Texts and Monographs in Computer Science

Object-Oriented Database Programming

Suad Alagić

13

Springer-Verlag
World Publishing Corp

Suad Alagić

Object-Oriented Database Programming

With 84 Illustrations

Springer-Verlag
World Publishing Corp

Suad Alagić
Department of Informatics
Faculty of Electrical Engineering
University of Sarajevo
71000 Sarajevo, Lukavica
Yugoslavia

Series Editor

David Gries
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853
USA

Library of Congress Cataloging-in-Publication Data
Alagić, Suad.
Object-oriented database programming/Suad Alagic
p. cm.—(Texts and monographs in computer science)
Bibliography: p.
Includes indexes.
1. Database management. 2. Object-oriented programming (Computer science) I. Title. II. Series.
QA76.9.D3A46 1988
005.74—dc19 88-13988

© 1989 by Springer-Verlag New York Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer-Verlag, 175 Fifth Avenue, New York, NY 10010, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use of general descriptive names, trade names, trademarks, etc., in this publication, even if the former are not especially identified, is not to be taken as a sign that such names, as understood by the Trade Marks and Merchandise Marks Act, may accordingly be used freely by anyone.

Reprinted by World Publishing Corporation, Beijing, 1992
for distribution and sale in The People's Republic of China only
ISBN 7-5062-1275-7

ISBN 0-387-96754-0 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-96754-0 Springer-Verlag Berlin Heidelberg New York

Preface

The major topic of this book is the integration of data and programming languages and the associated methodologies. To my knowledge, this is the first book on modern programming languages and programming methodology devoted entirely to database application environments. At the same time, it is written with the goal of reconciling the relational and object-oriented approaches to database management.

One of the reasons that influenced my decision to write this book is my dissatisfaction with the fact that the existing books on programming methodology and the associated concepts, techniques, and programming language notation are largely based on mathematical problems and mathematically oriented algorithms. As such, they give the impression that modern program structures, associated techniques, and methodologies, not to speak of the formal ones, are applicable only to problems of that sort. Although important, such problems are of limited applicability and scale. This does not apply to books in which modern concepts, techniques, methodologies, and programming language notation are applied to systems programming. But, even so, this does not demonstrate that in entirely application-oriented problems—those in which modern computer technology is most widely used—modern programming methodology is just as important.

This book is meant to be a step toward providing a more convincing support of such a claim and, thus, is based entirely on common, what one might call business-oriented, problems in which database technology has been successfully used. However, as far as I know, that usage has little to do with what in my opinion belongs to modern trends in programming languages and programming methodology. From the research point of

view, the integration of data and programming languages, and the associated methodologies, is certainly the major goal of this book. But, at the same time, its goal is also to demonstrate that such an integration has very important large-scale implications.

Another important reason for my decision to write this book is the situation in the database area where the proliferation of high-level, nonprocedural languages (such as relational languages) and other nonprocedural tools has established another wrong impression in which again, although for a different reason, the results in the development of modern programming language concepts, notation, and the associated methodologies do not seem to matter very much since so many things can be done nonprocedurally anyhow. Application programmers are offered interfaces of modern, high-level, nonprocedural languages and quite often old-fashioned programming-level tools, where the interface of the two is hardly (if ever) conceptually and notationally attractive.

Although modern nonprocedural database tools allow attractive solutions to problems of increasing complexity, this book demonstrates that there are many of them that require a good solution on the programming language level. Furthermore, some common application-oriented problems require the full power of integrated data and programming languages that comes from concepts like relational model and recursive procedures, exception handling, and attractive screen management—of course, with the condition that we apply the established programming methodology criteria to procedural solutions of such problems having to do with elegant, modular structure, correctness, adaptability, efficiency, etc.

This does not mean that I am not in favor of nonprocedural tools. In fact, we have carried out the development of such tools associated and tied in with the programming language notation used in this book very far, in order to implement a complete database programming environment. The existence of such nonprocedural tools influenced the decisions about the required features of the database programming language used in this book considerably.

As far as the notation is concerned, the database programming language used in this book has been derived from Modula-2 and is thus called MODULEX. It differs considerably from other proposed database extensions of languages like Pascal and Modula, as well as from other proposed database programming languages. Whereas the previous proposed extensions of Pascal and Modula, such as Pascal/R and Modula/R, are relational oriented, MODULEX is object-action oriented. It supports the relational model as a particular case, and explores the concepts of the definition and the implementation modules as defined in Modula-2 to obtain a simple and powerful object-oriented extension of that model.

