Object-Oriented Oracle

Wenny Rahayu
David Taniar
Eric Pardede

Object-Oriented Oracle™

Johanna Wenny Rahayu
La Trobe University, Australia

David Taniar
Monash University, Australia

Eric Pardede
La Trobe University, Australia

IRM Press

Publisher of innovative scholarly and professional information technology titles in the cyberage

Hershey • London • Melbourne • Singapore

Acquisitions Editor:	Renée Davies
Development Editor:	Kristin Roth
Senior Managing Editor:	Amanda Appicello
Managing Editor:	Jennifer Neidig
Copy Editor:	Shanelle Ramelb
Typesetter:	Cindy Consonery
Cover Design:	Lisa Tosheff
Printed at:	Yurchak Printing Inc.

Published in the United States of America by
IRM Press (an imprint of Idea Group Inc.)
701 E. Chocolate Avenue, Suite 200
Hershey PA 17033-1240
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@idea-group.com
Web site: http://www.irm-press.com

and in the United Kingdom by
IRM Press (an imprint of Idea Group Inc.)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 3313
Web site: http://www.eurospan.co.uk

Library of Congress Cataloging-in-Publication Data

Object-oriented Oracle / Wenny Rahayu, David Taniar and Eric Pardede, editors.
p. cm.
Summary: " The book covers comprehensive and fundamental aspects of the implementation of object-oriented modeling in a DBMS that was originated as a pure Relational Database, Oracle"--Provided by publisher.
Includes bibliographical references and index.
ISBN 1-59140-810-5 (hardcover : alk. paper) -- ISBN 1-59140-607-2 (softcover : alk. paper) -- ISBN 1-59140-608-0 (ebook : alk. paper)
1. Oracle (Computer file) 2. Object-oriented methods (Computer science) I. Rahayu, Wenny, 1968- II. Taniar, David. III. Pardede, Eric, 1975-
QA76.9.D26O23 2005
005.1'1--dc22
2005005340

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

All work contributed to this book is new, previously-unpublished material. The views expressed in this book are those of the authors, but not necessarily of the publisher.

Object-Oriented Oracle™

Table of Contents

Preface

Why This Book?

Object orientation has now invaded traditional relational database-management systems. Oracle™ without exception has included object-oriented features in its system. SQL is now richer due to these additional features. However, the object-oriented elements in Oracle™ will not be fully utilized without a proper database design. For example, a database application designed using a traditional database modeling, such as entity-relationship (E/R) modeling, will not be able to make use of most object-oriented features in Oracle™. This is simply due to the absence of object-oriented elements in the design. Even with a proper object-oriented design, without careful transformation from design to implementation, many of the object-oriented features will be lost.

Object-Oriented Oracle™ addresses this need by not only explaining the new object-oriented features in Oracle™, but most importantly how these features can be fully utilized in database applications. We put a heavy emphasize on how an object-oriented conceptual model is implemented in Oracle™. This includes the static aspect of an object-oriented conceptual model, including the inheritance, association, and aggregation relationships, as well as the dynamic aspect covering generic object-oriented methods and user-defined queries.

Just as we enjoyed writing this book, we hope that you will enjoy reading it, and most importantly gain valuable lessons from it. We trust that this book will give you a comprehensive insight into object-oriented Oracle™.

Distinguishing Features

Object-Oriented Oracle™ presents the right mix between theoretical and practical lessons on object-oriented features of Oracle™.

In the theoretical part, it describes the foundation of object-oriented concepts and how they are used in the implementation. The importance of these concepts is invaluable because without this understanding, the new object-oriented features offered by Oracle™ will not be fully utilized. Therefore, these theoretical elements serve as the foundation of object orientation in Oracle™.

In the practical part, the book contains two case studies (Chapters VII and VIII) that thoroughly explain the development of a database application using the object-oriented technology of Oracle™. The case studies start with the description of an application, followed by the appropriate object-oriented designs. The designs are then transformed for implementation in Oracle™.

Each chapter also contains extensive examples and code. These examples and code will give readers a better understanding of how object-oriented elements are used in Oracle™.

At the end of each chapter, a set of problems, together with their solutions, are given. These will be suitable exercises for the classroom. The solutions will be useful for both students and their teachers.

Topical Coverage

Object-Oriented Oracle™ contains eight chapters.

Chapter I starts with object-relational approaches that cover the object-oriented conceptual model. There have been many approaches in amalgamating the object-oriented model with database systems, from which the new era of object-relational databases is born.

Chapter II explains object-oriented features in Oracle™. These include the use of *type* and *object* in conjunction with table creation, *varray*, and *nested table*. These features, together with the *ref* relationships, index cluster, and the *under* clause for subtyping, change the whole concept of database modeling.

Chapter III describes how these object-oriented features should be properly used in Oracle™. This includes how the object-oriented conceptual model described in Chapter I is implemented using the features presented in Chapter

II. This chapter particularly focuses on the static aspect of the object-oriented conceptual model, including the inheritance, association, and aggregation relationships.

Chapter IV justifies how the dynamic aspect of the object-oriented conceptual model (encapsulation and object-oriented methods) is implemented using the new features of Oracle™, namely member procedures and functions.

Chapter V describes generic methods in Oracle™. This covers generic methods found in the object-oriented conceptual model, including the inheritance, association, and aggregation relationships. The generic methods comprise typical database operations (e.g., update, delete, and insert) applied to the member attributes of a class. The use of generic methods is a direct implementation of object-oriented encapsulation features.

Chapter VI focuses on user-defined queries. New SQL features, covering referencing and dereferencing using *ref*, super- and subclass accesses using *treat*, nesting techniques using *the* and *table*, are explained. The chapter also discusses the *varray* and nested-table collection types, object references *deref*, the *is dangling* clause, and object attributes.

Chapter VII introduces a university case study that contains a database to maintain the running of courses in a university. This case study shows the entire database-application development life-cycle process from the object-oriented design to transformation for implementation in Oracle™.

Finally, Chapter VIII presents another case study based on a retailer-chain company. In addition to using the object-oriented conceptual model for the database design, implementation is carried out using Oracle™ Form Developer. The aim is to show how a window-based database application can be developed using the object-oriented technology in Oracle™.

Intended Audience

Object-Oriented Oracle™ is intended for the following audiences.

- **Database Practitioners**

  Object orientation in Oracle™ has now opened a wide opportunity in exploring new ways for building database applications. This book shows how object-oriented features can be adapted for database-application development. It describes not only the practical aspects of database-application development, but also the theoretical foundations that lead to the use of the object-oriented technology in database applications using Oracle™. The two case studies included in this book show the two flavours of database applications using the object-oriented technology as their foundation whereby the first application is a text-based application, and the second is window-based using Oracle™ Form Developer.

- **College Students and Teachers**

  This book is suitable as a textbook for database courses at any level: an introductory database course whereby this book can be used as a supplement to the standard database-management textbook, or an advanced database course concentrating on object-oriented database development. Students who are learning the standard material of SQL are now able to learn, at the same time, the new object-oriented features of SQL. Furthermore, students are now able to relate how a database design, in this case using an object-oriented method, can smoothly be implemented in Oracle™, thus making the entire database-application-development life cycle transparent.

- **General IT Readers**

  General IT readers who are keen on the new technology of Oracle™ will find this book useful and informative. Object orientation has been an interesting topic in general due to the popularity of object-oriented programming languages, like C++ and Java. The object-oriented concepts, which underpin these programming languages, have been widely understood. However, their applications to database systems have not been broadly explored. This book demonstrates how object-oriented features could be used easily in Oracle™, and most of all, how they could be used appropriately and efficiently.

- **IT Researchers**

  Object orientation in relational database systems has been an active research area in the last decade. Many researchers have proposed methods for transforming object-oriented design to relational database implementation. Other groups of researchers have been concentrating on object-relational databases. Due to the increasing trend whereby most database-management-system vendors are positioning themselves in the object-oriented tracks, there are plenty of research opportunities in this important area. This book will give researchers the basic foundation for amalgamating two different elements: object-oriented and relational database systems.

Feedback and Comments

Although we have fully tested all code included in this book, should there be any problems or confusion about the code, please do not hesitate to contact us.

We would appreciate if you could also share any other comments or feedback with us so that we can incorporate them in a future edition. Comments and feedback may be sent directly to the publisher at

Object-Oriented Oracle™
Idea Group Inc.
701 East Chocolate Avenue, Suite 200
Hershey, PA 17033-1240, USA

Acknowledgments

Object-Oriented Oracle™ would not have been published without the support of a number of parties. We owe them our gratitude.

First of all, we would like to thank Mehdi Khosrow-Pour and Jan Travers of Idea Group Publishing for believing in us on this project. They supported our ideas in writing a book on this topic, and only because of their encouragement and trust, this book becomes a realization.

We would also like to thank the team at Idea Group for keeping the schedule on track. Their communication and support were very efficient and professional. We were glad for this opportunity to collaborate with them.

Finally, we would like to express our sincere thanks to our respective employers, the Department of Computer Science and Computer Engineering, La Trobe University, Australia, and the School of Business Systems, Monash University, Australia, for the facilities and time that we received during the writing of this book. Without these, the book would not have been written in the first place.

J. W. Rahayu

D. Taniar

E. Pardede

Melbourne, June 20, 2005

Chapter I

Object-Relational Approaches

This book focuses on the implementation of an object-oriented model into object-relational DBMS using Oracle™. All aspects of the object-oriented model, particularly those that play a significant role in database implementation, will be discussed in this book.

The object-oriented modeling technique is an important issue in this book because it is the underlying notion behind the development of the object-relational approaches. Therefore, in this chapter we will start with an outline of the object-oriented conceptual model (OOCM).

Object-Oriented Conceptual Model

An OOCM encapsulates the structural and static as well as behavioral and dynamic aspects of objects. The static aspects consist of the classes and objects, and the relationships between them, namely, inheritance, association, and aggregation. Each of these relationships is associated with a set of constraints. The dynamic aspect of the OOCM is divided into two types of methods: generic and user defined.

The object-oriented method promised to improve software quality and efficiency. One of the most enticing promises is that of real reusability: reusability of codes, program portions, designs, formal specifications, and also commercial packages. As software-development cost increases, more developers see the benefit of using reusable components. Solving the reusability problem essentially means reducing the effort required to write codes; hence, more effort can be devoted to improving other factors such as correctness and robustness.

