a distributed object environment as well as operations for defining and managing time-triggered events.
- *Security Service*: This service provides a framework for distributed object security, including authentication, access control lists, confidentiality, or the management of credentials delegation between objects.

A corresponding illustration of the OMA specifications, which includes three more services that still need to be specified (Trader Service, Change Management Service, Collections Service), is shown in Figure 5.4.

Figure 5.5 illustrates a possible application of the Object Management Architecture in a business environment. Among the application objects are features such as invoicing and acceptance of orders or repair work, which will typically take the form of encapsulated business objects. These objects may even represent encapsulated legacy systems, which only look like objects from outside. The common facilities could include databases or information systems which are managed and accessed by company-wide data servers. The object services needed in this scenario handle, for example, database interactions (queries, transactions, and so on), various management functions, and general functionalities such as name services or registration.

In the context of OMA the broker is regarded as a connection bus for the exchange of messages which may even span several systems. Physically, this can be realized as one component or as various components cooperating with one another. The basic idea behind the ORB is mediation between the service user and the

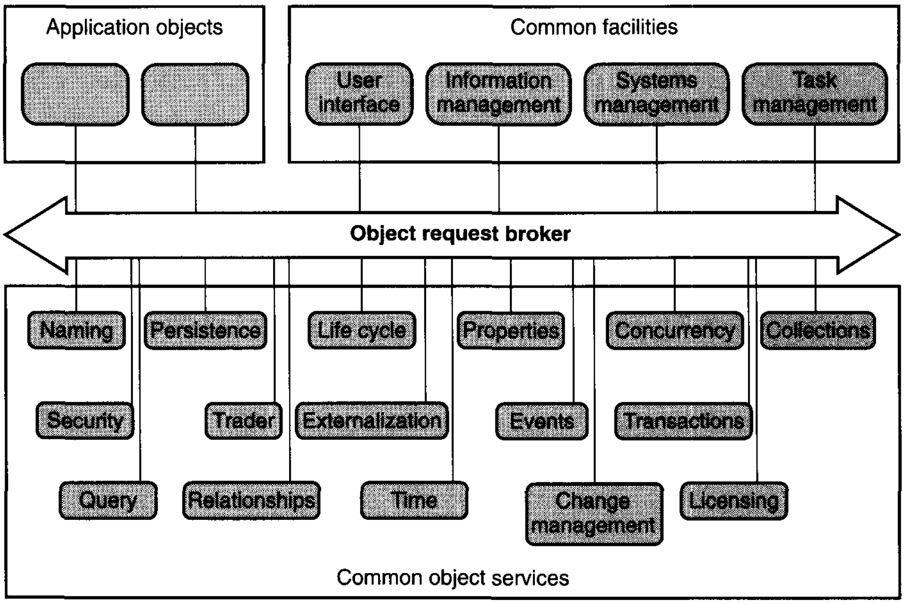


Figure 5.4 A more detailed view of the OMA.

service provider. The provider of a service informs the ORB about the type of service offered. As we have seen above, the architecture is based on client–server communication, but this time according to the model shown in Figure 5.6. A client acting as a caller of an object service communicates – assisted by a broker – with a server object which may perform the service. A request consists of an operation, a target object and (possibly) parameters as well as an optional context for the request. The task of the ORB is to find the object implementation that corresponds to the indicated target object, call it, hand over the request for processing, and return the results.

5.2.3 CORBA

The Common Object Request Broker Architecture (CORBA) acts as the central DOM architecture of the OMG, putting into concrete terms the synthesis, functionality and the interfaces of a broker. Figure 5.7 presents an overview of CORBA. The emphasis of the specification is on the interfaces which are offered by an ORB and which can be differentiated as follows:

- *call interfaces*, allowing client objects to send calls to a server object;
- *ORB interface*, allowing server and clients to access the infrastructure functions of the ORB.

Call interfaces can be divided into *static* and *dynamic* interfaces. Prior to executing the client program, static interfaces create a so-called *stub* (a fixed component of the

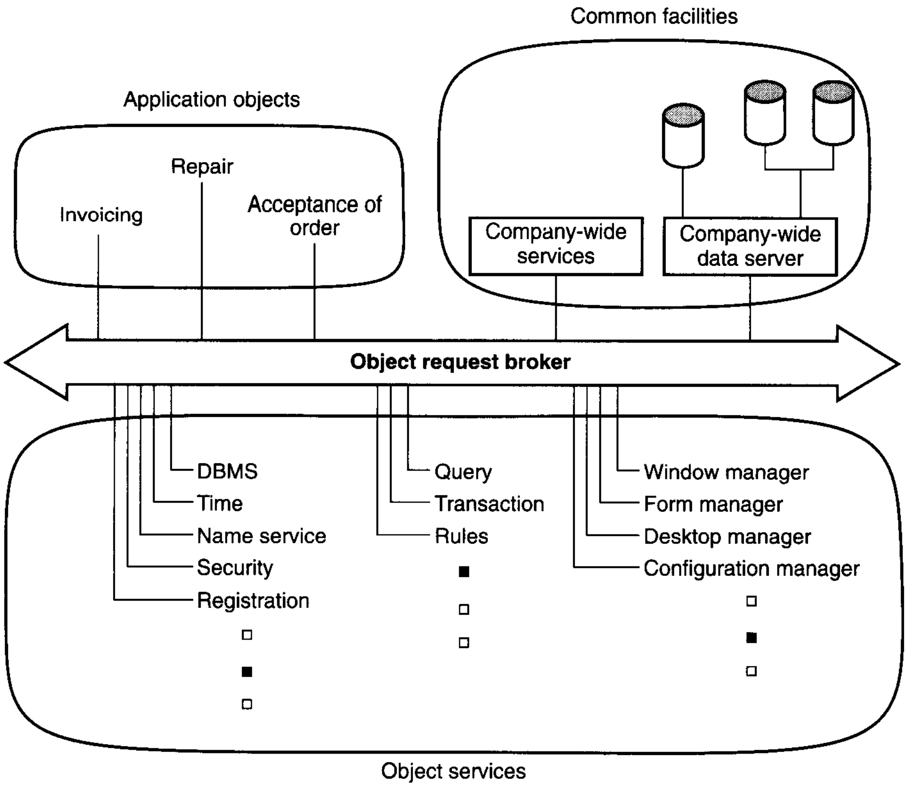


Figure 5.5 Example of the application of OMA.

program) from the interface description and statically bind it to the program. Dynamic interfaces allow the dynamic composition and invocation of calls. Both the static and dynamic client interfaces are defined in IDL (the Interface Definition Language mentioned earlier). A called object does not recognize which of the interfaces has been used to request its service. A call reaches the server object via the

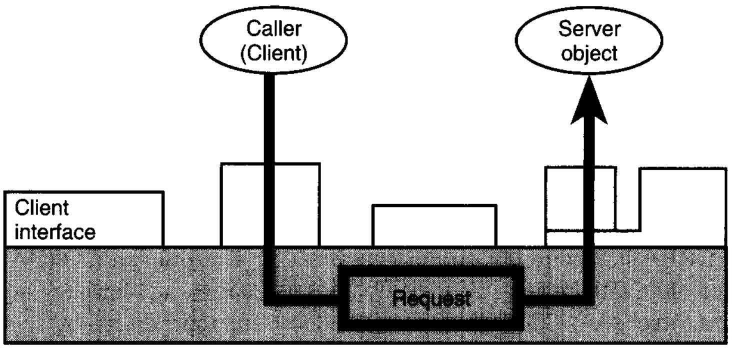


Figure 5.6 ORB mediation.

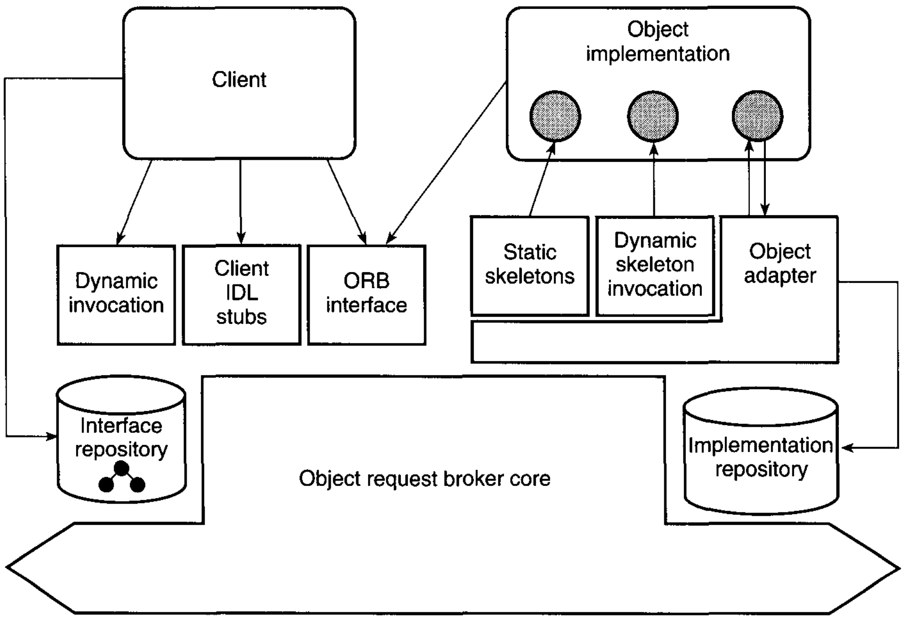


Figure 5.7 Overview of CORBA.

specific *object adapter* and the *IDL skeleton*. The caller will not know whether the desired object is placed locally or on a remote node. Again, there are static and dynamic skeletons: the static or *server IDL stubs* provide static interfaces to each service exported by the server, whereas the *dynamic skeleton interface* provides a run-time mechanism for servers that need to handle incoming method calls addressed to objects without compiled IDL stubs.

The Implementation Repository is a database containing information on implementations of server objects which can be used by an object adapter. It may include, for example, the name and location of a file storing the code to be executed for an object. Finally, the Interface Repository contains the IDL descriptions of the currently known server interfaces which may be used for the definition of new applications or for the construction of dynamic request (by clients).

The language IDL defines object interfaces which are determined independently of an object's implementation (as usual, implementations have to be provided in a corresponding programming language). The definition of an interface is made up of a number of method or operation signatures together with a (possibly empty) set of type declarations which define new types for use within the declaration. A signature of this kind consists of an operation name, a set of input parameters, a result and exceptions which can be triggered by the operation.

CORBA has currently reached version 2.0. The original CORBA 1.1 was concerned only with creating portable object applications, and left the ORB implementation to vendors. CORBA 2.0 has introduced interoperability, by specifying a so-called Internet Inter-ORB Protocol (IIOP). In essence, IIOP is the well-known TCP/IP protocol, enhanced by specific message exchanges which serve as a common backbone protocol. Platforms for cooperating objects or CORBA implementations (more precisely: CORBA-'compliant' brokers) are already commercially available; these include

ObjectBroker by Digital Equipment Corp.,
Distributed System Object Model (DSOM) by IBM,
ORBplus by Hewlett-Packard,
Orbix by Iona Technologies, and
NEO and JOE (Java Objects Everywhere) by Sun.

They all support object-oriented programming: that is, programs may be distributed over a heterogeneous network. A competing approach is given by the integration of application objects into so-called *compound documents* where a 'document' is made up of a set of objects. Two representatives of the latter category are the CORBA-compatible OpenDoc by Apple, IBM and others, which is now used by the OMG as the basis of their Compound Documents specification, and OLE (Object Linking and Embedding) by Microsoft, which is *not* CORBA compatible, but which is included in the Windows 95 and Windows NT 4 operating systems and hence comes essentially for free.

5.2.4 The OMG object model

OMG has decided against a single object model which would have to be supported by all parties intending to comply with a broker architecture, and instead opted for a twofold structure:

(1) The *core model* is, in a sense, the smallest common denominator on which the members of the consortium were able to agree. This model fulfils just the minimal requirements to justify its classification as 'object technology': identity, typing, operations, inheritance and subtyping.

(2) *Components* are compatible extensions of the core, such as attributes or relationships, which are needed in some applications, but not in others. (It should be noted that the use of the term 'component' in this context is different from that used in the context of component software, which refers to the modularization of software programs for easy or flexible configurability.)

A combination of the core model and one or more components is called a *profile*. Thus, a profile forms an object model which can be used in a specific area of application, for example in the domain of system software (object-oriented database systems, graphical user interfaces, programming languages, and so on) or concrete applications. In this sense the ODMG specifications (as described below) represent a particular profile relating to object-oriented databases and comprising components which together with the OMG model make up the ODMG model. We shall focus our attention on this model in the next section. Figure 5.8 illustrates the OMG concept of creating profiles.

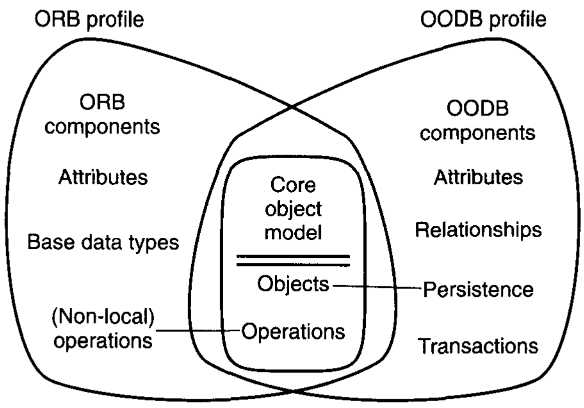


Figure 5.8 The OMG concept of profiles.

We want to conclude this section by pointing out that OMA, CORBA and the OMG object model are highly relevant to object-oriented databases, at least in the following respects:

- An object-oriented database system can be attached to a CORBA implementation so that the underlying ORB is employed to manage accesses to objects stored in the database. In this way applications get access to the database via the broker, and in fact they have access to *all* database systems connected to the broker. Using the appropriate CORBA services, it is possible to access multiple databases within a *single* application. This approach is shown in Figure 5.9 with one database system attached to the broker.