In spite of the fact that all the power of the relational model and the associated languages is offered in MODULEX, as well as its object-oriented extension in terms of modules, the programming language is kept

此为试读,需要完整PDF请访问: www.ertongbook.com

small and simple, just as Modula-2 is. The adopted extensions are kept to those that are absolutely necessary, and consequently, the book should be simple for readers and programmers with some familiarity with languages like Pascal and Modula. This approach to the programming language design makes MODULEX quite different from other proposed database programming languages. Indeed, rather than adding a number of new and fancy features that such languages have, MODULEX explores the power of the existing programming language concepts. The concept of a module as defined in Modula-2 plays a crucial role in this respect, and its appropriate interpretation leads to a particularly simple object-oriented extension of the relational model.

Representation of objects as modules is one of the major original contributions of this book. MODULEX is essentially designed by adding the entity set type and the associated action-composition rules (such as foreach) to the standard collection of programming language abstractions. These extensions are defined in such a way that they offer the power of the relational model and the relational languages in the programming language framework. Imposing modules on such a unified relational and programming environment produces the desired object-action-oriented environment.

In view of the above described approach to the object-oriented database programming language and its environment, the programming methodology presented in this book is, of course, object oriented or abstract data-type oriented. By this we mean that the cornerstone of our approach are object types whose properties and actions applicable to instances of those types are encapsulated in the definition modules of those objects. In comparison with the relational approach, users of those objects are still permitted (or may be allowed if appropriate) to state arbitrarily complex queries upon such objects, but other applicable actions are restricted to those specified in the definition modules of those object types. The actual representation (implementation) of such objects and their associated actions is separated from their definition and from their users as well. The object types in typical database application environments quite often have relational representation.

Conceptual modeling of application environments presented in this book is based on a collection of standard abstractions: aggregation, generalization, covering, partitioning, and recursion. They all have simple algebraic interpretation, as do the usual data abstractions in Pascal-like languages. At the same time, the relationship with the associated action-composition rules in such languages is easily established. The relational representation of the chosen collection of standard abstractions enriched with those rules and the structure of modules has been developed carefully in this book in an object-oriented style.

The graphical representation is a departure from what is customary in databases, and is based on the algebraic interpretation of standard ab-

stractions; so, an arrow simply denotes a function—what could be simpler and more consistent than that?

There is no question that the chosen framework of a single, small, and simple database programming language has its limitations, in terms of presentation of some important features of modern database systems. Although many typical integrity constraints are specified nonprocedurally in the MODULEX screen language, particularly those related to the relational representation of standard abstractions, the programming-language-based presentation in this book demonstrates how they are enforced procedurally.

This explains the role of the relational model in our approach. In fact, one of our major goals was to reconcile the relational and object-oriented approaches, and I have just given a brief and very general description of how this is achieved. Thus, in spite of the object-oriented methodology applied in this book, the presented material is to a large extent also relational-oriented programming methodology.

Perhaps it is a pity that this book is not more formal (i.e., mathematical) since virtually all the presented material has nice and sound mathematical foundation. This applies to standard abstractions, the relational model, action-composition rules, modules, and formal verification techniques for actions. Only the very basic mathematical concepts underlying the presented material in its entirety are given, in an informal manner, and I must admit that the reason for this is my desire to reach a wider audience, who may be uninterested in the underlying mathematical concepts. Therefore, the underlying concepts are used only when necessary in order to make the definitions more precise and clear. The basic formal theory of the relational model is also presented under these same limitations.

I hope this book presents a new way of teaching about databases. Indeed, introductory programming language courses based on languages like Pascal and Modula-2, and the familiarity with the very basic notions of modern mathematics are all that is required to use this book. It is my hope that readers with this background will find this book appealing, since it is entirely based on the programming language notation. I also hope this book will present some of the most important conceptual database problems to people whose work is in programming languages and programming methodology, in a manner that is much more appealing to them than the way databases are usually taught. The research on integration of data and programming languages, and the associated systems, has certainly a lot to offer to both database and programming language areas. It opens up a large number of new research problems whose solutions have practical large-scale implications.

Perhaps the exercises are the best part of this book. They expand the material in the main body of the text with a number of more advanced concepts, techniques, and methodologies. Some of the most attractive and rewarding examples and problems are given in the exercises. Working them out is the only way to master the underlying methodology presented

in this book and enables one to make quite a number of discoveries on one's own. The presented selection is large enough so that lecturers teaching a course on this subject can make the appropriate choice of assignments to their students.

This book has been written alongside the development of the MODULEX database programming environment, which, in addition to the compiler of the database programming language, has as its major feature the associated high-level, object-oriented, definition, query, manipulation, and control screen language. The MODULEX screen language is designed in such a way that it fits in the unified object-oriented framework, based on the database programming language presented in this book. In addition to object-oriented and relational-oriented queries, it supports conceptual design in the sense that it makes the automatic generation of the definition and the implementation modules of object types possible, together with the associated actions from the specifications given in the screen language. At the completion of this book, the MODULEX database programming environment has already been running on the VAX, and its reimplementation on the personal computer is also planned. Readers interested in the described software tool should contact the publisher (or the author) of this book for further details.