The main idea of the object-oriented method is that it provides a more natural way to model many real-world situations. The model obtained by the object-oriented method will be a more direct representation of the situations, providing a better framework for understanding and manipulating the complex relationships that may exist.

The basic segment of the object-oriented system is an *object*. Everything that exists and is distinguishable is an object. Each object has one or more unique attributes that make it distinguishable from the others.

However, several objects can also have the same structure of attributes and operations. Only after the attributes' values are given can an object be recognized. A set of attribute structures and operations applicable to those attributes is called a *class*.

In the object-oriented method, we also recognize the concept of *encapsulation*. Basically, from an outside point of view, each object is just a thing or a person (such as a student named Jennie, Andy, etc.). However, if each object is explored in greater detail, it actually consists of some attributes (identity, name, status, gender, etc.) for which each object has its own value and so is distinguishable, as are the operations that are applicable to those sets of data (print details, set details, etc.). In other words, an object is simply an encapsulation of data and their operations.

Static Aspects of OOCM

The static aspects of OOCM involve the creation of the objects and classes that also includes decisions regarding their attributes. In addition, the static aspects of OOCM are also concerned with the relationship between objects, that is, inheritance, association, and aggregation.

Objects and Classes

An object can be a physical object, such as a computer, vehicle, or person. It can be an event such as a queue in front of the teller, sales, and so forth. People's roles such as that of an officer, tutor, student, and so forth can also be classified as objects.

An object is a data abstraction that is defined by an *object name* as a unique identifier, valued *attributes* (instance variables) that give a *state* to the object, and *methods* or routines that access the state of the object. It is convenient to use a graphical notation to represent an object model. We will use a notation that is a modified UML notation (Booch, Rumbaugh, & Jacobson, 1999). The modifications will be clarified throughout this discussion. Most of these relate to the semantics and definitions of some terms such as composition, aggregation, and so forth. An object is often drawn as a rectangle having an object name and its properties (attributes and methods). With far fewer details, an object is often shown as a square with the object name only. Figure 1.1 gives an illustration of a graphical notation for objects.

The state of an object is actually a set of values of its attributes. The specified methods are the only operations that can be carried out on the attributes in the object. The client of the object cannot change the state except by method invocation. Thus, an object encapsulates both state and operations. In some languages, the methods are procedures and functions. A procedure may or may not have arguments, and it can be used to access the attributes of an object. A function is similar to a procedure, but it returns a value.

Objects are the basic run-time entities in an object-oriented system. An object can be created only during run time. Figure 1.2 shows an example where at run time an object Staff with name Adam is a staff member in the computer-science department.

Figure 1.1. Object

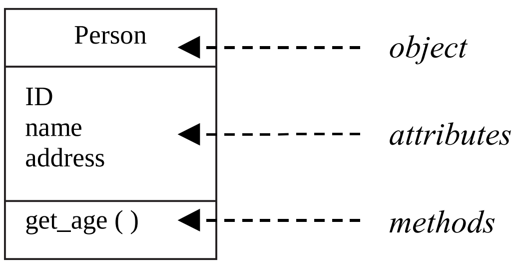
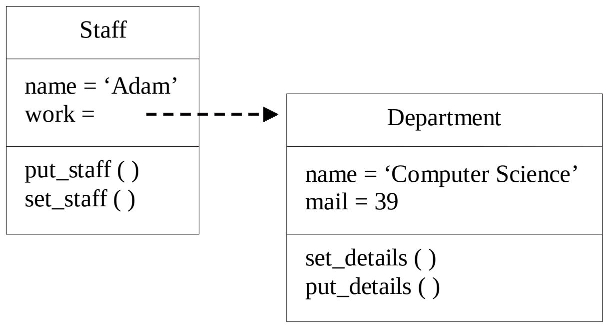


Figure 1.2. Object as run-time entity



Each object has an identity, called *object identity* (OID). An OID is an invariant property of an object that distinguishes it logically and physically from all other objects. An OID is therefore unique. Two objects can be equal without being identical.

Along with objects, we also need to understand classes. It is important to distinguish between them, and they should not be confused.

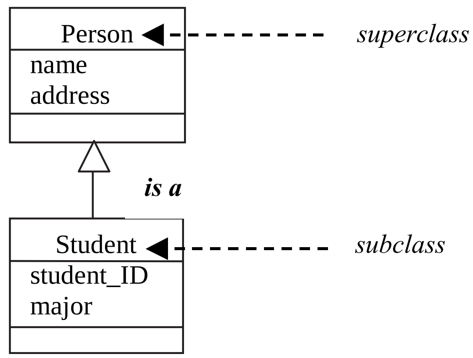
A class is a description of several objects that have similar characteristics (Dillon & Tan, 1993). Coad and Yourdon (1990) described class as a set of specifications that characterizes and is applicable to a collection of objects. Objects of the same class have common methods and, therefore, uniform behavior. Class is a compile-time notion, whereas objects exist only at run time. Therefore, a class has three aspects: the *type* as attributes and applicable routines, a *container* of objects of the same type, and an *instantiation mechanism*, such as to create.

Inheritance Relationships

An *inheritance* relationship is generally known as a generalization or specialization relationship, in which the definition of a class can be based on other existing classes. Given that a class inherits from another class, the former class is known as a *subclass*, whereas the latter is the *superclass*.

A subclass is a class that inherits from at least one generalized class that is the superclass. Consequently, a subclass must have all the properties of the superclass, and may have others as well. In other words, a subclass is more specialized than the superclass. Inheritance is a key feature of the object-oriented paradigm.

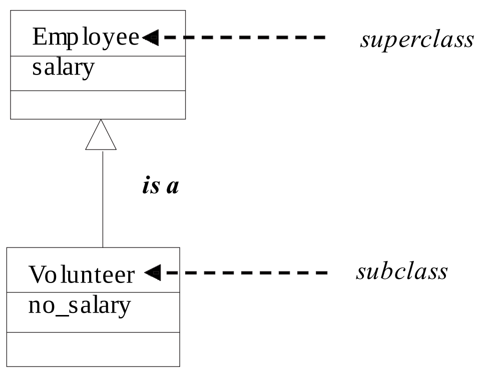
Figure 1.3. Inheritance relationship as an extension



Consider Figure 1.3 as an example. Suppose there are two classes: Person and Student. In this case, every student must be a person, so Student class inherits from Person class. All features that apply to a person are applicable to a student, and every student is a person. A student will also have a name and an address from Person class. Moreover, a student can have additional features. Therefore, the inheritance mechanism can be viewed as an extension of a superclass.

On the other hand, rather than being considered as an extension, inheritance can be viewed as a restriction on the superclass by hiding previously exported features of the superclass. Figure 1.4 shows an example of using inheritance as a restriction. Beside features such as name, address, and so forth, Employee class has an attribute salary, whereas Volunteer class, which is a special case of employee, does not receive any salary.

Figure 1.4. Inheritance relationship as a restriction



If several classes have considerable commonality, it can be factored out in a deferred or abstract class. The differences are provided in several subclasses of the deferred class. A deferred class provides only a partial implementation of a class or no implementation at all. From the design point of view, a deferred class provides the global view of a class, although the details have not yet been implemented.

Association Relationships

Association refers to a connection between object instances. Association is basically a reference from one object to another that provides access paths among objects in a system.

Objects are connected through an association link. The link can have a specified cardinality, such as one-to-one, one-to-many, and many-to-many. In addition to this, in object orientation, collection types have also been introduced and can characterize an association link.

One-to-One Association

In this type, only one object can be connected with another object of the other type for the particular association link, and vice versa.

For example, in Figure 1.5, Staff class and Office class are connected through a *work_in* association link. The link is one-to-one type because only one staff can work in one office, and one office can have only one staff working in it.

One-to-Many Association

In this type, the first object can be connected only with one of the second object, but the second object can connect with many of the first object.

Figure 1.5. One-to-one association

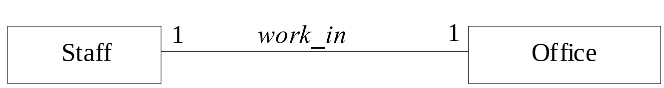


Figure 1.6. One-to-many association

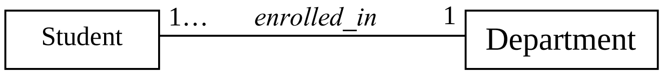


Figure 1.7. Many-to-many association

For example, in Figure 1.6, Student class and Department class are connected through an *enrolled_in* association link. The link is one-to-many type because one student can enroll only in one department, but one department can have many students enrolled in it.

Many-to-Many Association

In this type, one object can be connected with many objects of the other type for the particular association link, and vice versa.

For example, in Figure 1.7, Student class and Subject class are connected through a *takes* association link. The link is a many-to-many type because one student can take many subjects, and one subject can be taken by many students.

Aggregation Hierarchies

Aggregation is a tightly coupled form of association (Rumbaugh, Blaha, Premerlani, Eddy, & Lorensen, 1991). The main difference between aggregation and association is the underlying semantic strength. While an aggregation forms a method of organization that exactly maps human thinking, an association is a mere mapping between objects in an application (Coad & Yourdon, 1991).

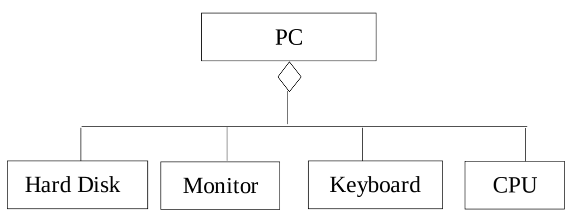
Aggregation is a composition or "part-of" relationship, in which a composite object (whole) consists of other component objects (parts). This relationship is used extensively in the areas of engineering, manufacturing, and graphics design. In these applications, when a composite object is created, one may merely want to know the type of the parts involved without being bothered with the details. At other times, one may need the details of a particular part only (Dillon & Tan, 1993).