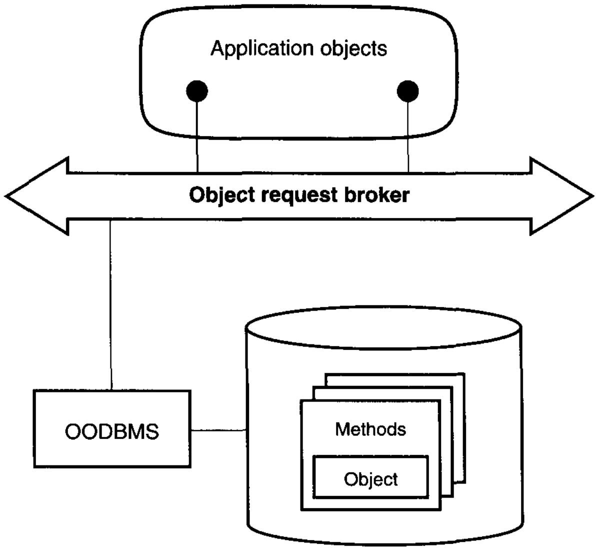


Figure 5.9 Database access via CORBA.

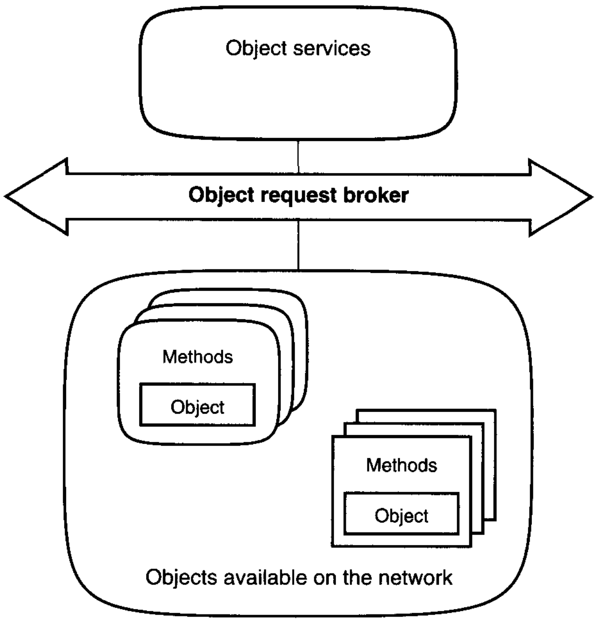


Figure 5.10 CORBA as a database system.

- The entirety of all objects within a distributed object architecture can itself be regarded as an object-oriented database, provided that relevant services with a database-like functionality are available. Since the Common Object Services of CORBA described earlier in this chapter (in paricular the Naming Service, the Transaction Service, the Relationship Service and the Query Service) provide several parts of such a functionality already, this approach is realistic. From this point of view CORBA is no longer merely a communication mechanism, but assumes the role of an internal implementation tool as illustrated in Figure 5.10.

Parts of this latter consideration have already been addressed by the OMG in more concrete terms: in a 2.0 implementation it is possible to exploit the Persistence Service of CORBA, also called the Persistent Object Service (POS), to create a heterogeneous system. Essentially, POS allows objects to persist beyond the lifetime of the application that creates the objects or beyond the clients that use them. POS allows the state of an object to be saved in a persistent store and restored from there when needed. The idea is to support a variety of storage services, including relational (SQL) database systems, object-oriented database systems, document filing systems and simple file systems. This approach is illustrated in Figure 5.11. In more detail, POS is composed of the following components: Client applications access Persistent Objects (POs), which are objects whose state is stored persistently. An object can be made persistent by having it inherit the PO class behaviour via a corresponding IDL specification. Every persistent object has a unique identifier (called

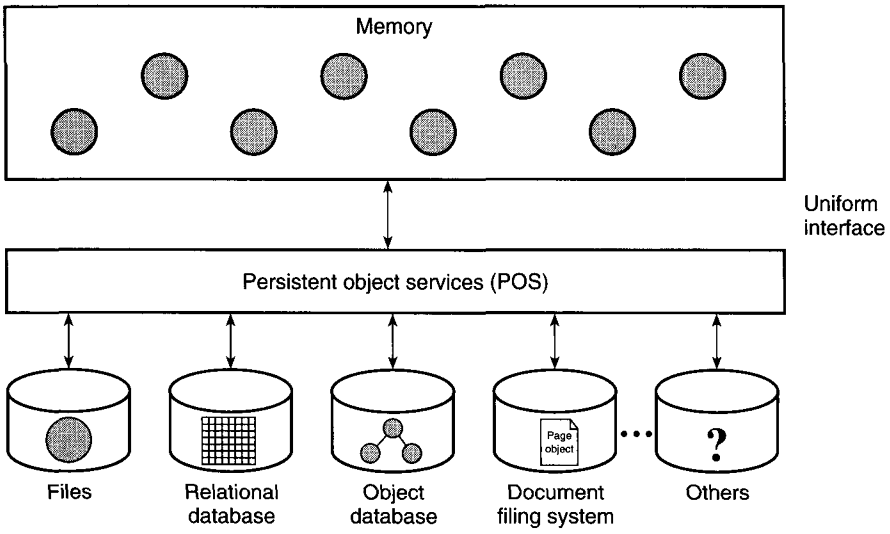


Figure 5.11 POS as a uniform storage interface.

a Persistent Identifier or PID for short) which describes the location of that object within a storage component. Next, the Persistent Object Manager (POM) is an implementation-independent interface for operations dealing with persistence. It shields the persistent objects from a particular Persistent Data Service (PDS). The latter provides an interface to a particular storage-system implementation. The POM can route PO calls to the appropriate PDS by looking at information in the given PID. The PDSs perform the task of moving data between an object and a storage system. They must implement the IDL-specified PDS interface, and may additionally support an implementation-dependent *protocol* which provides mechanisms for moving data in and out of an object. At the moment, POS specifies three such protocols for interfacing objects and stores; these are called Direct Attribute (DA), Object Database Management Group (ODMG-93) and Dynamic Data Object (DDO). Finally, datastores are the implementations that store an object's data persistently and independently of the address space containing the object. POS also provides IDL-defined interfaces which encapsulate the X/Open Call-Level Interface (CLI); these are collectively called the Datastore_CLI and provide an interface to SQL databases. The POS components are illustrated in Figure 5.12.

In conclusion, we may assert that OMA and CORBA can serve as an appropriate basis for the future development of object-oriented database systems and of DOM architectures which incorporate database or, more generally, data storage systems. As soon as full CORBA implementations are available, corresponding developments can be launched, and the resulting systems will even be able to comprise a large variety of software other than database systems. Thus future information systems are likely to look considerably different from the solutions we are used to today.

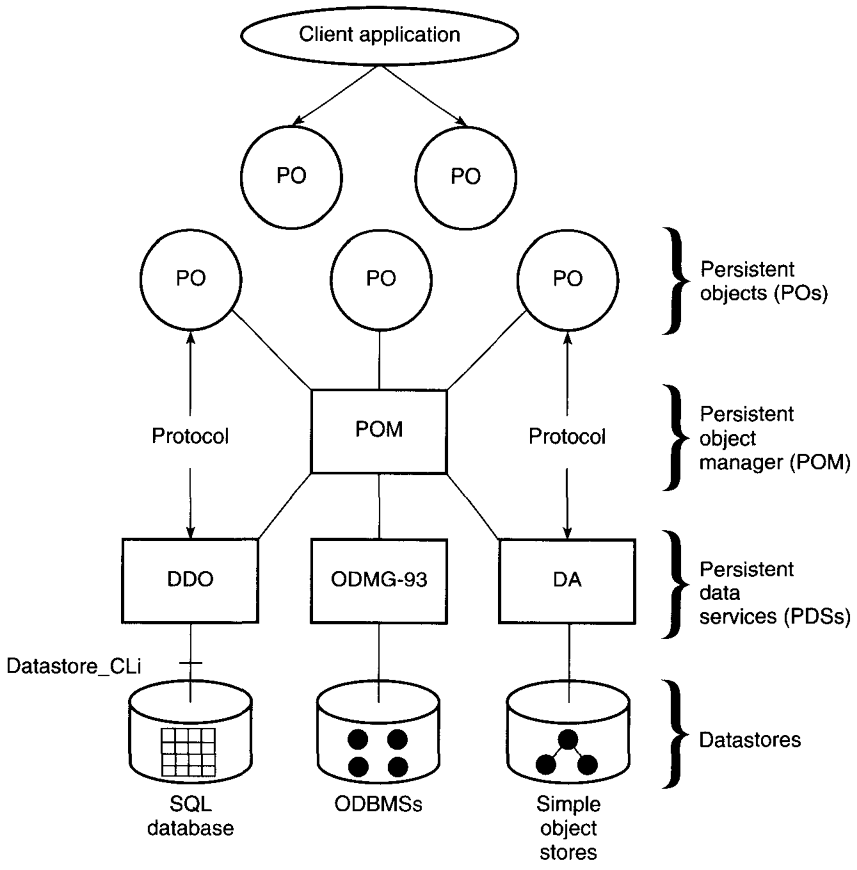


Figure 5.12 POS components.

5.3 The ODMG proposals

As already mentioned, developers and manufacturers of object-oriented database systems have joined together to form the ODMG (Object Database Management Group), a ‘subgroup’ of the OMG, and have proposed a standard for object-oriented database systems. This standard is known as ODMG-93 and roughly includes the following:

- an object model which originated from the OMG object model; in view of what has been said earlier, the ODMG model constitutes a special profile;
- an Object Definition Language (ODL) whose syntax is based on the IDL of the OMG;
- an Object Query Language (OQL), which is a declarative language oriented towards SQL, but not based on SQL3;
- bindings to object-oriented programming languages, in particular C++, Smalltalk and more recently Java.

In this section we shall briefly introduce the object model as well as fundamental aspects of the ODL and the OQL.

5.3.1 Foundations of the ODMG-93 object model

The object model according to the ODMG-93 proposal can briefly be characterized as follows:

- The *object* constitutes the central modelling construct and carries an object identity. Furthermore, an object can have one or more user-defined names.
- *Types* can be attached to objects, with all objects of one type having the same structure and the same behaviour.
- The behaviour of objects is determined by a *set of operations* which can be performed on an object of the respective type. Operations are defined by *signatures*. A signature establishes the name of an operation, the name and the type of the arguments and the return value and eventually of the exceptions.
- The state of an object is defined by the values of a *set of properties*. These properties can be either *attributes* of the object or *relationships* of the object to other objects. Attributes can have only literals as values. Relationships can only be binary and have a cardinality of $1:1$, $1:n$ or $m:n$. Each definition of a relationship can be associated with an *inverse* relationship.

The following example shows an interface definition for an object type and illustrates these basic aspects.

Example 5.7

We look at the definition of object types for persons and vehicles in our running example. It is important to know that the ODMG model does not provide object-valued attributes, but requires a relationship in order to express object references which may even need an inverse. Therefore, adapting our previous examples we obtain the following:

```
interface Person

// type properties
(       extent Persons
        key Name        )

// instance properties
{
         attribute String Name;
         attribute Integer Age;
         attribute String Domicile;
         relationship Set<Vehicle> Fleet
                 inverse Vehicle::Driven_by;
// instance operations:
.....
};
```

Every person has the usual attributes; however, Domicile is now assumed to be a simple type. The attribute Fleet is actually an object-valued attribute and references a set of objects in class Vehicle (defined next); therefore it becomes a relationship; the referenced objects need an inverse relationship. To this end, class Vehicle is defined as follows:

```
interface Vehicle

// type properties:
(       extent Vehicles
        key (Model, Manufacturer))

// instance properties:
{
        attribute String Model;
        attribute String Manufacturer;
        attribute String Colour;
        relationship Person Driven_by
                inverse Person::Fleet;

// instance operations:
.....
};
```

In this definition, we again depart from our example in not declaring the attribute Manufacturer as an object-valued attribute.

In principle a type has an interface or a signature and one (or several) implementations. A combination of signature and implementation is also called a *class*. Types are themselves objects and can have their own properties, which are referred to as type properties. Two of their three essential properties have already been used in the last example:

- *Supertypes*: Object types may be specializations or generalizations of each other and hence form IsA relationships. All attributes, relationships and operations of the supertype are inherited by the subtype. Subtypes may add further features or redefine inherited ones.
- *Extents*: The set of all instances of a type is its extent (extension). An extent may have a name (see Vehicles above).
- *Keys*: Optionally, attributes of a type may be marked as members of a key of this type.

Besides type properties, the signature of an object type can also contain *instance properties* and *instance operations*, with the operations being specified only by

- Denotable_Object
    - Object
        - Atomic_Object
            - Type
            - Exception
            - Iterator
        - Structured_Object
            - Collection <*T*>
                - Set <*T*>
                - Bag <*T*>
                - List <*T*>
                    - String
                    - Bit_String
                    - Array <*T*>
            - Structure $<e_1 : T_1 \ldots e_n : T_n>$
    - Literal
        - Atomic_Literal
            - Integer
            - Float
            - Character
            - Boolean
        - Structured_Literal
            - Immutable_Collection <*T*>
                - Immutable_Set <*T*>
                - Immutable_Bag <*T*>
                - Immutable_List <*T*>
                    - Immutable_String
                    - Immutable_Bit_String
            - Immutable_Array <*T*>
            - Enumeration
        - Immutable_Structure $<e_1 : T_1 \ldots e_n : T_n>$
            - Date
            - Time
            - Timestamp
            - Interval

Figure 5.13 Predefined ODMG type hierarchy.

means of a signature. Among the instance properties are the attributes and relationships mentioned above.

The ODMG-93 model provides a variety of predefined types, which are shown in Figure 5.13. At the root of this (only partially shown) hierarchy of object types is the type Denotable_Object, which is subdivided into *mutable* and *immutable*. In principle *objects* are mutable, whereas *literals* are immutable; in other words, objects can change their values, but literals (which *are* values) can be replaced only by other literals. In particular, the values of the attributes of an object or of the relationships in which objects participate can be changed; clearly, the identifier associated with an object cannot be modified. Both subdivisions are further divided into 'atomic' and 'structured'.

Note that all instances of type Denotable_Object have an identity; the identity of a literal, however, is identical to its value. Objects do not only have an identity, but may also have one (or several) user-defined names. Moreover, it is important to know that the type Object is equipped with the following predefined properties:

```
has_name?: Boolean;
names: Set<String>;
type: Type;
```

as well as with the following predefined operations

```
delete();
same_as?(oid: Object_id) → b: Boolean;
```

Objects are created with the help of the create operation, which allocates storage space for the newly created object and assigns an identifier.