The material presented in this book, as well as the MODULEX software environment, grew out of my research project "Extended Relational Database Programming Environment," which was support by the National Science Foundation under grant JFP/708. I would like to express my sincere appreciation for the work done by my research assistants Miroslav Kandić and Dragan Jurković. This applies not only to their contributions to the implementation of the MODULEX system, but also to their extremely valuable help in reviewing and editing the final version of the manuscript.

Sarajevo, Yugoslavia

SUAD ALAGIĆ

(Continued)

Texts and Monographs in Computer Science

Robert N. Moll, Michael A. Arbib, and A. J. Kfoury
An Introduction to Formal Language Theory

Franco P. Preparata and Michael Ian Shamos
Computational Geometry: An Introduction

Brian Randell, Ed.
The Origins of Digital Computers: Selected Papers

Arto Salomaa and Matti Soittola
Automata-Theoretic Aspects of Formal Power Series

Jeffrey R. Sampson
Adaptive Information Processing: An Introductory Survey

William M. Waite and Gerhard Goos
Compiler Construction

Niklaus Wirth
Programming in Modula-2

Contents

此为试读,需要完整PDF请访问: www.ertongbook.com

Introduction

1 Objects

In order to satisfy users' needs, database technology requires the design of a suitable representation of its application's environment. This representation is called a *conceptual model*. A conceptual model of an application environment is thus an abstract representation of that environment that contains only those abstract properties of the environment relevant for the information requirements of its users.

Our approach to the design of a conceptual model of an application environment is based on a collection of abstraction techniques. The most fundamental technique is classification of relevant objects in an application environment into a collection of object types. A set of objects is regarded as an object type if all the objects in that set share the same set of relevant properties (*attributes*). The relevance of a property of an object is, of course, determined by the purpose of the model. A particular object then becomes an instance of an object type. For example, in a university application environment, object types would be STUDENT, PROFESSOR, COURSE, DEPARTMENT, etc.

In order to apply the classification abstraction, we have to specify precisely the properties shared by a set of objects that belong to the same type. For example, such a specification for the object type STUDENT might look like this:

Object type: STUDENT

Attributes: Student identifier
Name

Level (undergraduate, graduate)
Address
Phone

An attribute of an object assumes particular values that also belong to a *type*, that is, to a set of objects that share the same set of properties. For example, the type of the attribute Name is a set of sequences of characters, and the type of the attribute Level is a two-element set.

It follows then that a new object type is defined in terms of other, already defined object types that become attributes of the new object type. We say that an object is an *aggregation* of its attributes. Aggregation is in fact another fundamental abstraction in the design of conceptual models of application environments. Its importance comes from the fact that an object type is defined as an aggregation of its attributes. In general, given object types E1,E2,...,En, we define their aggregate object type E in such a way that E1,E2,...,En become components (attributes in a particular case) of the object type E.

To clarify, we will use the programming language notation. The fact that an object type E has object types E1,E2,...,En as its components is expressed in terms of a record type. Such a type definition for the object type STUDENT has the following form:

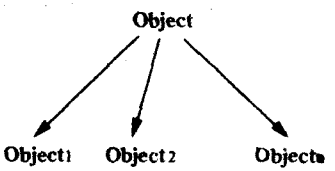
```
TYPE Student = RECORD Student#: String7;
                      Name:     String30;
                      Level:    (undergraduate,graduate);
                      Address:  String35;
                      Phone:    String7
               END
```

In general, we have

```
TYPE Object = RECORD Attribute1: Object1;
                     Attribute2: Object2;
                     .
                     .
                     .
                     Attributen: Objectn
              END
```

where Object1, Object2,..., Objectn and Object are object types.

The graphical notation used throughout this book is based on the mathematical interpretation of aggregation and has the following form:



The mathematical interpretation is in fact very simple. The set Object is the Cartesian product of sets Object1, Object2,..., Objectn; that is,

$$\text{Object} = \text{Object1} \times \text{Object2} \times \cdots \times \text{Objectn}.$$

Indeed, an instance of the type Object is a tuple (a1,a2,...,an) of values of attributes of that instance:

$$\text{Object} = \{(a1,a2,\ldots,an) \mid ai \text{ in Objecti for } i=1,2,\ldots,n\}.$$

Given an instance o of the type Object, its components are denoted in the programming language notation as o.Attribute1, o.Attribute2, ...,o.Attributen. So, for example, if s is of type Student, its attributes are denoted as

s.Student#, s.Name, s.Level, s.Address, s.Phone.