In an aggregation relationship, in which one whole can have many parts associated with it through a part-of relationship, the entire part-of relationship is viewed as one composition, not several association relationships. Let us consider an aggregation relationship between a PC (personal computer) as a whole and its parts consisting of the hard disk, monitor, keyboard, and CPU (Figure 1.8). It would be inappropriate to model the aggregation as an association since the composition semantic would be lost in the association. Modeling the above example as an association will form several association relations, namely, the PC and hard disk, PC and monitor, PC and keyboard, and PC and CPU. Instead of creating one composition, we will end up with several associations.

Because the relationship between the whole and the parts is very clearly designated in aggregation relationships, we should be able to retrieve all aggregate parts that belong to a whole by identifying the whole only. For example, when a PC object is accessed, the aggregate parts Hard Disk, Monitor, Keyboard, and CPU that belong to that PC can also be identified. Implementing the above aggregation as an association will require us to go through every association relationship in order to retrieve all parts that belong to a whole.

Dillon and Tan (1993), Dittrich (1989), and Kim (1990) identify four types of composition: sharable dependent, sharable independent, nonsharable dependent, and nonsharable independent. We will refer to nonsharable and sharable as *exclusive composition* and *nonexclusive composition*, and dependent and independent as *existence-dependent* and *existence-independent composition*, respectively.

Figure 1.8. Aggregation



Existence-Dependent and Existence-Independent Composition

When the existence of the part object is fully dependent on the whole object, then the aggregation relationship is of an existence-dependent type. In this type of aggregation, whenever the whole object is removed, then all its associated part objects will also be removed. Thus, no part object can exist without an associated whole object. This is the most common type of aggregation, where the whole object is more like a container object. When the existence of a part object is independent of any whole object, we will have an existence-independent aggregation.

Existence-dependent and existence-independent compositions are two aggregation types in which the dependencies between the whole object and its part objects are significant.

Figure 1.9 shows an example of an existence-dependent composition. In the example, a Course Outline object is an encapsulation of several part objects, that is, Course Objectives, Course Contents, and Course Schedule. When a whole object is accessed, its part objects can be identified without the necessity to trace every link from the Course Outline object. In an existence-dependent type of composition, the deletion of a course outline will cause the deletion of that particular course outline and all of its elements.

In an existence-independent type of composition, the existence of the part is independent. For example, in Figure 1.10, if for some reason Travel Documents is removed, the ticket, itinerary, and passport still exist.

Figure 1.9. Existence-dependent composition

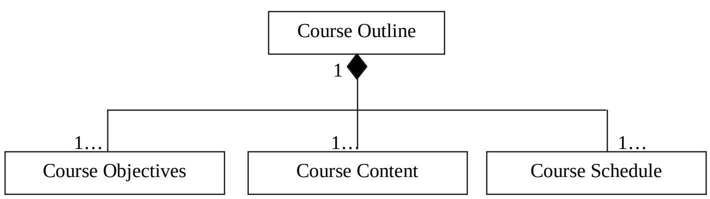
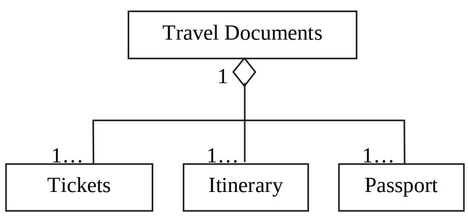


Figure 1.10. Existence-independent composition



Exclusive and Nonexclusive Composition

When in an aggregation relationship a particular part object can be shared by more than one whole object, then we have a nonexclusive type. Otherwise, when each part object is exclusive to a particular whole only, then it is an exclusive type of aggregation.

Creating an exclusive composition means that the whole object is the sole owner of the part objects. The need for exclusiveness arises particularly when modeling physical objects, such as vehicles, bridges, electronic devices, and so forth. In order to capture the semantics of such applications, the aggregation relationship should emphasise the exclusivity; for example, a laptop does not share a CPU or hard disk with other laptops.

In the example shown in Figure 1.11, we need to ensure that every part object is exclusively owned by a particular whole only.

In a nonexclusive composition, a part of one whole object may be shared or referenced by other whole objects, and thus the part is not exclusive. For example, a binary file or a text file can be referenced by more than one directory (see Figure 1.12).

It is important to note that in UML, the term composition refers to exclusive and dependent aggregation. However, we use composition interchangeably with aggregation and use qualifications to distinguish between the different categories.

Figure 1.11. Exclusive composition

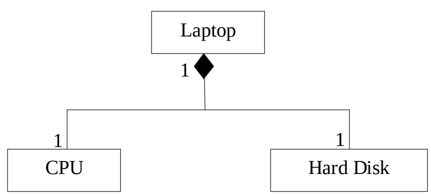
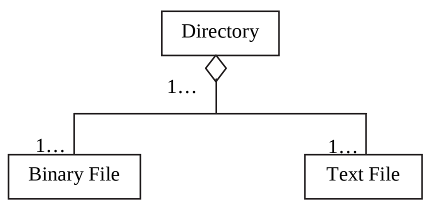


Figure 1.12. Nonexclusive composition



Homogeneous Composition

The previous examples are categorized into a heterogeneous composition since one whole object may consist of several *different types* of part objects. In contrast, homogeneous composition implies that one whole object consists of part objects of the same type.

In the example shown by Figure 1.13, a Hard Disk object consists of several Hard-Disk Controllers. Once we add another type under the whole, the type has changed into heterogeneous composition.

The main advantage of modeling the homogeneous type of composition is that the model is flexible enough for further extensions or modifications to include components of another type. In the case of a mixture of homogeneous and heterogeneous components, the homogeneous composition is indicated by the cardinality, namely, 1 to *n*.

Multilevel Composition Objects or Complex Objects

In many applications, the composition hierarchy may span an arbitrary number of levels. If one gets a composite or aggregated object design that has

Figure 1.13. Homogeneous composition

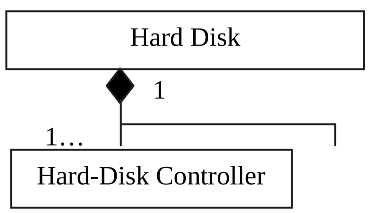
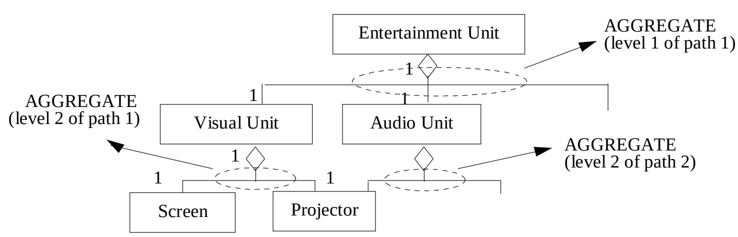


Figure 1.14. Entertainment-unit complex object



component objects that are themselves composite or aggregated objects, then one gets a two-level aggregation or composition hierarchy. This hierarchy could be repeated to several levels of composition or aggregation. Because of the arbitrary number of the part-of relationships between the objects, the objects involved in the composition are also known as *complex objects*.

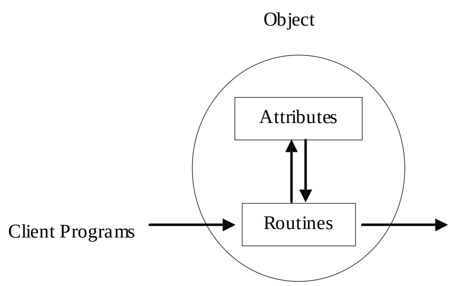
Figure 1.14 shows an example of an entertainment-unit multilevel composition hierarchy. The aggregation relationships in each level of the composition can be seen as a type of simple aggregation relationship (e.g., existence dependent or independent, exclusive or nonexclusive, or homogenous). However, a multilevel composition hierarchy may include different types of aggregation relationships at each level of the composition.

Dynamic Aspects of OOCM

Dynamic aspects can be called implementation or behavioral aspects of OOCM. They involve the creation of the routines. Routines are specified as operations or methods, which are defined in the class that describes the object. The specified routines are the only operations that can be carried out on the attributes in the object. The client of the object cannot change the state (attributes) except by routine call. Routines form the interface between the state of an object and the user.

Routines are implemented in OOCM using the encapsulation concept. Encapsulation, also known as information hiding, prevents the client programs from seeing the internal part of an object where the algorithm of the routines and the data structures are implemented, which does not need to be known by the clients. Figure 1.15 shows the encapsulation of an object.

Figure 1.15. Encapsulation of attributes and routines



Methods as a routine can be divided into two main parts: the generic method and user-defined method.

Generic Methods

Generic methods are used to access attributes of an object. The concept behind the need for generic methods is encapsulation, in which attributes associated with an object can be accessed directly only by the methods of the object itself. In object orientation, attributes refer to simple or primitive types (such as integer, string, etc.), user-defined objects (such as Person, Student, etc.), or collection types (such as list, array, set, and bag). Generic methods should provide ways for accessing the different types of attributes.

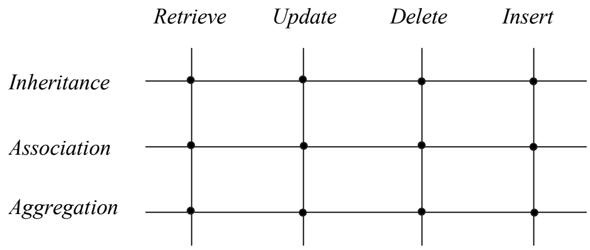
Generic methods may have the following operations: *retrieval*, *update*, *delete*, or *insert*. The retrieval generic methods are methods to retrieve the attributes' values. They are actually read-only methods and are often known as queries. The update generic methods are used to update the values of the specified attributes. The delete generic methods are used to delete the specified attributes' values. Since the update and the delete generic methods manipulate the values of the specified attributes, they are often associated with the data-manipulation language (DML). The insert generic methods insert new values to the specified attributes. This is similar to the concept of object creation in an object-oriented environment.

All of the above operations (i.e., retrieve, update, delete, and insert) can be applied to inheritance, association, and aggregation hierarchies. Generic methods on inheritance hierarchies are methods that access attributes in inheritance hierarchies. Normally, the method is declared in a subclass and accesses the value of the superclasses' attributes, and it may also access local attributes (attributes of the subclass) as well.