Among the literals are the usual base types ('atomic literals') such as Integer or Float. Structured objects and structured literals are created using constructors such as Set or List. The structure of an object of type Structure or of a literal of type Immutable_Structure is a tuple whose components can be assembled in any fashion. A literal of a particular kind is Enumeration, which generates an enumeration type.

As far as queries and working with a database in this model are concerned, the central construct is the *collection*, which is an object comprising other objects (of the same type). Queries use collections for *iteration* purposes: that is, the objects of a collection can be accessed one after the other. Iteration is achieved by an object of type Iterator, which in particular maintains a 'current position' while passing through a collection. All the subtypes of the type Collection <*T*> shown in Figure 5.13 have various predefined operations (for example, set operations for objects of the type Set <*T*>) which are partly inherited from the supertype and then overridden.

By way of conclusion we should like to add that the object model which emerges from the ODMG-93 proposal defines a number of other details, such as the possibility of fixing the *lifetime* of a (mutable) object or the notion of a *transaction*.

5.3.2 The Object Definition Language (ODL)

In this section we look at the Object Definition Language (ODL) of the ODMG-93 proposal, but we shall refrain from presenting its entire syntax. ODL is a specification language with the aim of defining interfaces or signatures for object types which conform to the ODMG-93 model. In this respect ODL is intended to support the portability of database schemas; it is – as stated earlier – based on the Interface Definition Language (IDL) of OMG.

In the example given in the previous subsection some important language elements have already been pointed out. A type is defined by the specification of a signature, generally in accordance with the following syntax:

```
interface_dcl ::= INTERFACE identifier [ inheritance_spec ]
                   type_property_list
                   [ : persistence_dcl ]
                   { [ interface_body ] } ;
persistence_dcl ::= PERSISTENT ( TRANSIENT
```

The characteristics of a type (supertype information, extent naming and key specification) are defined according to the following syntax:

```
inheritance_spec     ::= scoped_name_list
type_property_list   ::= ([ extent_spec ] [ key_spec ])
extent-spec          ::= EXTENT identifier
key_spec             ::= KEY[S] key-list
key                  ::= property_name_list
property_name        ::= scoped_name
scoped_name          ::= identifier
                         | :: identifier
                         | scoped_name :: identifier
```

Essentially, an interface_body consists of the declaration of attributes, relationships and operations; their principal syntax elements are as follows:

```
interface_body ::= export_list
export         ::= attr_dcl | rel_dcl | op_dcl

attr_dcl       ::= [ READONLY ] ATTRIBUTE
                   domain_type identifier
domain_type    ::= simple_type_spec
                   | struct_type
                   | enum_type
                   | attr_coll_spec literal
                   | attr_coll_spec identifier
attr_coll_spec ::= SET | LIST | BAG | ARRAY

rel_dcl        ::= RELATIONSHIP
                   target_of_path identifier
                   [ INVERSE inverse_path ]
                   [ { ORDER_BY attr_list } ]
target_of_path ::= identifier
                   | rel_coll_type identifier
inverse_path   ::= identifier :: identifier
attr_list      ::= scoped_name_list

op_dcl         ::= [ ONEWAY ] op_type_spec identifier
                   ( [ param_dcl_list ] ) [ raises-expr ]
op_type_spec   ::= simple_type_spec | VOID
param_dcl      ::= param_attr simple_type_spec declarator
param_attr     ::= IN | OUT | INOUT
raises_expr    ::= RAISES ( scoped_name_list )
```

We shall not describe the complete syntax for the specification of operations, but instead conclude this section with an example, which in fact is a continuation of Example 5.7.

Example 5.8

Again we turn to the automobile distributor for which we defined the object types `Person` and `Vehicle` in the last example. We shall supply the remaining definitions and again restrict ourselves to structural aspects:

```
interface Employee: Person
    ( extent EmplSet )
    {
      attribute set<String> Qualifications;
      attribute Float Salary;
      relationship Subsidiary is_Mgr_of
             inverse Subsidiary::Manager;
      relationship Subsidiary is_Empl_of
             inverse Subsidiary::Employees;
      relationship Company is_Pres_of
             inverse Company::President;
    };
```

On the one hand, the type `Employee` (here defined without family members) is a subtype of the above-defined type `Person`; on the other, it has three different relationships with two other types, namely `Subsidiary` and `Company`. The remaining type definitions are as follows:

```
interface Company
    ( extent Companies )
    {
      attribute String Name;
      attribute String Headoffice;
      relationship Employee President
             inverse Employee::is_Pres_of;
      relationship Set<Subsidiary> Subsidiaries
             inverse Subsidiary::is_Subs_of;

      hirePres (in Person);
      firePres (in Employee) raises (not_existing)
    };
```

For `Company` the signatures of two operations, `hirePres` and `firePres`, are indicated; the latter also includes an exception condition.

```
interface Subsidiary
   ( extent SubsSet )
   }
     attribute String Name;
     attribute String Office;
     relationship Employee Manager
            inverse Employee::is_Mgr_of;
     relationship Set<Employee> Employees
```

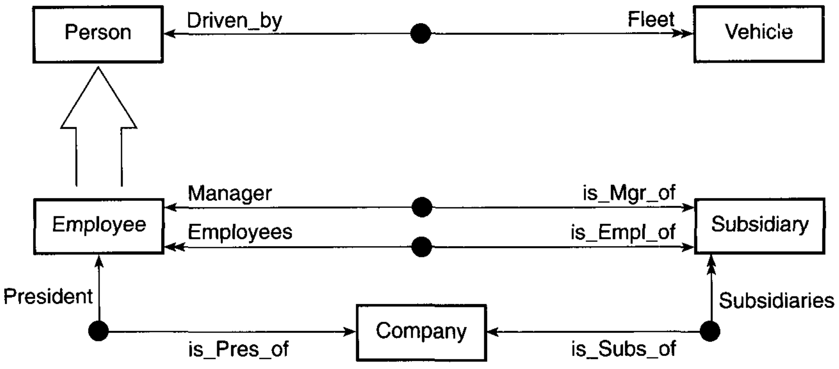


Figure 5.14 Running example written in the ODMG ODL (extract).

```
            inverse Employee::is_Empl_of;
      relationship Company is_Subs_of
            inverse Company::Subsidiaries;
   };
```

The types defined in the last example and in Example 5.7, and their relationships, are graphically summarized in Figure 5.14.

5.3.3 The Object Query Language (OQL)

We now want to turn our attention to the Object Query Language (OQL) proposed by the ODMG. This language is based on the following principles and assumptions:

- It uses the ODMG-93 object model described earlier as a foundation.
- OQL is a declarative and optimizable language, but not Turing-complete. Therefore, OQL is a 'classical' database query language, and in contrast to SQL3, it is not designed as a full programming language.
- The syntax of OQL is similar to that of SQL, which means the basic query syntax is the Select-From-Where construct. The syntax of OQL is still subject to refinements and enhancements; in particular, the integration into programming languages (including Smalltalk, C++, and Java) needs to be finalized.
- Unlike SQL, OQL does not favour the set as its primary query medium, but treats tuple structures or lists in the same way as sets.
- OQL has no explicit update commands or operations, but appropriate methods are supplied for the purpose of updating (that is, inserting, deleting or modifying) objects.

With respect to the third point it should be noted that the syntax of OQL is neither stable nor has it been implemented yet. In its current state (in early 1997) Version

1.2 of OQL is closely related to and based on O_2SQL, the query language of O_2 we discussed in the previous chapter. We shall thus refrain from describing the syntax of OQL, and restrict ourselves to giving a few examples.

OQL is a strictly typed language permitting queries which deliver atomic or structured objects or literals as answers. Among its design goals were orthogonality, that is, every mechanism of the language can be applied to every construct, and arbitrary nesting of queries; in other words, OQL is a functional language.

Example 5.9

The following query yields the age of persons with the name 'John Smith':

```
select distinct x.Age
from Persons x
where x.Name = "John Smith"
```

Note that this query addresses the extension `Persons` of type `Person` which was defined in Example 5.7. The result of this query is a literal of type `set<Integer>`.

Example 5.10

The next query yields the names of all employees who work in a subsidiary branch of a company whose president earns more than 100 000; it also supplies the name of that company:

```
select distinct struct(EName: x.Name, CName: y.Name)
from EmplSet x, Companies y
where x in (select z.Employees
            from SubsSet z
            where z in y.Subsidiaries)
       and y.President.Salary > 100000
```

Here the result is supplied as a tuple structure with two attributes.

OQL does not need Select-expressions for every query. If, for example, HenryFord is the name of an object of type Person, then

```
HenryFord
```

constitutes a valid query. Furthermore, the properties of this object can be directly accessed; as, for instance, in the expression

```
HenryFord.Age
```

Thus, queries may supply answers in the form of individual objects, sets of objects, individual literals or sets of literals.

Example 5.11

Following the relational SQL, the current version 1.2 of OQL provides capabilities for querying relational structures. A *selection* which just returns object identities would be written as

```
select x
from Persons x
where x.Name = "John Smith"
```

A *projection* of `Vehicle` objects onto the attributes `Model` and `Manufacturer` is expressed as follows:

```
select Model, Manufacturer
from Vehicle
```

We should like to conclude this section by emphasizing once again that our description of the activities of the ODMG reflects the current state of affairs, since the task of this group is not yet completed. The introduction of the standard in commercial products was scheduled for early 1995. However, it had to be postponed several times, and now it remains to be seen when it will actually be put into practice. Yet, the development of the ODMG standard is of great importance to the field of object-oriented databases.

Finally, we should also mention that efforts are under way to create a merger between SQL3 and ODMG's OQL. Since OQL is already based on the Select-From-Where construct, such a merger appears reasonable, in order to widen the applicability of both languages. Clearly, a number of details have to be taken care of, such as matching the strong typing of OQL in SQL, or extending SQL queries from tables to collections of objects and allowing queries to return such collections.

5.4 Bibliographical notes

General introductions to standardization activities as described in this chapter can be found in Kim (1995) or Simon (1995); in particular, we should like to refer the reader to Manola (1994) and Moss (1994). Specifications of CORBA and OMA can be found in the documents of the Object Management Group (1991, 1992, 1995); in-depth descriptions are also given by Orfali et al. (1996a) or by Siegel (1996). A detailed discussion of the ODMG-93 model is provided by Cattell (1994, 1996); for a short overview, the reader may also consult Bancilhon and Ferran (1995). Orfali et al. (1996b) give an introduction to client–server systems in general and to CORBA in particular. A detailed introduction to the Persistent Object Service of CORBA is given by Sessions (1996). A critical evaluation of CORBA can be found in Kleindienst et al. (1996) or in Wallace and Wallnau (1996). Vossen (1997) surveys CORBA and points to several projects that are based on it or use it. A typical application of CORBA can be expected in the area of multi-database systems; for an introduction to this subject, see Bukhres and Elmagarmid (1996). We finally mention that Figure 5.5 is an adaptation of a figure in Brodie and Stonebraker's book (1995).

We emphasize again that for activities like standardization, in which multiple parties sometimes all over the world are involved, it is becoming more and more common to exchange information through the World Wide Web, and to make documents and other information available on the Web. We therefore encourage the reader to monitor the corresponding Web pages for up-to-date information on the topics discussed in this chapter. The relevant locations include:

For SQL3	http://www.jcc.com/sql_stnd.html
For the OMG	http://www.omg.org
For the ODMG	http://www.odmg.org

Part III

Theoretical Concepts

6 Algebraic operations on databases

In this chapter, we offer some fundamental considerations about algebraic operations on object-oriented databases. In relational databases, algebraic operations have proved themselves not only as the basis for the semantics of query languages, but also as the basis for the optimization of queries. Moreover, algebraic languages are equivalent, for example, to calculus-based ones; they can thus be considered both as a 'robust' concept and as a measure of the expressive power of a language. The question arises as to whether similarly interesting properties of algebraic operations or languages occur in object-oriented databases; this chapter briefly introduces this issue. It is our objective to show the main analogies between classical relational algebra and so-called *object algebras* as well as to demonstrate the new problems to be solved when defining algebraic operations on complex structures. We do not discuss specific object algebras, but refer the reader to the references given at the end of the chapter.

6.1 Algebraic operations on relations

First, we compile several important aspects of the classical database algebra, the *relational algebra*, and then investigate these with regard to object orientation.

6.1.1 Relational algebra and its properties

We first remind the reader of the explanations in Section 1.3.2, where we briefly discussed the connection between SQL and relational algebra. As was demonstrated there, the fundamental operations of this algebra, that is, projection (π), selection (σ), the set operations union ($\cup$) and difference ($-$) and the natural join ($\bowtie$) can all be expressed in SQL. By means of these operations (and possibly the option of *renaming* attributes) numerous others can be defined; however, no fundamentally new possibilities arise to express queries to a relational database.

Relational algebra has several important properties which make it suitable as the basis of a semantics for relational query languages:

- Relational algebra is *complete*, which means that (given) relations are always transformed into (most frequently different) relations by means of the algebra's expressions. The result of the evaluation of an individual operation or of an expression can thus serve as input for the evaluation of another operation or a further expression.

  This observation can be formalized by perceiving queries to a relational database as *mappings* of (given) relations to (derived or newly constructed) relations; the queries which can be expressed in relational algebra are particular mappings enjoying specific properties.

- Relational algebra is *safe* in the sense that each result delivered by one of its expressions is *finite*: that is, contains a finite number of tuples.

- The language of relational algebra offers a tool for the definition of *external views*. In other words, the external layer according to the ANSI/SPARC three-layer architecture model can essentially be defined in terms of algebraic expressions which describe derived relations.

- All expressions of relational algebra can be *efficiently* evaluated, because the underlying operations all have *polynomial* time complexity. More precisely, the time needed to evaluate one of the operations is polynomial in the 'size' of the respective input, independent of how this input is stored.

  It can even be shown that the task of evaluating expressions of relational algebra is of *low* polynomial time complexity, which has both a positive and a negative consequence. On the one hand, an algebraic expression offers possibilities for optimization (see the following subsection) and in this context even possibilities of *parallel* evaluation. On the other hand, the *expressive power* of relational algebra is limited, which is generally considered the price to be paid for high efficiency.

The last-mentioned aspect is of fundamental importance for database languages. If it is to be *guaranteed* that each query which is expressible in a given language can be efficiently evaluated, limitations have to be imposed on the expressive power of that language. In relational algebra there is, for example, no recursion and no iteration, both constructs with which inefficient or even non-terminating programs can be written in high-level programming languages.