Selection of a component of an aggregate object (attribute of an object) is also called *projection* and is of fundamental importance for the aggregation abstraction. In fact, we can interpret Attributei as a function (projection)

$$\text{Attributei: Object} \rightarrow \text{Objecti}$$

defined as

$$(a1,a2,\ldots,an) \rightarrow ai.$$

The relational model of data is largely based on the above simple observations. In order to represent an application environment, it requires only classification and aggregation abstractions, where the latter is used only in a particular form where all the components of an object are simple. An instance of an object type is represented as a tuple (a1,a2,...,an) of values of its attributes, so that a particular set of objects of the same type is represented as a set of such tuples. But such a set is formally a relation. For example,

RegisteredStudents = StudentIdType ×
NameType ×
LevelType ×
AddressType ×
PhoneType

is a set (a relation) of registered students, where the type of that set is StudentSet defined in our programming language notation as

TYPE StudentSet = ENTITY SET OF Student.

The above genuinely simple and natural representation implies the following requirement: Given representations o1 and o2 of two distinct objects of the same type Object, we must have

$$o1.\text{Attributei} <> o2.\text{Attributei} \quad \text{for some } i=1,2,\ldots,n.$$

In other words, two distinct objects of the same type should be represented by two distinct tuples of values of their attributes. In order to guarantee this condition, it is sufficient if the two tuples have different values of at least one attribute.

The above reasoning makes obvious the importance of an attribute (or a set of attributes, in general) whose value determines a unique object of a given type. Such an attribute or a set of attributes is called an *object identifier*, or a *key*. In a relation that represents a particular set of objects of the same type, there is thus at most one tuple with a given key value. The requirement that each object type in a conceptual model of an application environment must have at least one key hardly needs any further justification. We simply need to know what actual object of an application environment each tuple of a relation represents.

2 Actions

A conceptual model of an application environment is meant to satisfy information needs of users of that environment via queries stated against that model. An example of a query would be

Print the names of all registered graduate students.

This query would be expressed in the programming language notation used in this book as

```
FOREACH s IN RegisteredStudents
WHERE   s.Level = graduate
DO      WriteString (s.Name); WriteLn
END
```

where s is of type Student and RegisteredStudents is of type StudentSet.

Set-oriented queries are typical for the relational model of data. In fact, in addition to the representation of sets of objects as relations, the relational model also offers a collection of general-purpose operators upon such a representation that act upon that representation in order to produce results that satisfy specific information needs of end users. Projection is one such operator, and the other involved in the above query is restriction—selection of a subset of a set of objects where each object in the subset satisfies a given condition. The result of the above query is expressed in the usual mathematical notation in terms of restriction and projection as follows:

$$\{s.Name \mid s \text{ in } RegisteredStudents \text{ and } s.Level = graduate\}.$$

A model of an application environment is of course useless unless we can specify actions against the model that would produce some useful

effects. This trivial observation indicates that a representation of an application environment in terms of classification and aggregation of types of objects in that environment is an extremely simplified view of the environment that has to be enriched by actions that may be performed on the represented objects.

Actions associated with an object type in fact correspond to activities in the application environment involving objects of that type. This means our approach to conceptual modeling presented so far has to be refined in a crucial way. An application environment is still viewed as a collection of object types, but the essential part of specification of such a type is a set of actions upon instances of that type that correspond to the actual activities involving those objects in the application environment. For example, conceptual representation of the object type STUDENT would now be completed in the following manner:

Object type: STUDENT

Attributes: Student identifier
Name
Level
Address
Phone

Actions: Enroll student
Drop student
Change level
Change address
Change phone

The refined object-action approach to conceptual modeling of application environments requires specification of a set of actions associated with objects of the same type as the most important specification of that type. If we carry out this approach to all its consequences, then an object type may be defined entirely in terms of actions that may be performed upon instances of that type. Indeed, we already observed that attributes of an object type in fact correspond to projections. If we accompany a set of actions associated with an object type with a set of projections (e.g., SelectStudentId, SelectName, SelectLevel, SelectAddress, SelectPhone) to the attributes of that type, we obtain an entirely action-oriented definition of an object type.

Although the action-oriented definition of an object type has a sound mathematical basis, it is not unusual for programmers to deal with an object entirely in terms of actions that may be performed upon that object. In fact, the whole point, is to forbid explicit manipulation of representation of an object by the clients of that object and force them to use a predefined set of actions given in the definition of that object type that are guaranteed to manipulate the actual representation correctly.

此为试读,需要完整PDF请访问: www.ertongbook.com