Generic methods on association structures are methods that access attributes of classes along an association structure. If two classes are associated through an association relationship, methods declared in one class may access attributes of the other class.

Generic methods on aggregation hierarchies are methods that access attributes of other specified classes in an aggregation hierarchy. If the method is declared in a whole class, the methods may access attributes of its part classes. The opposite is applied if the method is declared in a part class, where it may access attributes of the whole class as well as its own. Figure 1.16 illustrates the

Figure 1.16. A taxonomy for generic methods



taxonomy of generic methods in object orientation. The matrix indicates the operations in generic methods including retrieve, update, delete, and insert, and object hierarchies including inheritance, association, and aggregation hierarchies.

In the transformation of generic methods into object-relational operations, we consider all of the operations specified above (i.e., retrieval, update, delete, and insert) and operations on object hierarchies (i.e., inheritance, association, and aggregation).

In this book, a semiautomatic transformation of object-oriented generic methods into a set of object-relational operations is presented. These relational operations can subsequently be implemented as stored procedures. The transformation rules are determined by the different types of attributes being accessed by the generic methods (*result type*), as mentioned above, and the structure of the objects that own the generic methods.

User-Defined Methods

As suggested by the name, user-defined methods are nongeneric methods that are defined by users in order to perform certain database functionality. In this book, the representation of user-defined methods in object-relational databases is presented. The functions and expressions used to represent user-defined methods are supported by most commercial database systems available today. Ways by which to optimise queries that access the stored procedures are also described.

New Era of Object-Relational Approaches

As mentioned in the previous sections, object-oriented concepts provide an excellent basis for modeling because the object structures permit analysts and designers to focus on a problem at a high level of abstraction, but with a resulting design that can be easily and practically implemented. In the past few years, more software has been written using the object-oriented paradigm. Many prototypes as well as commercial object-oriented DBMSs (OODBMSs) such as O2, Versant, POET, ONTOS, Objectivity, GemStone, and ObjectStore have been developed by both industrial and research laboratories around the world (Deux, 1990; Kim, 1990; Robie, Lapp, & Achach, 1998; Stonebraker, 1990).

Nevertheless, object-oriented databases are still not as widely used as relational databases (RDBs) that rest on a firm formal foundation. Stonebraker (1996) reports that the OODBMS market is 100 times smaller in comparison with the RDBMS market, and it is expected that this figure will be maintained in many years to come. It is a fact that RDBs still largely dominate the database community. RDBMS technology is considered mature and has been the basis of a large number of applications around the world. However, the relational approach, when used to model real-world problems, is not nearly strong enough to model all the different kinds of relationships, both static and dynamic. This also includes the fact that the relational model has a lack of semantic features and an inability to represent complex structures and operations (Kim, 1995).

The object-oriented data model has significant benefits in the areas of semantic data modeling. These rich semantics are lacking in the relational model. On the other hand, in the implementation of the data model, there are major strengths of the existing RDBMS that OODBMS does not have. These include RDBMS's widespread acceptance as well as the simplicity of the query processing.

The above reasons have stimulated the emergence of a new approach in the development of database systems, namely, the object-relational approach. In general, this approach is a method of combining both object-oriented and relational approaches with the aim of incorporating the advantages of each and eliminating their drawbacks.

In the next sections, the object-relational approach is grouped into five major categories.

OOCM Implemented on Relational Databases

Despite the differences between the object-oriented and the relational paradigm, in reality, most of object-based development systems are still using the RDBMS engine as their persistence mechanism. Therefore, a transformation from object-oriented models into relational structures and operations is crucial.

Much work has been done in this area, where each structure in the OOCM is transformed into implementation in pure RDBMS (Rahayu, Chang, Dillon, & Taniar, 2000, 2001). This method is especially useful when the RDBMS chosen for the implementation is a pure RDB that does not support object-oriented extensions (SQL 92 standard).

Object Wrappers on Relational Systems

An *object wrapper* (see Figure 1.17) is basically a layer on top of a conventional RDB engine that simulates object-oriented features. One of the main aims of this layer is to transform object queries (OQL) submitted by users into relational queries. OQL is an enhanced relational query with additional capabilities to understand arbitrary complex types as well as user-defined operations. Thus, the user is allowed to interact with the system through the object wrapper as if it were an OODBMS even though the underlying mechanism is RDBMS.

It is necessary to have a solid transformation methodology that can be used by the object wrapper to perform the translations of the object-oriented features to their relational equivalent for interaction with the underlying RDBMS. The transformation methodology should not only provide translation techniques, but also ensure efficient access to the result of the translation process.

Figure 1.17. Object wrappers on relational systems

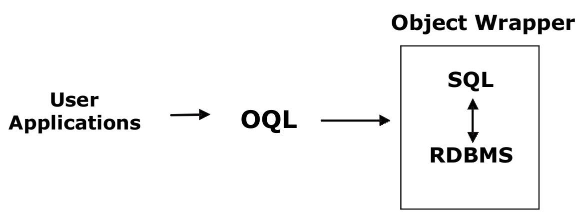
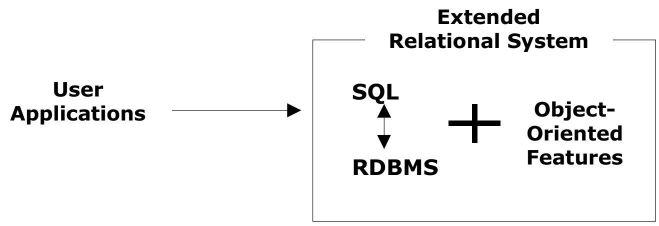


Figure 1.18. Extended relational systems



Extended Relational Systems

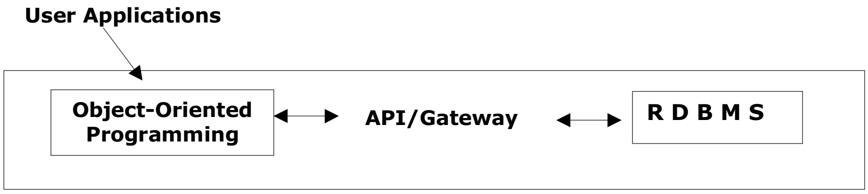
In this category, relational systems are extended in order to support object-oriented features (see Figure 1.18). The extensions include the support of object identifiers, inheritance structures, complex type representations, and user-defined operations.

The SQL3 standard and the forthcoming SQL4 may provide the solution to standardizing the extensions to RDBMS. However, until now, work on SQL4 is still ongoing, and none of the existing extended relational systems fully supports the standard, even for SQL3.

There are several different approaches that belong to this category. One of the approaches used for capturing the concept of complex structures is to allow relations to have attributes that are also relations, thereby abandoning the first normal form of the relational model. The model, which is known as the nested-relations or NF2 (nonfirst normal form) data model (Scheck & Scholl, 1986), can be used to represent composite objects and set-valued attributes. An example is a DBMS prototype developed by Dadam et al. (1986) that supports the NF2 model.

Another approach in this category is an extension of a conventional SQL that is used to retrieve and manipulate data. For example, POSTGRES (Stonebraker, 1986) provides an extended SQL called POSTQUEL query with the ability to capture the concept of abstract data types (encapsulated data structures and methods), inheritance structures, and object identity. Another example is Starburst (Lindsay & Haas, 1990; Schwarz et al., 1986) that extends the relational algebra and supports user-defined operations and complex types. Oracle™ 8 and above provide the implementation of most of the above extensions. It allows the creation of objects and user-defined types, encapsulation of data structure and methods, complex relationships including inherit-

Figure 1.19. Object-relational coexistence approach



ance and referencing, as well as composition through nested tables and collection types. Because of this, we will use Oracle™ throughout this book to demonstrate the design and implementation of object-relational databases.

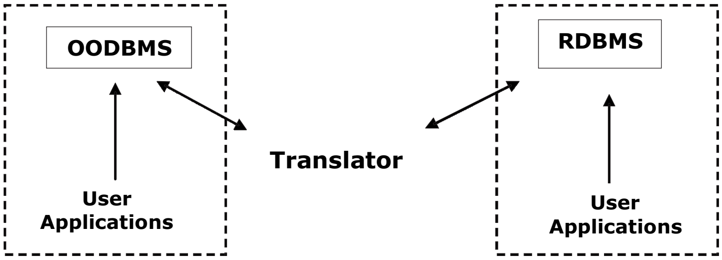
Object-Oriented System and RDBMS Coexistence

As opposed to a hybrid system in which both object-oriented and relational systems are combined into a single system, the coexistence approach provides an interface that allows object-oriented systems to access and manipulate a relational system by encapsulating RDB entities such as tables and queries into objects. For example, Borland Database Engine API for Borland C++ Builder allows an object-oriented programming language C++ to access standard data sources in Paradox, dBase, or Interbase format. Similar interfaces such as Microsoft Jet Database Engine are used by Microsoft Visual C++.

This coexistence approach (see Figure 1.19) is obviously quite attractive to many commercial vendors. The main reason for this is that the cost of building the overall system is minimized by taking the two systems (object-oriented system and RDBMS) and letting them coexist. The work required to accommodate the new functionality in both systems and to let them communicate in a coexistent environment is far less than the effort needed to combine both systems into a single hybrid system.

Even though no attempt is made to enforce the storage of the object instances within the schema for the RDBMS, it is essential to have a solid methodology for the transformation of the object model into the associated relational schemas that ensures correctness and efficiency of the data storage and retrieval.

Figure 1.20. OODBMS-RDBMS interoperation



OODBMS and RDBMS Interoperation

In the interoperation approach (see Figure 1.20), a request from an originating database side is translated and routed to a target database side for processing. The result is then returned to the originator of the request. To achieve transparency of the interoperation process, translation between the different models of the participating database systems must be performed during the data interchange (Ramfos et al., 1991). There are two major translations needed in this approach:

- schema translations, where the schema of the target database is translated into the data-definition language (DDL) of the originating database side, and
- query translations, where a query in the DML of the originating database side (posed against the above produced schema) is translated into the DML of the target database side.

This approach is frequently used in a multi-DBMS. A multi-DBMS is a system that controls multiple translators (or gateways), one for each remote database (Kim, 1995). In this type of environment, it is possible for one application program to work with data retrieved from both one OODBMS and one or more RDBMSs.