In the course of this section we will discuss two aspects of the algebra in more detail: the possibility of optimizing queries and the role of the algebra as a 'reference point' for other relational languages.

6.1.2 Algebraic optimization

One of the particularly important aspects of relational algebra is that a series of manipulation rules applies to its operations; these rules state, for example, when two operations *commute* or whether an operation is *associative*. An important application of such rules is the provision of methods to *optimize* queries: that is, to transform a given query into an 'equivalent' one which supplies the same result (if applied to the same underlying database), but which can be evaluated more efficiently (relative to a given cost measure such as the number of block accesses on disk). A few simple examples will illustrate this point:

- If a projection and a selection are to be applied to a given relation, these operations can be exchanged (commuted) if the selection attributes are amongst those onto which the projection is made.
- If a selection is applied to a join expression and if the selection attributes only occur in one of the join operands, then the selection can be applied to that operand before the join is computed; that is, it is distributed into the join expression.

The significance of such manipulation rules with respect to query optimization becomes clear if you visualize the role an algebra usually assumes in database systems: in real systems, algebraic operations are not used at the user interface, but internally in the system for implementation and especially for optimization, as indicated in Figure 6.1. There are several reasons why this role of an algebraic language is justified:

- Syntactically, relational algebra, when compared with SQL, can be characterized as a kind of 'assembly language', because a user can work only with a fixed set of basic operations in this algebra. Therefore, a system developer would only want to make it directly accessible to the ordinary database user in exceptional cases.
- From an internal point of view, a 'logical' algebra like the one discussed here is typically complemented with a 'physical' algebra in which operators are implemented as, and are directly executable as, system processes.

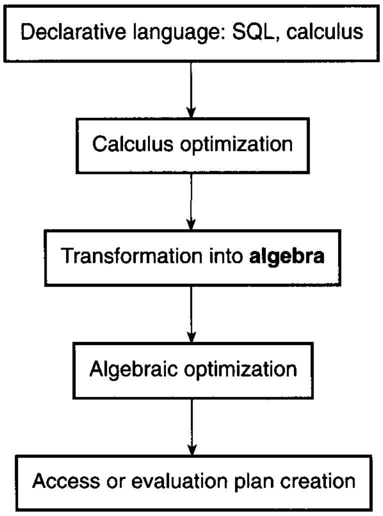

Declarative language: SQL, calculus
Calculus optimization
Transformation into algebra
Algebraic optimization
Access or evaluation plan creation

Figure 6.1 The role of algebraic languages.

Reasons like these justify attempts to imitate the algebraic language approach in other data models as well, especially in object-oriented ones. Problems that might arise from this will be discussed below.

6.1.3 Relational algebra as a language yardstick

Apart from relational algebra, the relational data model knows the *tuple calculus* as well as the *domain calculus* as declarative counterparts. These calculi are based on the perception that a relational schema R with n attributes (in databases) and a relation name R of arity n (in mathematical logic) are strongly related concepts. A set $\{R_1, ..., R_k\}$ of relational schemata of a database schema can thus be regarded as a vocabulary of relation names. Additionally, *variables* are needed for a logical language. If these variables represent tuples from relations, they are called *tuple variables*; if they represent individual domain elements, they are called *domain variables*. Formulas in the tuple calculus (with tuple variables) or in the domain calculus (with domain variables) are equipped with a semantics by interpreting them appropriately as formulas over a given database.

In principle, both calculi are *not* safe in the sense of the previous subsection and thus more powerful than the algebra, because they allow queries with infinite results to be posed. This additional expressive power is, however, largely insignificant for practical applications, so that the calculi are usually restricted (to 'safe formulas') in such a way that infinite results are no longer possible. It can then be shown that the safe calculi are both equivalent to relational algebra: that is, for each expression in one of the calculi there is an expression in the algebra yielding the same result if applied to the same database and vice versa. This means that the two approaches to language design (the algebraic one and the calculus one) are equivalent. One important consequence of this is a notion of *completeness* for relational query languages, the so-called *Codd-completeness*: a relational (query) language is called Codd-complete if its expressive power equals that of the relational algebra.

On closer examination the notion of Codd-completeness does not seem very far-reaching even in ordinary relational databases, since it can be shown that important queries (such as the transitive closure of a binary relation) are not expressible in the algebra nor in either of the calculi. Therefore, other forms of characterizing the expressive power of relational query languages have been developed in the past (see the bibliographic notes for references); these adapt concepts of theoretical computer science to relational databases in an appropriate manner.

The concept of a *computable* query is of particular significance in this context: in analogy to a computable function, a computable query is basically a partially recursive function, which (i) yields as output, for a database given as input, a relation over the database domain, and (ii) satisfies a *consistency criterion* which essentially requires that the query result is independent of the internal representation of the database. This criterion, also called *C-genericity*, captures the intuition that a query only 'uses' information that is available at the conceptual level of the database in question. In particular, distinct values can be treated differently only if they are distinguishable by means of conceptual information or appear explicitly in the query. More formally, this can be expressed as follows:

If C is a finite set of values from a universe U, and if q is a query which takes as input a set d of relations and maps this to an output relation, q is called *C-generic* if for each permutation ρ on U such that $\rho(x) = x$ for each $x \in C$ the condition

$$\rho(q(d)) = q(\rho(d))$$

holds; that is, the following diagram commutes:

$$\begin{tikzcd} d \arrow[r] \arrow[d, "\rho"'] & q(d) \arrow[d, "\rho"] \\ \rho(d) \arrow[r] & q(\rho(d)) = \rho(q(d)) \end{tikzcd}$$

If $C = \emptyset$, then q is called *generic*. Genericity thus requires that a query is not sensitive to a renaming of the constants (by means of permutation ρ) occurring in the database. Such a set C specifies 'special' constants, which are explicitly mentioned in the query, especially in selection conditions. However, each C-generic query can be understood as a generic query by incorporating the constants from C into the input. Thus studying generic queries (instead of C-generic ones) suffices.

As a result of these considerations, a (database) *query* is defined as a computable and generic (database) mapping. A language is called *complete* if it can express all queries. It is then easily seen that all queries expressible in relational algebra are also queries in this sense (that is, computable and generic); they are even C-generic, with C being the *active* domain of the given database. On the other hand, the algebra queries form a strict subset of the set of all queries, because, for example, the transitive closure is also a (computable) query but, as already mentioned, not expressible in relational algebra. The same statements are also true for calculus queries, which means that neither the algebra nor the calculi are complete.

Numerous proposals have been put forward for the design of a language in which all computable and generic queries can be expressed, the two most important of which we shall mention here. The first generalizes relational algebra into an imperative programming language by introducing variables which take relations as values, assignment statements, and a while-loop as a control structure. The second allows an iteration of a program or of part of a program as long as a condition (stated in first-order logic) is satisfied. Queries in the second type of proposal are constructed by using a calculus together with a fixpoint operator which binds a predicate name R occurring in the formula in question freely and positively only (that is, under an even number of negations). That formula is iterated until a fixpoint is reached and hence the result does not change anymore. Both approaches are important milestones towards complete languages; for further details refer to the literature given in Section 6.5.

6.2 Algebraic operations on nested relations

We now expand our study to include objects which are structured in a particular way but do not yet have an identity, the so-called *nested* relations. They are different from the previously studied flat relations in that individual attributes can have entire relations as values (compare Section 1.2).

Example 6.1

In our running example (see Figure 1.8), information on automobiles could be represented without object identities in a nested relation as follows:

Automobile	*Model*	...	*Car body*	*Drive*		
				Gearing	*Engine*	
					HP	*CC*
	Sierra		5 door	MT75	120	1998
	Mondeo		4 door	MT75	115	1795
		⋮			⋮	

In this case, the attribute *Drive* is even nested twice; however, the respective 'subrelations' are single-valued only. Nesting is often more appropriate if multiple values are associated with attributes, as for example with companies:

Company	*Name*	*Head office*	*Subsidiaries*		...
			Location	*Street*	
	Ford	Cologne	Cologne	Fordstrasse	
			Ghent	Brusselweg	
			Saarlouis	Autoallee	
	VW	Wolfsburg	Kassel	Hauptstrasse	
			Pamplona	Av. Carlos	

Algebraic operations can also be defined for the nested relational model; these follow in part as a direct generalization of the operations known from flat relations. We mention selection and projection as examples which generally work as on normal relations, but with the following differences:

- A selection operation may now, due to attributes having a relation as value and thus a set value, contain other forms of selection conditions, for example the creation of subsets.
- A projection can be applied to complex attributes (with a relation as value) as well as to flat ones; but generally, they cannot be directly applied to a component of a complex attribute.

Example 6.2

We again examine the two relations of the previous example and the following operations:

(1) A *selection* in relation *Company* according to the condition

"{ Cologne, Ghent } $\subseteq$ Subsidiaries.Location"

yields:

Company	*Name*	*Head office*	*Subsidiaries*		...
			Location	*Street*	
	Ford	Cologne	Cologne	Fordstrasse	
			Ghent	Brusselweg	
			Saarlouis	Autoallee	

(2) A *projection* of *Automobile* onto the flat attribute *Model* as well as the complex attribute *Drive* yields:

Automobile	*Model*	*Drive*		
		Gearing	*Engine*	
			HP	*CC*
	Sierra	MT75	120	1998
	Mondeo	MT75	115	1795
			⋮	

While the algebraic operations, which can be defined as generalizations of those of the ordinary relational model, work in the well-known fashion, which means they deliver result relations whose schemas have not been manipulated according to a possibly given hierarchical structure, there are new operations for nested relations which can transform complex attributes into flat ones and vice versa. These operations are called *nesting* and *unnesting*, respectively; their effect is demonstrated here with an example.

Example 6.3

An unnesting can be applied to relation *Company* of Example 6.1 regarding the attribute *Subsidiaries*; by unnesting, this attribute is replaced by its components and thus further tuples are created, whose values for the remaining attributes are replicated from the given values. In our example we obtain the following result:

Company	*Name*	*Head office*	*Location*	*Street*	...
	Ford	Cologne	Cologne	Fordstrasse	
	Ford	Cologne	Ghent	Brusselweg	
	Ford	Cologne	Saarlouis	Autoallee	
	VW	Wolfsburg	Kassel	Hauptstrasse	
	VW	Wolfsburg	Pamplona	Av. Carlos	

In this example nesting allows the unnesting which has just been demonstrated to be undone; for the subrelation to be created, an attribute name like Subsidiaries would have to be reintroduced.

Compared with the flat relational model, the nested relational model thus uses new operations, which have an effect on existing complex structures. It is obvious that such operations can be defined even if, apart from the tuple and set constructors, further constructors are admitted and their strictly alternating application is given up. In this way, algebraic operations become relevant or applicable to complexly structured objects, so that an attempt can be made to realize the attractive properties of an algebraic language approach known from the relational model even in such a context.

Concluding this section, we should like to add that, in the absence of object identity, it is indeed possible to generalize many algebra properties known from the relational model to complex structures. However, certain complications may occur. For example, in the nested relational model the operations of nesting and unnesting are generally *not* inverse to each other. While nesting can always be undone with a subsequent unnesting, the converse applies only under certain additional conditions.

6.3 Algebraic operations on object bases

Let us now return to object-oriented databases. According to the extensive preliminary considerations above, one might expect that a generalization of algebraic operations to object-oriented databases could be done in a similar way. However, it will turn out that such a venture creates substantial new problems.

6.3.1 Introductory considerations

It is apparent that an algebraic language for object bases must provide operations which are applicable to complex values and adequately support object identity. Operations on complex values or on sets of such values can result, as discussed above, from a generalization of the operations on nested relations. For operations on (sets of) objects at least the following distinctions must be made:

- Methods written for particular objects or classes, that is, the dedicated behaviour with which objects and classes are equipped, are *object-specific* or *explicit* operations. It should be noted that in some systems (for example, in GemStone) only this type of operation can be found. The fact that optimizability of such operations is not given and that type safety is difficult to guarantee could then prove problematic.
- Query operations that – in the style of relational algebra – are considered to be part of the data model and do not require an explicit definition relative to a particular class are *implicit* operations.

Following this classification, we shall look only at implicit operations here – we shall even disregard update operations. Implicit algebraic operations can, in principle, either manipulate existing class instances, that is, sets of objects, or deduce new object sets from existing ones. In this context, an answer to the question of what kind of result a query should deliver has significant influence. Basically there are two possibilities:

- A query always delivers a set of values, which means that values are always extracted from the objects accessed, but object identities are disregarded.
- A query delivers (always, or depending on the formulation) a set of objects.

It should be clear that the first option represents a severe limitation and, for example – according to our explanations in Section 2.1 – violates the criterion of closure:

when an operation is applied to an object set and supplies only values, another operation of the same kind cannot be directly applied. Analogous arguments show that the formation of views is impossible. Moreover, by proceeding in this way, object identities are lost; thus, objects cannot be compared with regard to their identities, and their associated methods are no longer applicable.

Therefore, it seems advisable that algebraic operations should be introduced in such a way that they are able to handle object sets and, in particular, to yield object sets as results. We shall examine this in more detail, first under the assumption that algebraic operations preserve the identities of existing objects; they are then also referred to as *object-preserving* operations. They are confronted with the new task of classifying or typifying a query result; this will be discussed in Section 6.3.2 by means of examples. One solution to this classification problem is to make the operations *object-creating*; this will be discussed in Section 6.3.3. Note that here we take up ideas again which were previously discussed in connection with Example 2.19.

6.3.2 Object-preserving operations

Object-preserving operations, as their name suggests, maintain the identity of the objects processed by the operations. If the result of such an operation is to be placed in a given class or type hierarchy, for example in order to apply existing methods to the obtained objects, certain difficulties arise, as shown in the following examples.

Example 6.4

Consider the following instance of the class *Automobile* (compare Figure 1.9):

Automobile

	Drive	*Car body*
#10	#20	Sedan
#11	#21	Hatchback
#12	#22	Sedan

A *selection* of all automobile objects that have a sedan car body results in the following object set:

	Drive	*Car body*
#10	#20	Sedan
#12	#22	Sedan

Apparently, the type has not changed with regard to the operand *Automobile*; but a subset of the original object set has been generated and consequently a subclass of the class *Automobile*.