To develop a comprehensive translator, the identification of the schemas and operations owned by each of the participant database sides, OODBMS and RDBMS, needs to be fully understood. A complete methodology that supports the theoretical mapping from the originating schema into the target schema is essential. Ideally, this mapping methodology should cover both the structural component as well as the dynamic component of the database systems.

Object-Relational Database System

The relational data model has a sound theoretical foundation, is based on the mathematical theory of relations and first-order logic, and gives the users a simple view of data in the form of two-dimensional tables. Many DBMSs use this relational model. Even nonrelational systems are often described as having supporting relational features for commercial purposes. The model's objectives were specified as follows.

- To allow a high degree of data independence. The application programs must not be affected by modifications to the internal data representation, particularly by the changes of file organizations, record orderings, and access paths.
- To provide substantial grounds for dealing with data semantics, consistency, and redundancy problems.
- To enable the expansion of set-oriented DMLs.
- To become an extensible model that can describe and manipulate simple and complex data.

The first two objectives have been achieved by the relational model, mainly because of the simplicity of the relational views presenting the data in two-dimensional tables and the application of the normalization theory to database design.

The third objective has been achieved by the use of relational algebra, which manipulates tables in the same way that arithmetical operators manipulate integers, and by nonprocedural languages based on logical queries specifying the data to be obtained without having to explain how to obtain them.

The last objective is the essence of current developments concerning extended relational and object-relational models.

Case Study

The Australian Education Union (AEU) keeps the records of its properties and activities in a database using an object-oriented concept. Property can be divided into two main categories: building and vehicle. Beside these two, there are also other minor properties that are not categorized into building and vehicle. Each building has several rooms and each room has computers in it. Some of the rooms also have overhead projectors (OHPs).

The union employees' records are kept in a separate class. Employees can be divided into two types: office staff and organizers. Management is not included in these two categories, although their data is also kept in the employee class. While office staff work only internally in the union, the organizers have to represent teachers in the area to which they have been assigned. One organizer can represent many teachers, but one teacher can have only one organizer as her or his representation. For this purpose, each organizer has been given one vehicle, and that vehicle may be used only by that particular organizer. Each organizer will be assigned only one area, which can be divided into several suburbs. The area and suburb data are also kept in separate classes.

The union also keeps records for teachers who are union members. All of these teachers have to work in government schools. Although it is not common, a teacher can work in more than one school. The type of school that can liaise with AEU has to be categorized into one of the three distinct types: primary school, secondary school, and technical college (TechC).

We will draw an object-oriented model of the AEU database and determine the type where necessary. We will identify the objects and the relationships as follows.

- **Identify Objects**

  To start with, we know that there will be a union object to store the data about the AEU organization. It also has a property object that can be divided into building and vehicle objects. Furthermore, there is a room object that is composed of PC and OHP objects.

  Next, we will need an employee object for AEU's employee records. Their types are also objects: Office Staff and Organizer. For working area and suburb, we need two new objects as well.

Finally, as employees will need to work with teachers, we need a teacher object. Along with that, the additional objects of School and its specializations—Primary, Secondary, and TechC—will be added.

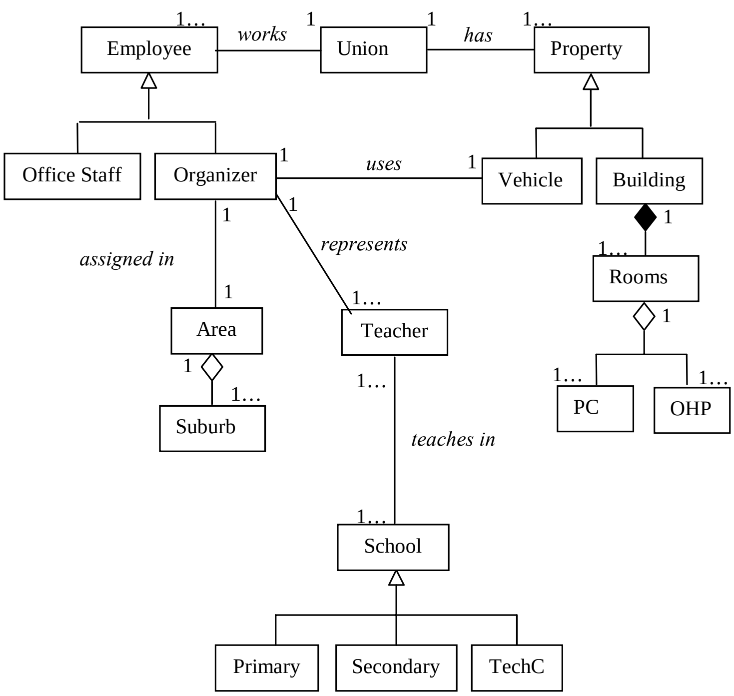
- **Identify Relationships**

  There will be three types of relationships that we need to recognize before producing the object-oriented model diagram.

  First, we need to identify inheritance relationships. Inheritance can be shown by the generalization-specialization feature. One of them is between Employee and its specializations Office Staff and Organizer. Property can also be specialized into Vehicle and Building. And the last one is the specialization of School into Primary, Secondary, and TechC.

  Second, we need to identify association relationships. This relation is usually the most frequent relation in an object-relational system. From the union object there are two associations: one to Property (one to many) and the other one to Employee (one to many). Organizer has three

Figure. 1.21. Object-oriented diagram of AEU case study



association relationships, that is, associations to Vehicle (one to one), Area (one to one), and Teacher (one to many). The last association relation is between Teacher and School (many to many).

The last relationship type is aggregation. Building has two levels of aggregation. The first level is homogeneous aggregation to Room, and the second level is to PC and OHP. Another homogeneous aggregation relationship is between Area and Suburb.

After identifying the objects and their relationships, we can draw down the object-oriented model for the AEU case study as it is shown in Figure 1.21.

Summary

An approach to a new model in database systems is needed due to the limitation of the relational model that is widely used commercially. The relational model is not rich enough to represent the high complexity of real-world problems. On the other hand, the object-oriented model that is well recognized as a very powerful approach to model high-complexity problems, such as in procedural languages, is not a well-known database system model. Also, users still like the ease of use of the relational model.

Although the most widely used model of current database systems is a relational model, it can also be extended to adopt the concept of the object-oriented model. In an object-oriented model, the objects encapsulate their attributes and their methods from other objects, thereby facilitating the concept of information hiding. This model also accommodates the structural relationship of classes and objects, which can be categorized into inheritance, association, and aggregation, and the implementation of methods that consist of generic methods and user-defined methods.

References

Ambler, S. W. (1997). Mapping objects to relational databases. In *Building object applications that work.* SIGS Books.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide.* Reading, MA: Addison-Wesley.

Coad, P., & Yourdon, E. (1990). *Object-oriented analysis.* Englewood Cliffs, NJ: Yourdon Press.

Coad, P., & Yourdon, E. (1991). *Object-oriented design.* Englewood Cliffs, NJ: Yourdon Press.

Dadam, P., et al. (1986). A DBMS prototype to support extended NF2 relations: An integrated view on flat tables and hierarchies. *Proceedings of the ACM SIGMOD Conference.*

Deux, O. (1990). The story of O2. *IEEE Transactions on Data and Knowledge Engineering TKDE, 2*(1), 91-108.

Dillon, T. S., & Tan, P. L. (1993). *Object-oriented conceptual modeling.* Prentice-Hall.

Dittrich, K. R. (1989). Object-oriented database systems for information systems of the future. In *Seminar notes.* Melbourne, Australia.

Halper, M., Geller, J., & Perl, Y. (1992). "Part" relations for object-oriented databases. *Proceedings of the 11th International Conference on the Entity-Relationship Approach.*

Kim, W. (1990). *Introduction to object-oriented databases.* The MIT Press.

Kim, W. (1995). *Modern database systems.* Addison-Wesley.

Lindsay, B. G., & Haas, L. M. (1990). Extensibility in the starburst experimental database system. In *IBM symposium: Database systems of the 90s* (pp. 217-248). Springer-Verlag.

Rahayu, J. W., Chang, E., Dillon, T. S., & Taniar, D. (2000). A methodology for transforming inheritance relationships in an object-oriented conceptual model to relational tables. *Information and Software Technology Journal, 42*(8), 571-592.

Rahayu, J. W., Chang, E., Dillon, T. S., & Taniar, D. (2001). Performance evaluation of the object-relational transformation methodology. *Data and Knowledge Engineering, 38*(3), 265-300.

Ramfos, A., et al. (1991). A meta-translation system for object-oriented to relational schema translations. In M. S. Jackson & A. E. Robinson (Eds.), *The Proceedings of the Ninth British National Conference on Databases (BNCOD*).

Robie, J., Lapp, J., & Achach, D. (1998). XML query language (XQL). *The Query Languages Workshop.*

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-oriented modeling and design.* Prentice Hall.

Scheck, H. J., & Scholl, M. H. (1986). The relational model with relation-valued attributes. *Information Systems, 11*(4).

Schwarz, P. M., Chang, W., Freytag, J. C., Lohman, G. M., McPherson, J., Mohan, C., et al. (1986). Extensibility in the starburst database system. *Proceedings of OODBS 1986* (pp. 85-92).

Stonebraker, M. (1986). Object management in postgres using procedures. *Proceedings of OODBS 1986* (pp. 66-72).

Stonebraker, M. (1990). The postgres DBMS. *Proceedings of SIGMOD 1990* (p. 394).

Chapter Problems

1. List five major categories of an object-relational approach.
2. Discuss the static and dynamic aspects of an object-oriented model.
3. Discuss the background of object-relational DBMS (ORDBMS) development.
4. Explain the terms existence-dependent, existence-independent, exclusive–composition, and nonexclusive composition for aggregation relationships.
5. Each postgraduate student at *L* University needs to maintain a list of references that he or she needs for research. For this purpose, references used are categorized into four types: book, article in a journal, conference paper, and PhD thesis. A reference can be included in one type only. The fields of each type of reference are listed in the following table.