The situation described in the last example, namely that an (object-preserving) selection delivers a subclass of the given operand with the same type, may lead to problems if the operand already has subclasses. If Automobile had, for example, the subclass Fourdoor, we should have to clarify the way the above selection result is related to this class. Whether the two subclasses are disjoint or incomparable with respect to set inclusion depends on their current state and cannot be determined a priori.

Example 6.5

Again consider the instance of class *Automobile* given in the last example. Now we are interested in a *projection* onto the attribute *Drive*. The result is as follows:

	Drive
#10	#20
#11	#21
#12	#22

The set of objects has not changed in this case, but the type has: the type of the result is a supertype of the type of the operand, because the operand type has (as a tuple type) more attributes!

In analogy to the question raised above for selections it might, in the case of a projection, be appropriate to clarify how the type of a projection result is related to other supertypes of the operand type: in our running example, for instance, to the type of class Vehicle.

Considerations analogous to those for selection apply to the set operations *difference* and *intersection*:

Example 6.6

It is reasonable that an (object-preserving) difference operation is applied only to object sets that have a non-empty intersection, thus for example to classes which are subclasses of each other. In our running example the expression

Person – Employee

yields all persons (more precisely: all person objects) that are not employees. This means that a subclass of persons with the same type as class *Person* is formed.

Whereas in the previous examples either the type of the result was different from that of the operand or a subclass was formed, both aspects change with a *union*, as is shown in the following example.

Example 6.7

Consider the modification of our running example shown in Figure 2.1, especially the subclasses *Shareholder* and *Employee* of the class *Person.* A union of the form

$$\text{Shareholder} \cup \text{Employee}$$

yields a heterogeneous object set whose type is a subtype of both *Shareholder* and *Employee* and which is a superset of both operands.

Concluding this section, we should like to remark that these considerations can also be applied to join-like operations, which in general lead to type extensions and consequently to the formation of subtypes.

Example 6.8

If a join of classes Person and Address is formed in our example via the attribute Domicile, this can be regarded as an unnesting which, for each object of class Person, replaces the identity of an address, which is stored as the value of attribute Domicile, by a value tuple with the associated values for Street and Location.

In principle, the problems of classification and typification in object-preserving operations that are discussed here can be remedied in at least two different ways:

- Allow the existence of several classes for one type. This is the view typically underlying a programming language, where one data type can have several variables of this type, which are distinguished by their names. In addition, this is also the view of ODMG, which considers only a combination of type signature and type implementation as a class (see Section 5.3.1).
- Use object-creating operations instead of object-preserving ones.

The latter option will be discussed in more detail next.

6.3.3 Object-creating operations

Object-creating operations assemble *new* sets of objects from the given class instances, which means that a new result class is created whose objects are characterized by new identities.

Example 6.9

Consider again the instance of class *Automobile* which was used earlier:

	Automobile	
	Drive	*Car body*
#10	#20	Sedan
#11	#21	Hatchback
#12	#22	Sedan

An *object-creating* projection onto attribute *Drive* can then produce the following result:

	Drive
#110	#20
#111	#21
#112	#22

The result class contains new objects; its type is a supertype of the type of the operand.

The result class of an object-creating operation exists parallel to all other classes in the given class hierarchy and thus cannot inherit any methods defined on the operand classes. The most general class Object, if such a class exists, is the only exception. Object creation can be performed in different ways: for example, *implicitly*, which means it cannot be influenced from outside; *freely*, which means it is controlled from outside by the user in more or less any manner; or *functionally*, which means it is controlled from outside in a predefined and controlled manner.

If we assume that an object-creating operation, such as the projection shown in the last example, obtains the values of new objects by copying the values of existing objects, the use of object references might become problematic. In the last example the newly created objects reference the same drive objects as the original automobile objects. One of the decisions in the design of an object algebra with object-creating operations is how tolerable this is.

Concluding this section, we should like to remark that virtually all object algebras proposed in the literature possess object-preserving operations, but only a few have object-creating ones. Selection, projection, various join operations and set operations are included in virtually every object algebra; differences exist, for example, regarding the application of functions in selection predicates, the existence of restructuring operations or the availability of an explicit object-creation operation.

6.4 On the completeness of object-oriented languages

In this section we outline some essential features of the theory of languages for object-oriented databases and try to indicate to what extent formal investigations in this area are based on previous investigations done in terms of the relational model or its extensions, or how they extend these.

One of the principal objectives also pursued by object-oriented databases is to come up with a formally precise notion of a query, in analogy to the above-discussed notion for relational databases. The notion of a query (as a function) is extendible to the notion of a *database transformation* which also encompasses updates. A language for writing (deterministic) transformations generally has the novel capability of *inventing* values: that is, it can introduce and use new domain elements (at least in the intermediate results of a computation). In the context of relational languages this represents a possibility of making a language complete. In terms of object-oriented databases, value invention, applied to identities, now means object creation, and, as already mentioned for algebras, the ability to generate objects in final results is important for object-oriented databases. For that reason, we shall discuss in more detail the concept of database transformation in connection with the type of databases examined here.

Let us assume that D is a set of constants and O is a set of object identities. A *DO isomorphism* is an isomorphism on $D \cup O$, which maps D onto D and O onto O. Analogously, an *O isomorphism* is a *DO* isomorphism h with $h(x) = x$ for each $x \in D$. *DO* isomorphisms on object base instances can now be understood as binary relations, so that it is possible also to consider *non-deterministic* queries; in the presence of object creation this is necessary, since creation cannot normally be controlled from outside. More precisely, a binary relation γ on instances is referred to as a *database transformation* if it satisfies the following conditions:

(1) $(\exists S, S')\ \gamma \subseteq \text{inst}(S) \times \text{inst}(S')$, which means that γ is well typed;

(2) γ is recursively enumerable (which means it is effectively computable);

(3) $(I, J) \in \gamma$ implies $(h(I), h(J)) \in \gamma$ for each *DO* isomorphism h, which means γ is generic;

(4) $(I, J_1,), (I, J_2) \in \gamma$ implies that an O isomorphism h' exists such that $h'(J_1) = J_2$ holds, which means that the output may contain new object identifiers.

Property (4) makes a transformation to what is referred to as *determined* in the literature, and represents a possible limitation of the allowed non-determinism: even though such a transformation is non-deterministic, the possible results of the transformation coincide up to a renaming of the new domain elements, if it is applied to a given input. In other words, object creation happens 'almost deterministically' in a determined transformation: the O isomorphism h' is to be the identity on I. Thus the non-determinism is limited to the newly created objects.

The notion of a determined query hence represents a possible answer to the question of how the notion of a query, according to the above discussion, can be transferred to object-oriented databases. A 'complete' language must then be able to express all determined queries. The language IQL (see the bibliographical notes) is complete in this sense, but only up to an elimination of copies; its incompleteness is due to the fact that the notion of determination does not take into consideration how IQL creates new objects. An extension of IQL to a complete language therefore requires the introduction of a (complex) mechanism for the elimination of copies. A proposal to solve the copy elimination problem is the *constructive* queries. Compared with determined queries, constructive queries represent a restricted concept, which nevertheless seems more adequate in this context. Moreover, IQL is complete with respect to constructive queries, and the same applies to a series of languages equivalent to IQL.

One generalization of determination is *semi-determinism*, where the O isomorphism must leave the (entire) input invariant, which means it must be a DO automorphism. Certain strictly non-deterministic queries or transformations are now admitted; many of these involve choices based on the symmetries (automorphisms) present in the input.

6.5 Bibliographical notes

Foundations of relational algebra are discussed, for example, in Abiteboul et al. (1995), Elmasri and Navathe (1994), Kanellakis (1990), Paredaens et al. (1989), Silberschatz et al. (1997), Ullman (1988) and Vossen (1994). An overview of completeness notions for relational languages with numerous bibliographical references can be found in Vossen (1996); see also Abiteboul et al. (1995) as well as Abiteboul and Vianu (1992). Algebraic concepts for nested relations were introduced by Jäschke and Schek (1982); also refer to Schek and Scholl (1986), Thomas and Fischer (1986), Paredaens et al. (1989), or Gyssens and Van Gucht (1991). Our discussion of algebraic operations on object bases is based on, for example, Beeri (1994), Freytag et al. (1994) and Kim (1990); more on this topic can be found in Abiteboul et al. (1995) or Hull and Su (1993). Of the concrete algebras which have been proposed in the literature we would like to mention the following:

Algebra	*Source*
Cool	Schek and Scholl (1990)
Encore	Shaw and Zdonik (1990)
Excess	Vandenberg and DeWitt (1991)
Reloop	Cluet et al. (1990)
Straube-Algebra	Straube (1990)

Numerous other algebraic operations are proposed or discussed in these publications, for example multiset operations (in Excess), function application, a 'map' operator (in Straube) or join-like operations which have an object-preserving effect (in Cool). A more recent proposal which aims to achieve optimizability and is under discussion as an implementation vehicle for the language OQL of ODMG can be found in Cluet and Moerkotte (1995). Refer to Abiteboul et al. (1995) for a discussion of completeness notions for object-oriented languages; see also Van den Bussche and Van Gucht (1997) as well as Van den Bussche (1993) on this issue. Database transformations go back to the work of Abiteboul and Vianu (1990, 1991). The language IQL is introduced by Abiteboul and Kanellakis (1989).

7 Object orientation and rules

Relational database languages allow a set-oriented processing of relations at a high level of abstraction. They lack, however, the expressive power of common imperative programming languages. One approach to increase the power of relational database languages while still allowing processing at a high level of abstraction is to add deductive rules to the data manipulation language. Deductive rules are first-order predicate logic formulas in an intuitive syntactical form, which allow recursion to be expressed and increase the expressive power.

Languages with a high level of abstraction are attractive because by using them the programmer can concentrate on the *what* of the problem, delegating the actual (algorithmic) *how* of its implementation to the machine. In particular, the machine is provided with a facility to optimize the resulting programs automatically. Languages with a high level of abstraction free the programming process from many cumbersome details which are not only secondary to finding solutions but also obscure the actual structural nature of the problems and consequently contribute to the creation of inscrutable and error-prone programs. Moreover, in comparison with imperative languages, they allow more compact programs to be formulated.

Therefore, it is desirable to try to combine the advantages of rule-based programming with those of object orientation. In the following sections we first explain what we understand by rules and then how rules can be used for computation. We initially confine our discussion to the relational context and do not deal with aspects pertaining to object orientation. Subsequently, we look at a particular approach to the integration of object orientation and rules in more detail.

7.1 Rules

In Section 1.4 we discussed the following SQL expression:

```
SELECT EmplNo
FROM Company, Subsidiary, SubsEmpl, Employee
WHERE Company.Name = 'Ford'
     AND Company.CompanyID = Subsidiary.CompanyID
     AND Subsidiary.Location = 'Ghent'
     AND Subsidiary.CompanyID = SubsEmpl.CompanyID
     AND Subsidiary.NameSubs = SubsEmpl.NameSubs
     AND Subs.Empl = Employee.EmplNo
     AND Employee.Name = 'Lacroix';
```

This expression determines whether an employee with name 'Lacroix' is employed in Ford's subsidiary in Ghent.

To give an intuitive understanding of the nature of a rule, we shall rewrite the above SQL expression. In essence, a rule is made up of a conclusion, the so-called *head* of the rule, and a condition, the so-called *body* of the rule; both are separated by an implication symbol (←). By a rule we define the content of a relation, which is the set of those tuples for which the conditions of the body are fulfilled. Intuitively, a rule is an *if–then* instruction: *if* the statement in the body can be fulfilled, that is, its conditions are said to be *true*, *then* the statement in the head is also to apply. Accordingly, we obtain the following rule in our example:

```
Result (Z) ←  Company(X, 'Ford',_,_,_),
              Subsidiary(X,Y,_, 'Ghent' ,_),
              SubsEmpl(X,Y,Z),
              Employee(Z, 'Lacroix',...)
```

Relation Result contains exactly one tuple for employee 'Lacroix', provided he meets the stipulated requirements in the body of the rule. The employee 'Lacroix' is said to meet the requirements, if there are tuples in the relations Company, Subsidiary, SubsEmpl and Employee, for which the following holds. The tuples must be equal in their first component with respect to Company, Subsidiary and SubsEmpl; the second component of the Company tuple has the value 'Ford' and so on. Note that the body of the rule consists of a series of expressions – so-called *atoms* – in the form $R(a_1, ..., a_k)$ with R being the name of the relation in question and $a_1, ..., a_k$ being constants or variables. Each occurrence of '_' denotes a new variable, which is distinct from all variables occuring anywhere else. In order to test whether relation R contains a desired tuple, we bind each variable by a constant. If the body is made up of several atoms, testing must be conducted simultaneously for every atom in order to consider the same variables with regard to the same binding in all atoms.

Rules express *if–then* instructions: *if* there are tuples which hold true for the condition on the right-hand side of the rule, *then* the components of these tuples can be used to generate new tuples whose structure takes on the form on the left-hand side; for these tuples the head of the rule is said to hold true, as well.

We shall now examine these issues in greater detail. First, we shall deal with the syntax of rules. Let us suppose that **R** is a set of *relational names* $\mathbf{R} = \{R_1, ..., R_n\}$ with each such R having assigned a fixed number of arguments $k \geq 1$.

A *rule* can be written as a formula as follows:

$$\forall X_1 \ldots \forall X_n(H \leftarrow (G_1 \wedge \ldots \wedge G_m)),$$

where H and $G_1, ..., G_m$ are atoms, $m \geq 0$, and $X_1, ..., X_n$ are all the variables appearing in the rule. A set of rules is called a *rule program P*.

We prefer the use of relational names over predicate names, because we consider rules as a means to increase the power of relational database languages. '$\leftarrow$' denotes logical implication and $\wedge$ denotes conjunction. H is also referred to as the *head* of the rule and $(G_1 \wedge \ldots \wedge G_m)$ as the *body* of the rule. Since every variable of a rule is always $\forall$-quantified, we shall omit the quantifier prefix as well as the outer parentheses. Furthermore, we substitute a comma for the conjunctions in the body and omit the parentheses of the body. Rules therefore can be written as follows:

$$H \leftarrow G_1, ..., G_m$$

A special case of interest is a rule with an empty body, that is, $m = 0$ applies, for which the head H does not contain a variable. Instead of $(R(a_1, ..., a_k) \leftarrow)$ we write shorter $R(a_1, ..., a_k)$ and refer to it as a *fact*. An expression is called *ground* if it is variable-free.

Rules which are defined in this way are also called *Horn*-rules. It is characteristic of Horn-rules that neither in the body nor in the head of the rule is a negation symbol allowed to appear. For the following discussions it is sufficient to restrict ourselves to Horn-rules. Let us suppose that P is a given rule program; a relational name R is said to be an *input name* of P, if in no rule of P does the name R appear in its head.

Rules define the *contents* of relations. *If* all variables can be bound by constants in such a way that all atoms of a body are true, *then*, correspondingly, the head of the rule holds true as well. We shall now look at the semantics of a rule program. For this purpose we consider a mapping I which assigns to every relational name R a corresponding relation r; I is called an *interpretation* of **R**.

Let a *substitution* be defined as a set θ of variable bindings $[X_1 / a_1, ..., X_r / a_r]$ with X_i denoting the various variables and a_i constants, $1 \leq i \leq r$. If G is an atom and θ a substitution of the variables in G, then θ applied to G, expressed as $G\theta$, denotes a ground atom generated from G by substituting a constant for every variable X in G according to θ.

Let I be an interpretation.

- A ground atom $G = R(a_1, ..., a_k)$ is true under I if $(a_1, ..., a_k) \in I(R)$.
- A rule $H \leftarrow G_1, ..., G_m$ is true under I if for every substitution θ of the variables such that the ground atoms $G_1\theta, ..., G_m\theta$ are true under I, $H\theta$ also holds true under I.

Let I_e be an interpretation which assigns a relation to the input names for P and the empty set to all other relational names. I_e is called the *input*. An interpretation M is a called *model* for a rule program P with respect to input I_e, if for every $R \in \mathbf{R}$ $M(R) \supseteq I_e(R)$ applies and each rule of P is true under M. M is called a minimal model of P with respect to the input I_e, if for every other model M^* with respect to I_e for all relational names $R \in \mathbf{R}$, $M(R) \subseteq M^*(R)$ holds.

It is well known in the literature that there exists a unique minimal model for Horn-rule programs. The *meaning* of a Horn-rule program is given by its minimal model.

Example 7.1

Let us assume that the schema of a relation Employee is defined as follows:

```
CREATE TABLE Employee
(EmplNo INT NOT NULL,
Name VARCHAR NOT NULL,
Boss INT NOT NULL,
PRIMARY KEY (EmplNo),
FOREIGN KEY (Boss)
REFERENCES Employee (EmplNo)
);
```

Let us consider a relation for this schema:

Employee

EmplNo	Name	Boss
100	abc	200
101	def	200
200	ghi	300
201	jkl	300
300	mno	400
400	pqr	400

If we want to determine for each employee the direct and indirect superiors, we can define a corresponding relation Superior by means of the two following rules:

```
Superior(X,Y) ← Employee(X,_,Y)
Superior(X,Y) ← Employee(X,_,Z), Superior(Z,Y)
```

The first rule defines every *direct* superior as a superior; the second rule defines the *indirect* superiors. Note that the second rule is recursive, because the relation Superior occurs in both the head and the body of the rule.

Let us now assume that the above-stated relation is an input to the rule program. The meaning of the program is given by its minimal model:

Superior	
EmplNo	*Boss*
100	200
101	200
100	300
101	300
100	400
101	400
200	300
201	300
200	400
201	400
300	400
400	400

We now want to demonstrate how we can compute the minimal model of a given rule program. First we need to clarify what we understand by *applying* a rule to the relations which are mentioned in its body. Let us, therefore, consider the first rule of the above example:

```
Superior(X,Y) ← Employee(X,_,Y)
```

Since all variables of this rule are implicitly ∀-quantified, the rule states that for each substitution of the variables X,_,Y, with the property of the resulting (X,_,Y)-tuple being a tuple of the relation Employee, there has to be a tuple (X,Y) in the relation Superior as well. The rule is applied in such a manner that first *all* (X,_,Y)-tuples are determined and subsequently, using the X- and Y-components of all these tuples, corresponding tuples of the relation Superior are generated.

We can apply this simple process to the second rule as well:

```
Superior(X,Y) ← Employee(X,_,Z), Superior(Z,Y)
```

Since all variables of the rule are again ∀-quantified, the rule here states that for every substitution of the variables X,_,Z,Y, with the property of the resulting (X,_,Z)-tuple being a tuple of the relation Employee and on the other hand the corresponding resulting tuple (Z,Y) being a tuple of the relation Superior, there must also be a tuple (X,Y) in the relation Superior. The rule is now applied in such a manner that first all (X,_,Z,Y)-tuples are generated by way of calculating the natural join between the relations Employee and Superior. In order to compute the join, the variables are treated as attributes of the relations. Subsequently, the respective tuples of the relation Superior are generated using the X- and Y-components of the tuples of the relation which has been computed by means of the join.

Applying the rules as described, we can derive the relations defined in the head of the rules. The content of such a relation always depends on the contents of the relations in the body of the rule under consideration. Since these relations may themselves be defined by rules, for example in the case of recursive rules, an iterative approach is used in which the rules are applied until the relations defined by the rules do not change any further. New tuples can only be generated by using the constants in the tuples of the relations of the body; therefore, as relations are assumed to be finite, the termination of this iteration process and thus the computation of the minimal model for the rule program under consideration is guaranteed.

Example 7.2

The iteration process needed to compute the relations defined by the rules is best illustrated by looking at the *rounds* of the iteration of the applications of the rules. In each round all rules are simultaneously applied to the relations which exist at the beginning of a round. Since the relation to Superior is empty at the beginning, the following round protocol emerges. We list only the changes to relation `Superior`: that is, we restrict ourselves to the tuples which are newly derived in the respective round:

Round protocol

Round	*Superior*
0	∅
1	(100, 200), (101, 200), (200, 300), (201, 300), (300, 400), (400, 400)
2	(100, 300), (101, 300), (200, 400), (201, 400)
3	(100, 400), (101, 400)

The technique described in the previous example is applicable to any rule program. It is also known as *forward chaining* or *bottom-up* calculation. For a more in-depth discussion of this method we refer the reader to the literature, particularly regarding the rules which are not Horn-rules. What we have explained in this section will be sufficient for the comprehension of what is to follow.

7.2 Object orientation

In this section we should like to study the way in which complex object structures can be processed by the rules. For this purpose let us return to our running example, but now using an object-oriented setting (cf. Section 1.4). We have the following class definitions:

```
Company: [
      Name: String,
      Headoffice: [Street: String, Location: String],
      Subsidiaries: {Subsidiary},
      President: Employee]
```

```
Subsidiary: [
        Name: String,
        Office: [ Street: String, Location: String ],
        Manager: Employee,
        Employees: { Employee }]

Employee: [
        Name: String,
        ... ]
```

To define rules on such complex object structures is difficult for various reasons. The obvious approach would be to use class names instead of relational names. An atom then would be an expression $K(a_1, ..., a_k)$ with K being a class name and $a_1, ..., a_k$ being constants or variables. The question now is what kind of domain would be appropriate, because we should like on the one hand to allow the processing of complex values, such as tuple values for addresses or set values, and on the other hand to have access to, for example, the location in an address or an element of a set. Moreover, such a domain must also include object identities.

7.2.1 Complex objects

Let **D** be the set of all atomic values: that is, values of type integer, float, boolean or string. **D** is the *domain* of our object-oriented rule language. In addition to representing the properties of objects, we shall assume that elements of **D** can also be used in the role of *object identities*. If a string is used for this purpose, it may also be termed an *object name* if this appears to be more appropriate. Object identities are indicated by the '#'-symbol, as we have already done earlier. It should be noted that *complex* values are not included in **D**. Our framework will allow a formally simple treatment of complex objects, avoiding semantic problems that may occur if, for example, complex values were contained in the domain as well (cf. the literature cited in Section 7.3).

Complex objects are called *molecules*. A molecule is an expression in the form *O*[*method-calls*] where *O* is an *ID-term* and *method-calls* is a sequence of scalar and set-valued expressions separated by ';' of the form:

$$ScalarMethod@Q_1, ..., Q_k \rightarrow T$$
$$SetMethod@Q_1, ..., Q_k \rightarrow \{S_1, ..., S_m\}$$

where $k, m \geq 0$. For the time being, an ID-term is either a constant or a variable; later we shall need a more general form. ID-terms serve to identify objects. Therefore, constants at the position of O have the role of object identities. For representing variables we choose capital letters from the end of the alphabet or class names spelled in lower-case letters – as we have done in the preceding chapters. Variables of the latter kind can be bound only by object identities which occur in the extensions of the respective classes; these variables are referred to as *class variables*.

Within a method call *ScalarMethod* and *SetMethod* represent names of methods and $Q_1, ..., Q_k$, T and $S_1, ..., S_m$ represent ID-terms. The Q_i are the parameters of the method calls and T and S_i respectively are the results. If constants are used in these positions, they may take the role of an object identity or of a 'normal' value, depending on the signatures of the methods. We do not make a distinction between attributes and methods; attributes are treated as methods without parameters. The symbol @ is omitted if there are no parameters.

Example 7.3

To give an example for a molecule let us look at object #41 of class `Company` (see also Section 1.4):

```
#41[ Name —> 'VW';
Headoffice —> #80;
Subsidiaries —> { #50,#51 };
President —> #60 ]
```

Note that, unlike in the above-mentioned example, the addresses of the head offices are represented by object identities and not listed directly as complex values. This is a consequence of our formal framework in which complex values are not contained in the domain.

The following molecule is a query asking for certain subsidiaries of a company:

```
X[ Name —> 'South';
Office —> #83;
Manager —> U;
Employees —> { Z } ]
```

This query asks for the identity, the manager and the set of employees working in a subsidiary with name `'South'`. Note that in terms of syntax we are not searching for the set of employees working in the subsidiary but merely for an element. We obtain the set of all employees as the set of all possible elements of this set. We shall discuss sets in more detail in section 7.2.4.

7.2.2 Rules

As before, rules are expressions in the form

$$H \leftarrow G_1, ..., G_m,$$

where H and $G_1, ..., G_m$ are molecules, $m \geq 0$.

Example 7.4

With the following rule we are looking for our well-known employee 'Lacroix'. *Result* is a predicate:

```
Result (Y) ←company[ Name→'Ford'; Subsidiaries→{X} ],
            X[ Office → _ [Location → 'Ghent' ]; Employees → {Y}],
            Y [ Name → 'Lacroix' ]
```

Again, variables can be represented by '_', if their binding is of no interest; moreover, we can confine ourselves to those method calls in a molecule which are of interest to define the properties of the object referenced in the head of the rule. In situations where we have a rule with the structure

... ← ..., O[... → X], ..., X[*method-calls*],...

we can more compactly write:

... ← ..., O[... → X[*method-calls*]],...

Thus the above rule can be written in a more compact form:

```
Result (Y) ←company[ Name→'Ford';
                      Subsidiaries→{ X[Office→_ [Location→'Ghent'];
                              Employees→{Y[Name→'Lacroix']}]}]
```

In this example the resulting body of the rule consists of only one molecule. Such a body resembles a path expression.

Next we shall turn our attention to semantics. For this purpose we need to transform molecules into their atomic components. An *M-atom* is an expression of one of the following forms:

$O[ScalarMethod@Q_1, ..., Q_k \rightarrow T]$
$O[SetMethod@Q_1, ..., Q_k \rightarrow \{S\}]$
$O[SetMethod@Q_1, ..., Q_k \rightarrow \{\,\}]$

where O, Q_1, ..., Q_k, S, T are ID-terms. Note that with respect to an M-atom, a set-valued method accepts only sets consisting of one element or the empty set.

The set of M-atoms for a molecule G is obtained as follows: Let G be of the form O[*method-call*, ...; *method-call*] with *method-call* being a scalar or a set-valued expression of the form:

$ScalarMethod@Q_1, ..., Q_k \rightarrow T$
$SetMethod@Q_1, ..., Q_k \rightarrow \{S_1, ..., S_m\}$

Each scalar method-call $ScalarMethod@Q_1, ..., Q_k \rightarrow T$ implies an M-atom:

$$O[ScalarMethod@Q_1, ...,Q_k \rightarrow T]$$

Each set-valued method-call $SetMethod@Q_1, ..., Q_k \rightarrow \{S_1, ..., S_m\}$ implies the following M-atoms:

$$O[SetMethod@Q_1, ..., Q_k \rightarrow \{\,\}]$$
$$O[SetMethod@Q_1, ..., Q_k \rightarrow \{S_1\}], ...$$
$$O[SetMethod@Q_1, ..., Q_k \rightarrow \{S_m\}].$$

An interpretation I is now a set of ground M-atoms and, correspondingly, I_e is a set of ground M-atoms called the *input*. The definition of a substitution remains largely the same; we also require that class variables must be bound only by object identities of the respective class extension. Thus we define as follows:

- A ground M-atom G is *true* under I if $G \in I$ applies.
- A ground molecule G is *true* under I if each of its M-atoms is true under I.
- A rule $H \leftarrow G_1, ..., G_m$ is *true* under I if for each substitution θ of the variables, for which the M-atoms of the ground molecules $G_1\theta, ..., G_m\theta$ are true under I, $H\theta$ also is true under I.

Models are defined in the same way as before. The meaning of a rule program with respect to a given input I_e is the unique minimal model which contains the input. A minimal model of this kind can be derived by means of a bottom-up process. However, when we permit more general ID-terms later, termination is not always guaranteed.

Example 7.5

The following rules define the superiors for certain employees.

```
Result [ Superiors @ employee → { X } ] ← employee [ Superior → X ]
Result [ Superiors @ employee → { X } ] ← Result [ Superiors @ employee
                                              → { Y } ], Y [ Superior → X ]
```

It should be noted that in this example employee is a class variable which is bound only by object identities of class `Employee`. To derive object `Result` a bottom-up evaluation may be carried out in the same manner as in Example 7.1. After three rounds we get the following:

```
Result [ Superiors @ 100 → { 200, 300, 400 } ]
Result [ Superiors @ 101 → {200, 300, 400 } ]
Result [ Superiors @ 200 → { 300, 400 } ]
Result [ Superiors @ 201 → { 300, 400 } ]
Result [ Superiors @ 300 → { 400 } ]
Result [ Superiors @ 400 → { 400 } ]
```

7.2.3 Object identities

Thus far we have only used rules in order to define the truth value of predicates (cf. Example 7.4) or to define results of methods with regard to a given object (cf. Example 7.5). But often one wishes to define new objects without knowing in advance how many and what kind of objects there will be.