Reference Type	***Fields***
Book	title of the book, list of authors, publisher
Article Journal	title of the paper, list of authors, title of the journal, volume, editor, publisher
Conference Paper	title of the paper, list of authors, title of the conference, publisher
PhD Thesis	title of the thesis, author, school

Assuming that there are five classes, that is, References, Book, Article_Journal, Conference_Paper, and PhD_Thesis, develop the class hierarchy for the above description, and draw the corresponding class diagram. You also need to identify the relationship between references and another class, Postgraduate. Assume some attributes where necessary.

6. AllBooks Library wants to extend its database by using the object-oriented concept. For this purpose, in the database the authors are categorized according to their backgrounds: industry based or academic. If the author is an academic, the database needs to be further categorized into research or teaching staff. They found that many academics are also involved in industry and vice versa. However, it is found that an academic may simultaneously be involved in both research and teaching. To simplify the database, the developer decides that an academic can only be recorded as a research staff or a teaching staff depending on his or her primary role.

   In the database, the books that the authors have written or edited are kept in a different object named Course_Manual. For each datum in Course_Manual, there are descriptions of each chapter that are kept as another object. Draw the diagram for the object-oriented model described above.

7. A new fast-food company, Hungry Burger, has just opened its first restaurant in the country. One of its main menu items is called Huge Meal. The Huge Meal package includes a big special hamburger, a drink, and a generous-size bag of fries. The construction of the hamburger at Hungry Burger has a special order that has to be followed. On the lower half of the bun, kitchen staff will put a slice of meat patty, followed by two pieces of lettuce, a slice of cheese, and a slice of tomato. The fries are made of potatoes fried in grease. The hamburger and the fries may be sold separately or with another package on the menu.

Draw the aggregation diagram for Hungry Burger's Huge Meal. Explain the dependency, order, and also the exclusiveness where necessary.

8. The Fast Run Bicycle Company is a major bicycle retailer. Unlike other bicycle companies, it assembles its own bicycles to meet the customers' requirements. The three main components are seats, frames, and wheels. These three main components are inherited from the part class and these parts are bought from several manufacturers. There are three categories of bicycles assembled by Fast Run: racing, mountain, and road bicycles. From the description given, draw a diagram for Fast Run that shows the aggregation, inheritance, and association relationships.

Chapter Solutions

1. Five major categories of an object-relational approach are as follow.

   - OOCM implemented on relational databases
   - Object wrappers on relational systems
   - Extended relational systems
   - Object-oriented systems and RDBMS coexistence
   - OODBMS and RDBMS interoperation

2. Static aspects of an object-oriented model include the object and class data structure that facilitates encapsulation, and the relationships that can be divided into three major divisions: inheritance, association, and aggregation. The dynamic aspects of an object-oriented model include the implementation of methods or operations, both generic methods and user-defined methods.
3. ORDBMS is developed to add the desirable features of the object-oriented model to the relational database system. RDBMS has been widely used commercially and in addition, it is also reasonably simple to implement. However, RDBMS cannot be used to represent certain complex problems.

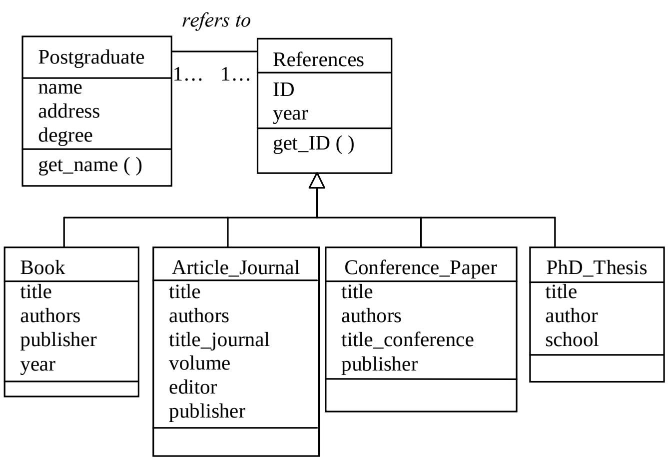
An object-oriented model can capture most complex-problem domains; however, the database based on the object-oriented model, OODBMS, is still not as widely used. It is expected that instead of replacing the earlier RDBMS, OODBMS will coexist in order to serve some specific problem areas only. Therefore, the combination of both strengths have been explored and implemented in the new database system: ORDBMS.

4. Existence-dependent composition is the type of aggregation where the part objects are totally dependent on the whole object. Thus, by removing the whole object, we will automatically remove the part objects. On the other side, existence independent is the type of aggregation where the part object can still exist although its whole object is removed.

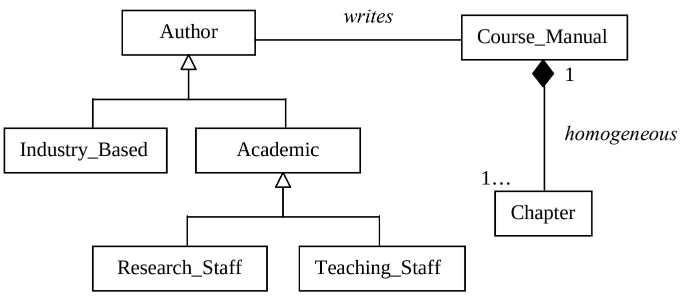
   Exclusive composition is the type of aggregation where the whole object is the sole owner of the part object. On the other side, nonexclusive composition is the type of aggregation where a part object of one whole object may be shared or referenced by other whole objects.

5. There is an inheritance relationship between the reference object to the subclass type.

   The association between Postgraduate and References is many to many, where each reference can be used by many postgraduates, and each postgraduate can refer to many references.

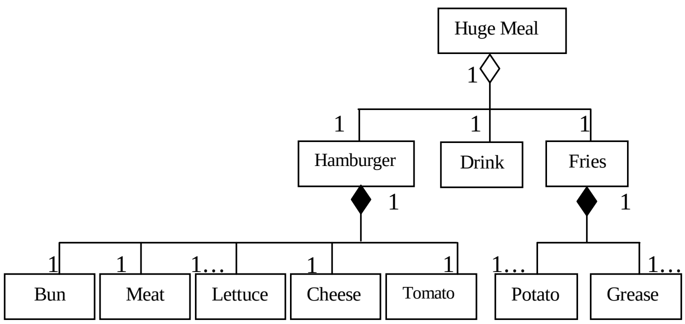


6. There is an inheritance relationship between superclass Author and its subclasses. There is also an aggregation relationship between the Course_Manual and Chapter classes, which in this case is homogeneous. The whole object consists of part objects that are the same type.



7. The aggregation at Level 1 is existence independent because the part object can be sold separately without the whole object. It is an exclusive composition because one part, for example, one hamburger, can only be a composite of one whole part.

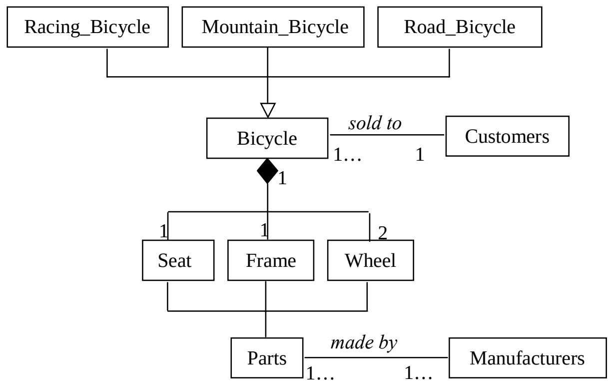
   The aggregation at Level 2 is existence dependent. There is room for argument for this one. Although all parts can exist on their own, they do not have value. This aggregation is also an exclusive composition because one part, for example, one particular bun, can only be a part of one particular hamburger.



8. Bicycle is an aggregation of Seat, Frame, and Wheel. The type is an exclusive composition as a particular part can only be incorporated into a particular whole. It is also an existence-dependent composition because the seat, frame, and wheels do not have their own value at Fast Run unless they are assembled into a bicycle.

   The bicycle class also has an inheritance relationship to the racing, mountain, and road bicycles. The parts class with the seat, frame, and wheel classes show another inheritance relationship.

Finally, there is a one-to-many association relationship between Customers and Bicycle, and a many-to-many relationship between Parts and Manufacturers.



Chapter II

Object-Oriented Features in Oracle™

In this chapter, we will describe Oracle™ features that can be used to support the implementation of an object-oriented model. As an overview, Section 2.1 will outline some of the original features within a standard relational model. The next sections will illustrate the additional object-oriented features. We will use these new features for our implementation in the subsequent chapters.

Relational-Model Features

In a relational model, the attributes are stored as columns of a table and the records are stored as rows of a table. As in most standard RDBMSs, Oracle™ provides a create-table statement following the SQL standard. After the declaration of the table name, we define the attributes' names and their data types. We can also perform the checking of attribute value. In the table, Oracle™ enables users to determine the uniqueness of the records by defining the primary key.

Oracle™ also enables the usage of a foreign key. The foreign-key attribute in a table refers to another record in another table. In addition to the foreign key,

Figure 2.1. Create-table statement

```
General Syntax:

CREATE TABLE <table schema>
   (key attribute    NOT NULL,
    attribute        attribute type,
    attribute        attribute type
    [CHECK (<attribute value> IN (set of values))]
    PRIMARY KEY      (key attribute));

Example:

CREATE TABLE Employee
   (id         VARCHAR2(10)      NOT NULL,
    name       VARCHAR2(20),
    address    VARCHAR2(35),
    emp_type   VARCHAR2(8)
       CHECK(emp_type IN ('Manager', 'Worker')),
    PRIMARY KEY (id));
```

we can specify the referential integrity constraint every time we want to manipulate the target of a foreign-key reference. There are three types of constraint.

- **Restrict:** The manipulation operation is restricted to the case where there are no such matching attributes; it will be rejected, otherwise.
- **Cascade:** The manipulation operation, such as delete and update, cascades to the matching attributes.
- **Nullify or set null:** The manipulation operation is done after the foreign key is set to null.

Oracle™ performs the restrict integrity constraint as default. It prevents the update or deletion of a superclass key if there is a row in the subclass table that is referencing the key. However, Oracle™ provides only an on-delete integrity constraint. Therefore, to perform integrity constraint on other manipulations such as insert and update, we might need to use triggers.