Example 7.6

For each pair `Company-Name` and `Employee-Name` we want to define an object for which two methods, one for each of the respective names, are defined. This is difficult inasmuch as we do not know how many objects need to be defined, and consequently, it is also not clear which object identities are required:

```
? [Company-Name → X, Employee-Name → Y ] ←
                               company[ Name → X; Subsidiaries → {
                                         X[Employees → { _[Name → Y]}]}]
```

In the previous example one object is required for each pair of names; therefore for each such object an identity is needed. In order to achieve this we have to use ID-terms in a more general way. Let **F** be a set of names which we shall call *object constructors*. Object constructors give us the required flexibility to define ID-terms. We proceed as follows:

- Every constant and every variable is an ID-term.
- Let $t_1, ..., t_l$, $l \geq 1$ be ID-terms and $f \in$ F an object constructor. Then $f(t_1, ..., t_l)$ is also an ID-term.

Let **O(F)** be the set of all ground ID-terms; every element of this set is a potential object identity. As a consequence, we extend our previous domain **D** by the elements of **O(F)**.

In order to complete our last example, we shall use the object constructor Comp-Empl to derive the needed object identities. We can write the head of the rule as follows:

```
Comp-Empl(X,Y) [Company-Name → X, Employee-Name → Y ]
```

Since rules are implicitly ∀-quantified, a new object is defined for each substitution that makes the body true. Observe that it is ensured that exactly one object identity will be defined for each distinct pair of names.

Example 7.7

The following two rules define objects describing company locations; the first rule defines one such object, even if several companies have their head offices in that location, whereas the second rule defines an object for each company notwithstanding that another company may have its head office in the same location.

```
Co-Location(X) [Location →X] ← company[Headoffice →_[Location →X]]
Location-Co(company) [Location →X] ← company[Headoffice →
                                          _[Location →X]]
```

`Co-Location` and `Location-Co` are object constructors. Note that according to the choice of variables the second rule may distinguish between objects which have the same values with respect to their methods. This means that we can handle duplicates in the sense of objects with equal values.

7.2.4 Processing of sets

The approach we have chosen does not allow variables to be quantified over sets. This follows from the fact that our underlying domain **D** does not contain sets. The obvious question which now immediately arises is whether in such a framework the processing of sets can still be done in a reasonable way. In order to answer this question we shall examine several examples.

Example 7.8

A typical operation for the definition of sets is *grouping*, respectively *nesting*. Grouping means that elements that fulfil a certain condition are to be part of a set. With the help of the following rule we group into a set all employees who have the same superior:

```
Nested { Group @ X → { employee } ] ← employee [ Superior →X ]
```

A nested structure can be unnested as follows:

```
employee [ Superior →X ] ←Nested [ Group @ X → { employee } ]
```

Since our framework allows sets to be defined by elements only, we have to show that a set as a whole is defined as well. The following serves to clarify this point. Let us consider the two M-atoms $a[m \twoheadrightarrow \{1\}]$ and $a[m \twoheadrightarrow \{2\}]$ with m being a set-valued method. Let us further assume that I is an interpretation for which both M-atoms are true. Since a ground molecule is true under an interpretation when all its ground M-atoms are true, $a[m (\twoheadrightarrow \{1,2\}]$ is true under I as well.

Another interesting question is how equality of sets can be tested, since sets cannot be compared in their entirety by equating variables, for example. Here, we can distinguish two aspects.

Let us assume that we want to regard every set as an object. For example, let a, b be the identities of such set objects: $a[m \twoheadrightarrow \{...\}]$ and $b[m \twoheadrightarrow \{...\}]$. If $a = b$ holds true, then a and b represent the same set. Testing the equality of two sets is then reduced to testing the equality of two object identities.

Now let us consider set-valued methods. We need rules to determine whether two set-valued methods define the same sets. The required rules for this purpose, however, contain negations in the body and are therefore beyond the scope of our formal framework. For the sake of completeness we shall look at the needed rules briefly. Consider the two M-atoms $a[m \rightarrow \{...\}]$ and $b[m \rightarrow \{...\}]$. The rules listed below test whether m defines the same set with regard to a and b. The third rule can be applied after the truth-value of *unequal* has been determined.

$unequal \leftarrow a[m \rightarrow \{X\}], \neg b[m \rightarrow \{X\}]$
$unequal \leftarrow b[m \rightarrow \{X\}], \neg a[m \rightarrow \{X\}]$
$equal \leftarrow \neg\, unequal$

The usual set operations can be expressed in a similar way. We consider, for instance, $\cup$, $\cap$ and $\setminus$:

$union(a,b)[m \rightarrow \{X\}] \leftarrow a[m \rightarrow \{X\}]$
$union(a,b)[m \rightarrow \{X\}] \leftarrow b[m \rightarrow \{X\}]$

$intersect(a,b)[m \rightarrow \{X\}] \leftarrow a[m \rightarrow \{X\}], b[m \rightarrow \{X\}]$

$minus(a,b)\ [m \rightarrow \{X\}] \leftarrow a[m \rightarrow \{X\}], \neg b[m \rightarrow \{X\}]$

The above rules define new objects which are identified by *union*(a,b), *intersect*(a,b) and *minus*(a,b), depending on the objects a, b to be compared. The names *union*, *intersect* and *minus* are used as object constructors here.

7.2.5 Classes

Let us now look at objects in connection with classes. We assume that **K** is a set of classes. Objects are associated with classes, that is, defined to be members of the class extension, by means of *ISA-terms* of the form $O : K$, with O being an ID-term and K a class of **K**. Classes are arranged in a hierarchy denoted *IsA*.

IsA-terms and molecules referring to the same objects can be integrated to form one expression. If $O[ScalarMethod@Q_1, ..., Q_k \rightarrow T]$ is a molecule, then

$O : K\ [ScalarMethod@Q_1 : K_1, ..., Q_k : K_k \rightarrow T : K']$

integrates the additional information $O : K$, $Q_1 : K_1, ..., Q_k : K_k$ and $T : K'$. For example, in

```
X: Company [ Name → Y; Subsidiaries → { Y:Subsidiary [Employees → {
                                          Z:Employee }]}]
```

it is established that X must be bound only by ID-terms of class *Company*, Y only by ID-terms of class *Subsidiary* and Z only by ID-terms of class *Employee*.

IsA-terms may occur in both the head and the body of a rule. Let us look at the semantics. An interpretation now is a set of ground M-atoms and ground IsA-terms.

- A ground M-atom or a ground IsA-term G is *true* under I if $G \in I$ applies.
- A ground molecule G is *true* under I if each of its M-atoms and IsA-terms is true under I.
- A rule $H \leftarrow G_1, ..., G_m$ is *true* under I if for each substitution θ of the variables, for which the ground M-atoms or ground IsA-terms $G_1\theta, ..., G_m\theta$ are true under I, $H\theta$ is also true under I.

Class extensions can be defined by rules, as demonstrated in the next example.

Example 7.9

For class `Subsidiary` we have so far looked at the method `Manager` which delivers the manager to a subsidiary. The following rules define the inverse relationship by indicating for each manager the branches where he or she is manager. The first rule defines a set-valued method, but the second rule defines a scalar method. Since different subsidiaries having the same manager should not be ruled out, the latter rule defines an individual object for each relationship between manager and subsidiary. `Result` is a class to collect the objects defined by the respective rule.

```
Subs-Man(X):Result [Man-Name → X; Subs-Name → { Y }] ←
        _:Subsidiary [Name → Y, Manager → _:Employee [ Name → X ] ]

Subs-Man(X,Y):Result [Man-Name → X; Subs-Name → Y] ←
         _:Subsidiary [Name → Y, Manager →_:Employee [ Name →  X ] ]
```

Whenever class extensions are defined by rules, this mechanism can be regarded as analogous to the definition of views in relational databases.

Example 7.10

The following rule defines that each employee who is the manager of a subsidiary is also an instance of the class `Man-Cla`. Note that, compared with the previous example, no new objects are created here; however, the object is defined to become an instance of an additional class.

```
X : Man-Cla ← _:Subsidiary [Manager → X:Employee]
```

7.3 Bibliographical notes

There are a number of approaches incorporating object orientation and rules; however, they are mainly different in the way in which they treat sets and, if applicable, object identities. In LDL (Beeri et al. 1987) sets are permitted as elements of the domain, and the semantic implications are discussed in detail. Abiteboul and Grumbach (1991) examine the rule language COL, which introduces set-valued data functions for the treatment of sets. Kuper (1990) presents the rule language ELPS in which $\forall$-quantifiers in the body of rules are permitted for the treatment of sets. In IQL (Abiteboul and Kanellakis, 1989) rules and object identities and in particular the problem of elimination of copies is discussed.

The rule language underlying our discussion is Frame-Logic (Kifer et al., 1995), which itself is based on O-logic as regards the treatment of sets and object identities (Kifer and Wu, 1993). Other important concepts of Frame-Logic not discussed here are data-dependent class hierarchies, (monotonic) inheritance of signatures and (non-monotonic) inheritance of values. Moreover, Frame-Logic has a higher-order syntax with a first-order semantics, which allows an integrated processing of data and metadata.

There are several prototypes for object-oriented rule languages; we shall mention only some of them. ConceptBase (Jarke et al. 1995) is a rule-based object manager, which is primarily intended for the management of metadata. Chimera (Ceri et al., 1996) integrates object orientation, deductive rules and active rules, whereas in Rock&Roll (Barja et al., 1994) and Coral++ (Srivastava et al., 1993) the integration of a rule language is achieved with an object-oriented imperative language. In QUIXOTE (Yasukawa et al., 1992), the object-oriented representation of knowledge is emphasized and, finally, FLORID (Frohn et al. 1997) implements almost all features of Frame-Logic.

For more information about the prototypes contact the following Web pages:

For ConceptBase:
http://www-i5.informatik.rwth-aachen.de/CBdoc/cbflyer.html

For Chimera:
http://www.elet.polimi.it/section/compeng/db/active/idea/chimera

For Coral++:
http://www.cs.wisc.edu/coral/

For FLORID:
http://www.informatik.uni-freiburg.de/~dbis/flsys/

For QUIXOTE:
http://www.icot.or.jp/ICOT/IFS/IFS-abst/011.html

For Rock&Roll:
http://www.cee.hw.ac.uk/Databases/rnr.html

References

Abiteboul S. and Grumbach S. (1991). A rule-based language with functions and sets. *ACM Transactions on Database Systems*, **16**, 1–30

Abiteboul S. and Kanellakis P.C. (1989). Object identity as a query language primitive. In *Proc. ACM SIGMOD International Conference on Management of Data*, 159–73. New York: ACM

Abiteboul S. and Van den Bussche J. (1995). Deep equality revisited. In *Proc. 4th International Conference on Deductive and Object-Oriented Databases.* LNCS 1013, pp. 213–28. Berlin: Springer-Verlag

Abiteboul S. and Vianu V. (1990). Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, **41**, 181–229

Abiteboul S. and Vianu V. (1991). Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, **43**, 62–124

Abiteboul S. and Vianu V. (1992). Expressive power of query languages. In *Theoretical Studies in Computer Science* (Ullman J.D., ed.), pp. 207–51. Boston, MA: Academic Press

Abiteboul S., Hull R. and Vianu V. (1995). *Foundations of Databases.* Reading, MA: Addison-Wesley

Atkinson M., Bancilhon F., DeWitt D., Dittrich K., Maier D. and Zdonik S. (1989). The object-oriented database system manifesto. In *Proc. 1st International Conference on Deductive and Object-Oriented Databases*, pp. 40–57

Bancilhon F. and Ferran G. (1995). The ODMG Standard for Object Databases. In *Proc. 4th International Conference on Database Systems for Advanced Applications* (DASFAA), pp. 273–83

Bancilhon F., Barbedette G., Benzaken V., Delobel C., Gamerman S., Lecluse C., Pfeffer P., Richard P. and Velez F. (1988). The design and implementation of O_2, an object-oriented database system. In *Advances in Object-Oriented Database Systems* (Proc. 2nd International Workshop on Object-Oriented Database Systems) (Dittrich K.R., ed.), pp. 1–22. LNCS 334. Berlin: Springer-Verlag

Bancilhon F., Delobel C. and Kanellakis P. (1992). *Building an Object-Oriented Database System – The Story of O_2.* San Francisco, CA: Morgan Kaufmann

Barja M.L., Paton N.W., Fernandes A.A.A., Williams M.H. and Dinn A. (1994). An effective deductive object-oriented database through language integration. In *Proc. 20th International Conference on Very Large Data Bases*, pp. 463–74. San Francisco, CA: Morgan Kaufmann

Beeri C. (1994). Query languages for models with object-oriented features. In *Advances in Object-Oriented Database Systems* (Dogac A., Özsu M.T., Biliris A. and Sellis T., eds). NATO ASI Series F: Computer and System Sciences, Vol. 130, pp. 47–71. Berlin: Springer-Verlag

Beeri C., Naqvi S., Shmueli S. and Tsur S. (1987). Sets and negation in a Logic Database Language (LDL). In *Proc. 6th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 21–37. New York: ACM

Bernstein P.A., Hadzilacos V. and Goodman N. (1987). *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley

Bertino E. and Martino L. (1991). Object-oriented database management systems: concepts and issues. *IEEE Computer*, **24**(4), 33–47

Bertino E. and Martino L. (1993). *Object-Oriented Database Systems*. Wokingham: Addison-Wesley

Booch G. (1991). *Object Oriented Design with Applications*. Redwood City, CA: Benjamin/Cummings

Brodie M.L. and Stonebraker M. (1995). *Migrating Legacy Systems – Gateways, Interfaces & The Incremental Approach*. San Francisco, CA: Morgan Kaufmann

Bukhres O.A. and Elmagarmid A.K., eds (1996). *Object-Oriented Multidatabase Systems*. Englewood Cliffs, NJ: Prentice-Hall

Butterworth P., Otis A. and Stein J. (1991). The GemStone object database management system. *Communications of the ACM*, **34**(10), 64–77

Cattell R.G.G. (1994). *Object Data Management – Object-Oriented and Extended Relational Database Systems*, revised edition. Reading, MA: Addison-Wesley

Cattell R.G.G., ed. (1996). *The Object Database Standard: ODMG-93, Release 1.2*. San Francisco, CA: Morgan Kaufmann