Once we have created the table, we can perform the data manipulation. The manipulation can take form in the insertion, deletion, or update of data. The syntax of each of these is shown.

Figure 2.2. Create table with referential integrity checking

```
General Syntax:

CREATE TABLE <table schema> OF <object schema>
   (key attribute    NOT NULL,
    attribute        attribute type,
    PRIMARY KEY      (key attribute),
    FOREIGN KEY      (key attribute)
    REFERENCES <referenced table schema>(key attribute)
    [ON DELETE][CASCADE|SET NULL]);

Example:

CREATE TABLE Student
   (id         VARCHAR2(10) NOT NULL,
    course     VARCHAR2(10),
    year       VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person ON DELETE CASCADE);
```

Figure 2.3. Data manipulation in Oracle™

```
General Syntax of Insertion:

INSERT INTO <table schema> [(attribute, ....,attribute)]
VALUES (attribute value, ....,attribute value);

Example:

INSERT INTO Student
VALUES ('1001', 'BEng', '2005');

General Syntax of Deletion:

DELETE FROM <table schema>
WHERE <statements>;

Example:

DELETE FROM Student
WHERE id = '1001';

General Syntax of Update:

UPDATE <table schema>
SET <statements>
WHERE <statements>;

Example:

UPDATE Student
SET year = '2005'
WHERE id = '1001';
```

Object-Oriented Features

More recent commercial RDBMSs such as Oracle™ 8 and above (Loney & Koch, 2000, 2002; ORACLE™ 8, 1997) have extended their database systems with object-oriented features. In Oracle™ 8 and above, these features include enhancement of the existing data types with new data types including object types and user-defined types, and so forth.

Object Types and User-Defined Types

In Oracle™, a statement "create type" is used to create a new data type (object type) that can then be used as a generic type to create a table using the statement "create table," or to create another data type. The general syntax for these two create statements is shown in Figure 2.4. "As object" is used after creating an object type. Note that "or replace" is optional. By having this additional phrase, an object with the same name will automatically be replaced with the newest version of the object type. Figure 2.4 also shows an example of using object type Person_T as an attribute type in a new table, Course.

Figure 2.4. Oracle™ object type

```
General Syntax:

CREATE [OR REPLACE] TYPE <object schema> AS OBJECT
   (attribute  attribute type, ....,
    attribute  attribute type)
/

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
   /

CREATE TABLE Course
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(20),
    lecturer         Person_T);
```

Figure 2.5. Oracle™ varying array type

```
General Syntax:

CREATE [OR REPLACE] TYPE <object schema> AS VARRAY(n) OF (object/data
type)
/

Example:

CREATE OR REPLACE TYPE Persons AS VARRAY(3) OF Person_T
/

CREATE TABLE Course
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(20),
    lecturer         Persons);
```

Collection Types

Oracle™ allows the creation of an array type (*varray* or varying array). The syntax is basically using the same statement "create type" with the additional statement "as varray(*n*) of" followed by the object or the data type. Following Figure 2.4, it is possible to have more than one lecturer for a particular course, and therefore a new array of Persons can be defined.

Another extension is the support of nested tables, as shown in Figure 2.6. To create a table object, we use the same "create type" statement with the additional "as table of" statement following the name of the object table. This object table can then be used as a column in a table. When a table type appears as the type of a column in a table or as an attribute of the underlying object type, Oracle™ stores all of the nested table data in a single table, which is associated with the enclosing table or object type. Every time we create a table with columns or column attributes whose type is a nested table, we have to include the nested-table storage clause, "nested table (object table column schema) store as" followed by the separate storage-table name. Using the previous example from Figure 2.4, another data type called Person_Table_T can be created based on the Person_T data type to store the instances of a person. Note that Oracle™ 9 and above have also enabled users to create multilevel nested tables.

Figure 2.6. Oracle™ nested table

```
General Syntax:

CREATE [OR REPLACE] TYPE <object table schema> AS TABLE OF (object
schema)
/
CREATE TABLE <table schema>
   (attribute        attribute type, ....,
    attribute        attribute type,
    nested item      object table schema);
   NESTED TABLE nested item STORE AS storage table schema;

CREATE TABLE <table schema>
   (attribute              attribute type, ....,
    outer nested item      object table schema);
   NESTED TABLE <outer nested item>
      STORE AS <outer storage table schema>
      (NESTED TABLE <inner nested item>
         STORE AS <inner storage table schema>);

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
   /

CREATE OR REPLACE TYPE Person_Table_T AS TABLE OF Person_T
/

CREATE TABLE Course
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(20),
    lecturer         Person_Table)
    NESTED TABLE lecturer STORE AS Person_tab;
```

Object Identifiers

In an object-oriented system, the OID is system generated and is used as a reference to locate the particular object. In Oracle™, the notion of an OID as a logical pointer is not supported; however, the concepts of an OID to uniquely identify a record (i.e., as a primary key) can be used. This is particularly useful in a deep inheritance hierarchy, where all subclasses have to carry the OID of the superclass in order to establish the connection between the superclass and its subclasses.

Figure 2.7. Oracle™ object-identifiers implementation

```
General Syntax:

CREATE TABLE <table schema> OF <object schema>
   (key attribute    NOT NULL,
    attribute        attribute type,
    PRIMARY KEY      (key attribute),
   FOREIGN KEY       (key attribute)
   REFERENCES <referenced table schema>(key attribute);

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
   /

CREATE OR REPLACE TYPE Employee_T AS OBJECT
   (person_id        VARCHAR2(10),
    title            VARCHAR2(10),
    salary           NUMBER)
   /

CREATE TABLE Person OF Person_T
   (person_id  NOT NULL,
    PRIMARY KEY (person_id));

CREATE TABLE Employee OF Employee_T
   (person_id  NOT NULL,
    PRIMARY KEY (person_id),
FOREIGN KEY (person_id) REFERENCES Person(person_id));
```

Figure 2.7 illustrates the implementation of using OID to keep the inheritance between the superclass and its subclasses. Note that we can create a table from an object and determine the primary keys and foreign keys in this table. Every time we determine the foreign key, we have to use a "references" statement followed by the table and the column that is being referred. The general syntax for this primary key and foreign key implementation is shown in Figure 2.7. The table created is derived from an object type. Thus, we do not have to specify the attribute type anymore. They have to be identified while we create the object type. Note, however, that we can add a constraint "not null" statement to avoid a "null" value of an attribute. It is needed for particular attributes.

Figure 2.8. Oracle™ relationship using object references

```
General Syntax:

CREATE TABLE <table schema>
   (object REF (object schema) [SCOPE IS (table schema)]);

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
   /

CREATE TABLE Academic_Staff OF Person_T;

CREATE TABLE Course
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(20),
    lecturer   REF Person_T SCOPE IS Academic_Staff);
```

Relationships using *Ref*

Oracle™ provides a way of referencing from one object to another by using the keyword *ref*. This object-referencing technique can be used to replace the standard "join" operations to traverse from one object to another.

We can then run a query:

```
SELECT C.course_name
FROM Course C
WHERE C.lecturer.person_name = 'Rahayu';
```

In the example above, the "scope is" statement is used to specify the exact table being referenced by the object. Whenever the scope parameter is used, the database engine will perform a join operation, which can be optimized using indexes. On the contrary, if the scope parameter is omitted and more than one table has been created using the given object type, the database engine will navigate through a set of object reference values in order to identify the location of the requested records (Dorsey & Hudicka, 1999).

In the following chapters, we will not use the "scope is" parameter in our table-creation statement. In most situations, we will not build more than one table for each object type we declared, thereby avoiding the situation where the database engine has to navigate through a number of object references. When only one table is created for the object type, the ref operator will directly point to the associated reference.

Cluster

Oracle™ provides a clustering technique that can be very useful for an aggregation relationship. A cluster is created and will be defined in terms of all components that take part in the aggregation relationship, as is shown in Figure 2.9.

Figure 2.9. Oracle™ cluster

```
General Syntax:

CREATE CLUSTER <cluster schema>
   (cluster attribute           attribute type);

CREATE TABLE <table schema>
   (cluster attribute      attribute type,
    attribute              attribute type, ....,
    attribute              attribute type)
   CLUSTER <cluster schema> (cluster attribute);

CREATE INDEX <index schema> ON CLUSTER <cluster schema>;

Example:

CREATE CLUSTER HD_Cluster
   (hd_id            VARCHAR2(10));

CREATE TABLE Hard_Disk
   (hd_id            VARCHAR2(10) NOT NULL,
    capacity         VARCHAR2(20),
    PRIMARY KEY (hd_id))
   CLUSTER HD_Cluster(hd_id);

CREATE INDEX HD_Cluster_Index
   ON CLUSTER HD_Cluster;
```

Inheritance Relationships using *Under*

Oracle™ 9 and above have a new feature that accommodates inheritance-relationship implementation. We do not have to use a primary-foreign-key relationship in order to simulate the relationship between a superclass and its subclasses.

To implement subtypes, we need to define the object as "not final" at the end of its type declaration. By default, without the keyword, the object type will be treated as final and no subtypes can be derived from the type. Oracle™ provides the keyword *under* to be used with the statement "create type" to create a subtype of a supertype such as shown in Figure 2.10.

Figure 2.10. Oracle™ "under" features

```
General Syntax:

CREATE [OR REPLACE] TYPE <super-type object schema> AS OBJECT
   (key attribute    attribute type,
    attribute        attribute type,...,
    attribute        attribute type) [FINAL|NOT FINAL]
/

CREATE [OR REPLACE] TYPE <sub-type object schema> UNDER <super-type
object schema>
   (additional attribute   attribute type, ....,
    additional attribute   attribute type)
    [FINAL|NOT FINAL]
/

CREATE TABLE <super-type table schema> OF
<super-type object schema>
   (key attribute    NOT NULL,
    PRIMARY KEY (key attribute));

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (id         VARCHAR2(10),
    name       VARCHAR2(20),
    address    VARCHAR2(35)) NOT FINAL
/

CREATE OR REPLACE TYPE Student_T UNDER Person_T
   (course     VARCHAR2(10),
    year       VARCHAR2(4))
/
CREATE TABLE Person OF Person_T
   (id   NOT NULL,
    PRIMARY KEY (id);
```

Encapsulation

Oracle™ provides two different types of encapsulation for an object-relational model. The first is through stored procedures or functions. The second is through member procedures or functions.