Ceri S., Fraternali P., Paraboschi S. and Brance L. (1996). Active rule management in Chimera. In *Active Database Systems* (Widom J. and Ceri S., eds), pp. 151–76. San Francisco, CA: Morgan Kaufmann

Chamberlin D. (1996). *Using the New DB2 – IBM's Object-Relational Database System*. San Francisco, CA: Morgan Kaufmann

Cluet S. and Moerkotte G. (1995). Nested Queries in Object Bases. *Technical Report 95-6*, Dept. of Computer Science, Techn. Univ. of Aachen, Germany

Cluet S., Delobel C., Lecluse C. and Richard P. (1990). RELOOP, an algebra based query language for an object-oriented database system. *Data & Knowledge Engineering*, **5**, 333–52

Codd E.F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, **13**, 377–87

Date C.J. (1995). *An Introduction to Database Systems* Vol. 1, 6th edn. Reading, MA: Addison-Wesley

Date C.J. and Darwen H. (1993). *A Guide to the SQL Standard*, 3rd edn. Reading, MA: Addison-Wesley

Deux O. et al. (1990). The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, **2**, 91–108

Dittrich K.R., Dayal U. and Buchmann A.P. (1991). *On Object-Oriented Database Systems*. Berlin: Springer-Verlag

Elmasri R.A. and Navathe S.B. (1994). *Fundamentals of Database Systems*, 2nd edn. Redwood City, CA: Benjamin/Cummings

Freytag J.C., Maier D. and Vossen G., eds (1994). *Query Processing for Advanced Database Systems*. San Francisco, CA: Morgan Kaufmann

Frohn J., Lausen G. and Uphoff H. (1994). Access to objects by path expressions and rules. In *Proc. 20th International Conference on Very Large Data Bases*, pp. 273–84. San Fancisco, CA: Morgan Kaufmann

Frohn J., Himmeröder R., Kandzia P., Lausen G. and Schlepphorst C. (1997). FLORID – a prototype for F-logic. In *Proc. 13th Conference on Data Engineering*, pp. 583. Los Alamos, CA: IEEE Computer Society Press

Gray J. and Reuter A. (1993). *Transaction Processing 7: Concepts and Techniques*. San Mateo, CA: Morgan Kaufmann

Gupta R. and Horowitz E. (1991). *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*. Englewood Cliffs, NJ: Prentice-Hall

Gyssens M. and Van Gucht D. (1991). A comparison between algebraic query languages for flat and nested databases. *Theoretical Computer Science*, **87**, 263–86

Hull R. and Su J. (1993). Algebraic and calculus query languages for recursively typed complex objects. *Journal of Computer and System Sciences*, **47**, 121–56

Jarke M., Gallersdörfer R., Jeusfeld M.A., Staudt M. and Eherer S. (1995). ConceptBase – a deductive object base for meta data management. *Journal of Intelligent Information Systems*, **4**, Special Issue on Advances in Deducutive Object-Oriented Databases, 167–92

Jäschke G. and Schek H.J. (1982). Remarks on the algebra of non first normal form relations. In *Proc. 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 124–38. New York: ACM

Kanellakis P. (1990). Elements of relational database theory. In *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics* (Van Leeuwen J., ed.), pp. 1073–156. Amsterdam: North-Holland

Kanellakis P., Lecluse C. and Richard P. (1992). Introduction to the data model. In *Building an Object-Oriented Database System – The Story of* O_2 (Bancilhon F., Delobel C. and Kanellakis P., eds), pp. 61–76. San Francisco, CA: Morgan Kaufmann

Kemper A. and Moerkotte G. (1994). *Object-Oriented Database Management – Applications in Engineering and Computer Science*. Englewood Cliffs, NJ: Prentice-Hall

Khoshafian S.N. (1993). *Object-Oriented Databases*. New York: Wiley

Khoshafian S.N. and Abnous R. (1990). *Object Orientation – Concepts, Languages, Databases, User Interfaces*. New York: Wiley

Kifer M. and Wu J. (1993). A logic for programming with complex objects. *Journal of Computer and System Sciences*, **47**, 77–120

Kifer M., Kim W. and Sagiv Y. (1992). Querying object-oriented databases. In *Proc. ACM SIGMOD International Conference on Mangement of Data*, pp. 393–402. New York: ACM

Kifer M., Lausen G. and Wu J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, **42**, 741–843

Kim W. (1990). *Introduction to Object-Oriented Databases*. Cambridge, MA: The MIT Press

Kim W., ed. (1995). *Modern Database Systems – The Object Model, Interoperability, and Beyond*. Reading, MA: Addison-Wesley

Kim W. and Lochovsky F.H., eds (1989). *Object-Oriented Concepts, Databases, and Applications*. Reading, MA: Addison-Wesley

Kleindienst J., Plasil F. and Tume P. (1996). Lessons learned from implementing the CORBA persistent object service. In *Proc. OOPSLA '96 Conference*, ACM SIGPLAN Notices **31**(10), 150–67

Kuper G.M. (1990). Programming with sets. *Journal of Computer and System Science*, **41**, 44–64

Lamb C., Landis G., Orenstein J. and Weinreb D. (1991). The ObjectStore database system. *Communications of the ACM*, **34**(10), 50–63

Lecluse C. and Richard P. (1989). The O_2 database programming language. In *Proc. 15th International Conference on Very Large Data Bases*, pp. 411–22. San Francisco, CA: Morgan Kaufmann

Lecluse C., Richard P. and Velez F. (1988). O_2, an object-oriented data model. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 424–33. New York: ACM, and in *Proc. 1st International Conference on Extending Database Technology*. LNCS 303, pp. 556–62. Berlin: Springer-Verlag

Loomis M.E.S. (1995). *Object Databases – The Essentials*. Reading, MA: Addison-Wesley

Maier D., Stein J., Otis A. and Purdy A. (1986). Development of an object-oriented DBMS. In *OOPSLA '86 Proceedings*, pp. 472–82

Manola F. (1991). Object Data Language Facilities for Multimedia Data Types. *Technical Report TR-0169-12-91-165*. Waltham, MA: GTE Laboratories

Manola F. (1994). An Evaluation of Object-Oriented DBMS Developments – 1994 Edition. *Technical Report TR-0263-08-94-165*. Waltham, MA: GTE Laboratories

Melton J. and Simon A.R. (1993). *Understanding the New SQL: A Complete Guide*. San Francisco, CA: Morgan Kaufmann

Moss E., ed. (1994). Special issue on emerging object query standards. *Bulletin of the Technical Committee on Data Engineering*, **17**(4). IEEE Computer Society

Object Management Group (1991). *The Common Object Request Broker: Architecture and Specification*. OMG Document Number 91.12.1, Revision 1.1

Object Management Group (1992). *Object Management Architecture Guide*. OMG Document Number 92.11.1, Revision 2.0

Object Management Group (1995). *The Common Object Request Broker: Architecture and Specification*. Version 2.0, OMG Document

Orenstein J.A., Haradhvala S., Margulies B. and Sakahara D. (1992). Query processing in the ObjectStore database system. *In Proc. ACM SIGMOD International Conference on Management of Data*, pp. 403–12. New York: ACM

Orfali R., Harkey D. and Edwards J. (1996a). *The Essential Distributed Objects Survival Guide*. New York: Wiley

Orfali R., Harkey D. and Edwards J. (1996b). *The Essential Client/Server Survival Guide*, 2nd edn. New York: Wiley

Paredanes J., De Bra P., Gyssens M. and Van Gucht D. (1989). *The Structure of the Relational Database Model*. EATCS Monographs on Theoretical Computer Science Vol. 17. Berlin: Springer-Verlag

Penney D.J. and Stein J. (1987). Class modification in the GemStone object-oriented DBMS. In *OOPSLA '87 Proceedings*, pp. 111–17

Purdy A., Schuchardt B. and Maier D. (1987). Integrating an object server with other worlds. *ACM Transactions on Office Informations Systems*, **5**, 27–47

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. (1991). *Object Oriented Modelling and Design*. Englewood Cliffs, NJ: Prentice-Hall

Schek H.J. and Scholl M.H. (1986). The relational model with relation-valued attributes. *Information Systems*, **11**, 137–47

Schek H.J. and Scholl M.H. (1990). A relational object model. In *Proc. 3rd International Conference on Database Theory*, LNCS 470, pp. 89–105. Berlin: Springer-Verlag

Sciore E. (1994). Query abbreviation in the Entity–Relationship data model. *Information Systems*, **19**, 491–511

Sessions R. (1996). *Object Persistence – Beyond Object-Oriented Databases.* Upper Saddle River, NJ: Prentice-Hall PTR

Shaw G.M. and Zdonik S.B. (1990). An object-oriented query algebra. In *Database Programming Languages – The Second International Worshop* (Hull R., Morrison R. and Stemple D., eds.), pp. 103–12. San Francisco, CA: Morgan Kaufmann

Siegel J., ed. (1996). *CORBA Fundamentals and Programming.* New York: Wiley

Silberschatz A., Korth H.F. and Sudarshan S. (1997). *Database System Concepts*, 3rd edn. New York: McGraw-Hill

Simon A.R. (1995). *Strategic Database Technology: Management for the Year 2000.* San Francisco, CA: Morgan Kaufmann

Soloviev V. (1992). An overview of three commercial object-oriented database management systems. *ACM SIGMOD Record*, **21**(1), 93–104

Srivastava D., Ramakrishnan R., Seshadri P. and Sudarshan S. (1993). Coral++: adding object-orientation to a logic database language. In *Proc. 19th International Conference on Very Large Data Bases*, pp. 158–70. San Francisco, CA: Morgan Kaufmann.

Stonebraker M. (1996). *Object-Relational DBMSs – The Next Great Wave.* San Francisco, CA: Morgan Kaufmann.

Straube D.D. (1990). Queries and Query Processing in Object-Oriented Database Systems. *PhD Dissertation (Techn. Report TR90-33).* University of Alberta, Canada.

Tanenbaum A.S. (1995). *Distributed Operating Systems.* Englewood Cliffs, NJ: Prentice-Hall

Thomas S.J. and Fischer P.C. (1986). Nested relational structures. In *Advances in Computing Research* Vol. 3: *The Theory of Databases* (Kanellakis P.C. and Preparata F.P., eds), pp. 269–307. Greenwich, CT: JAI Press

Ullman J.D. (1988). *Principles of Database and Knowledge-Base Systems* Vol. I. Rockville, MD: Computer Science Press

Van den Bussche J. (1993). Formal Aspects of Object Identity in Database Manipulation. *PhD Dissertation*, University of Antwerp, Belgium

Van den Bussche J. and Van Gucht D. (1997). A semi-deterministic approach to object creation and non-determinism in database queries. *Journal of Computer and System Sciences*, **54**, 34–47

Van den Bussche J. and Vossen G. (1993). An extension of path expressions to simplify navigation in object-oriented queries. In *Proc. 3rd International Conference on Deductive and Object-Oriented Databases.* LNCS 760, pp. 267–82. Berlin: Springer-Verlag

Vandenberg S.L. and DeWitt D.J. (1991). Algebraic support for complex objects with arrays, identity, and inheritance. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 158–67. New York: ACM

Velez F., Bernard G. and Darnis V. (1989). The O_2 object manager: an overview. In *Proc. 15th International Conference on Very Large Data Bases*, pp. 357–66. San Francisco, CA: Morgan Kaufmann.

Vossen G. (1993). Bibliography on Object-Oriented Database Management (3rd and final edition). *Technical report no. 9301*, Computer Science Group, University of Giessen, Germany

Vossen G. (1994). *Data Models, Database Languages, and Database Management Systems* (in German), 2nd edn. Bonn: Addison-Wesley

Vossen G. (1996). Database theory: an introduction. In *Encyclopedia of Computer Science and Technology* (Kent A., Williams J. G., eds), Volume 34, Supplement 19, pp. 85–127. New York: Marcel Dekker, Inc.

Vossen G. (1997). The CORBA specification for cooperation in heterogeneous information systems. In *Proc. 1st International Workshop in Cooperative Information Agents*, LNCS 1202, pp. 101–15. Berlin: Springer-Verlag

Wallace E. and Wallnau K.C. (1996). A situated evaluation of the Object Management Group's (OMG) Object Management Architecture (OMA). In *Proc. OOPSLA '96 Conference*, ACM SIGPLAN Notices **31**(10), 168–78

Yasukawa H., Tsuda H. and Yokota K. (1992). Objects, properties, and modules in QUIXOTE. In *Proc. International Conference on Fifth Generation Computer Systems*, pp. 257–68. ICOT, Tokyo

Zaniolo C. (1983). The database language GEM. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 207–18. New York: ACM

Zdonik S.B. and Maier D., eds (1990). *Readings in Object-Oriented Database Systems.* San Francisco, CA: Morgan Kaufmann

Index

OBJECT-ORIENTED DATABASES

Models and Languages of Object-Oriented Databases

Georg Lausen | Gottfried Vossen

With the advent of a new generation of object-oriented (OO) databases, there have been significant advances in the theory underlying the technology. This book brings together a selection of key developments in OO databases from a number of different areas, including semantic modelling, formal data models, language design issues, object algebra and rule-based query languages. It shows how these elements may interact within an object-oriented *database system and how current commercial and* experimental systems fit into the picture.

Both the practical and theoretical angles of the subject are covered in detail by this book. A section containing numerous case studies of current systems explores the different ways in which an OO database may be constructed, and also covers the relevant standards. The subsequent section goes on to describe the more formal and theoretical aspects of the subject, discussing semantic and formal OO data models as well as algebraic and rule-*based approaches to language design. This unique* approach makes it ideal as a text for theory-based courses on OO databases, or as a state-of-the-art reference for database designers.

Models and Languages of Object-Oriented Databases includes case studies based on the following systems:

- GemStone
- O2
- ObjectStore
- *Illustra*

Readers should be familiar with the basic concepts of relational databases and their application, and with OO programming techniques.

About the authors:
Georg Lausen is Professor of Computer Science at the University of Freiburg and Gottfried Vossen is Professor of Computer Science at the University of Münster, both in Germany. They have extensive experience in teaching database courses to students and practitioners. Both authors carry out research on the *foundations, as well as the applications,* of object-based systems in general and object-oriented database concepts in particular.

ISBN 0-201-62431-1

90000>

9 780201 624311

ADDISON-WESLEY