Stored Procedure or Function

The declaration of a stored procedure or function is basically very similar to the standard procedure declaration in many procedural languages. The encapsulation is provided by giving a grant to a specific role or user to access the particular stored procedure or function.

We need to use a "create procedure" statement. As in other create statements, the "or replace" statement is optional.

A stored procedure can have parameters attached to it, each of which must be followed by its type. We can also add the mode of the parameters between the parameter and the parameter type that is optional. There are three parameter modes (Oracle™, 1998).

- *In*. The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter is considered read only: It cannot be changed. Then the procedure finishes and control returns to the calling environment; the actual parameter is not changed.
- *Out*. Any value the actual parameter has when the procedure is called is ignored. Inside the procedure, the formal parameter is considered write only; it can only be assigned to and cannot be read from. When the procedure finishes and control returns to the calling environment, the contents of the formal parameter are assigned to the actual parameter.
- *In Out*. This mode is a combination of the two previous modes. The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter can be read from and written to. When the procedure finishes and control returns to the calling environment, the contents of the formal parameter are assigned to the actual parameter.

Figure 2.11. Stored-procedures general syntax

```
General Syntax:

CREATE [OR REPLACE] PROCEDURE <procedure name>
   [parameter [{IN | OUT | IN OUT}] parameter type,
    ....,
    parameter [{IN | OUT | IN OUT}] parameter type)] AS

   [local variables]

BEGIN
   <procedure body>;
END <procedure name>;

GRANT EXECUTE ON <procedure_name> TO <user>;

Example:

CREATE OR REPLACE PROCEDURE Delete_Student(
   delete_id         Student.id%TYPE) AS

BEGIN
   DELETE FROM Student
   WHERE id = delete_id;
END Delete_Student;
/
GRANT EXECUTE ON Delete_Student TO Principal;
```

The stored procedure can have local variables in it. These are variables that are used only in the procedure body. Within the procedure body, we can use SQL statements such as select, insert, update, and delete. Thus, methods that are used to manipulate the database tables can be encapsulated within stored procedures. To run the procedure, we use the general syntax below.

```
General Syntax to Run the Stored Procedure:
EXECUTE procedure name [parameter,...,parameter];
EXECUTE Delete_Student['1001'];
```

Apart from stored procedures, stored functions are also available. Similar to stored procedures, stored functions can be likewise declared as in Figure 2.12. Note that for a function, we have to declare the type of the return value after

Figure 2.12. Stored-functions general syntax

```
General Syntax:

CREATE [OR REPLACE] FUNCTION <function name>
   [parameter [{IN}] parameter type,
    ....,
    parameter [{IN}] parameter type)]
   RETURN datatype IS


   [local variables]


BEGIN
   <function body>;
   RETURN value;
END <function name>;

Example:

CREATE OR REPLACE FUNCTION Student_Course(
   s_id        Student.id%TYPE)
   RETURN VARCHAR2 IS

   v_course    VARCHAR(10);

BEGIN

   SELECT course INTO v_course
   FROM Student
   WHERE id = s_id;

   RETURN v_course;

END Student_Course;
/
```

we declare the function name. In addition, a stored function can take "in" parameters only.

Member Procedure or Function

Member procedures and member functions are physically implemented as PL or SQL procedures or functions, and they are defined together within the specification of the object type. Figure 2.13 demonstrates the general syntax.

Unlike stored procedures, by using member methods we can identify the visibility scope of the methods. There are three types: public, private, and protected. By default, the attributes will be declared public.

Public attributes are visible from the class' interface (Fortier, 1999) and can be accessed by other types, tables, or routines. Private attributes are only visible from internal methods and will not be visible from outside the class specification. Finally, protected attributes are accessible from its own class or from any table or methods that use the class as a subtype.

The biggest advantage of methods over stored routines is the visibility gained by being part of the class. Methods will have access to attributes, procedures, and functions that may not be visible at the class interfaces (private or protected). On the other hand, a stored routine does not have access to these types of attributes, procedures, and functions.

Furthermore, the visibility of the methods inside a class can also be specified as private and protected. Same as attributes, the private methods can only be accessed by internal methods inside the particular class, and protected methods can be accessed only by its own user-defined types or any supertype interface of the particular class. We cannot apply this for stored routines.

Figure 2.14 shows an example of different visibility scopes of attributes and methods. All the attributes in Person are declared public and thus can be visible outside of the type interface. Some attributes in Staff, however, are declared private and protected. These attributes require additional internal methods for access, such as the function RetrieveTotalPayment to access the attributes StaffPayRate and StaffCommRate, and return the total payment. The function RetrieveStaffPhone in Person can be used to access the protected attribute in its subtype, StaffPhone. The procedure RetrievePersonDetail can be used to retrieve the attributes inside Person, including the private function RetrieveStaffPhone.

Finally, member methods have substitutability featured in the inheritance structure. Very often, when we insert data into a table, we wish to store different subtypes derived from a single or multiple supertypes. Using stored routines, we will require a different routine for a different parameter. With the substitutability feature, an instance of a subtype can be used in every context where an instance of a supertype can be used (Fortier, 1999). The context includes the use of different subtypes as parameters of the same function.

Figure 2.13. Method implementation of member procedure

```
General Syntax:

CREATE [OR REPLACE] TYPE <object schema> AS OBJECT
   (attribute  attribute types,
    ....,
    attribute  attribute types,

    MEMBER PROCEDURE <procedure name>
       [(parameter [{IN | OUT | IN OUT}] parameter type,
       ....,
       parameter [{IN | OUT | IN OUT}] parameter type)],

    MEMBER FUNCTION <function name>
       [(parameter [{IN}] parameter type,
       ....,
       parameter [{IN}] parameter type)]
       RETURN datatype);
/

CREATE [OR REPLACE] TYPE BODY (object schema) AS

   MEMBER PROCEDURE <member procedure name>
      [parameter [{IN | OUT | IN OUT}] parameter type,
       ....,
       parameter [{IN | OUT | IN OUT}] parameter type)] IS

      [local variables]

   BEGIN
      <procedure body>;
   END <member procedure name>;

    MEMBER FUNCTION <function name>
       [parameter [{IN}] parameter type,
       ....,
       parameter [{IN}] parameter type)]
       RETURN datatype IS

      [local variables]

   BEGIN
      <procedure body>;
   END <member function name>;

END;
/
```

Figure 2.13. (continued)

```
Example:

CREATE OR REPLACE TYPE Student_T AS OBJECT
    (id                     VARCHAR2(10),
     course                 VARCHAR2(20),
     year                   VARCHAR2(4),

     MEMBER PROCEDURE
       Delete_Student )
/

CREATE OR REPLACE TYPE BODY Student_T AS

   MEMBER PROCEDURE
   Delete_Student IS

   BEGIN
         DELETE FROM Student
         WHERE Student.id = self.id;
   END Delete_Student;

END;
/
```

Figure 2.14. Visibility scope inside a class

```
CREATE TYPE Person
   (PersonID            VARCHAR(10),
    FirstName     VARCHAR(20),
    LastName            VARCHAR(20),
    Domicile            ADDRESS,
    BirthDate     DATE,

    PRIVATE FUNCTION RetrieveStaffPhone,
    PUBLIC PROCEDURE RetrievePersonDetail);

CREATE TYPE Staff UNDER PERSON
   (PUBLIC        StaffStartDate    DATE,
    PROTECTED     StaffPhone        CHAR(10),
    PRIVATE       StaffPayRate            DECIMAL(5,2),
    PRIVATE       StaffCommRate           DECIMAL(5,2),

    PUBLIC FUNCTION RetrieveTotalPayment)
```

Figure 2.15. Member-methods substitutability

```
CREATE TYPE <object1 schema>
(attr_1 data type,...,
 attr_j data type,

 PROCEDURE <procedure1 name>(          -- Original Method
 param_1 parameter type data type,...,
 param_n parameter type data type);
) NOT FINAL/

CREATE TYPE <object2 schema> UNDER <object1 schema>
(attr_1 data type,...,
 attr_j data type,

 OVERRIDING PROCEDURE <procedure1 name>(   -- Overriding Method
 param_1 parameter type data type,...,
 param_n parameter type data type);
)/
```

Summary

Similar to many other DBMSs, Oracle™ was first targeted for RDBs. It has supported standard relational features in SQL including the data-definition language and the data-manipulation language. Due to the increased demand of a more powerful database, Oracle™ has added some object-oriented features into its DBMS. This chapter introduces some of them including the object type, collection type, inheritance, nested tables, and so forth. A list of references below provides more information on the syntax and definition of the features described in this chapter.

References

Dorsey, P., & Hudicka, J. (1999). *Oracle™ 8 design using UML object modelling* (chap. 1). Oracle Press, Osborne McGraw Hill.

Fortier, P. (1999). *SQL3 implementing the SQL foundation standard.* McGraw Hill.

Loney, K., & Koch, G. (2000). *Oracle™ 8i: The complete reference.* Osborne McGraw-Hill.

Loney, K., & Koch, G. (2002). *Oracle™ 9i: The complete reference.* Oracle Press.

ORACLE™ 8. (1997). *Oracle™ 8 product documentation library.* Redwood City, CA: Oracle Corporation.

Urman, S. (2000). *Oracle™ 8i advanced PL/SQL programming.* Oracle Press, Osborne.

Chapter Problems

1. Using Oracle™, create a table to store book records. Each record has the title, the author, the publisher, and the ISBN (International Standard Book Number) that uniquely differentiate the book.
2. Continuing from Question 1, now we want to refer the attribute publisher into a table Publisher that has "name" as the primary key. If a deletion is performed in the publisher table, the associated referring key will be nullified. Alter your create-table statement from Question 1.
3. Write a statement in Oracle™ to implement an ordered collection type of the 20 most expensive book prices in a bookstore.
4. As in Question 1, you want to create a table Book. However, you want to instantiate the table from a specified Book_Type. Write the create-type and create-table statement.
5. *Movie Guide* magazine wants to keep a database of directors and the films that they directed. The director table has the attributes of name, age, and residence. The film is saved as an object with the attributes of title, genre, year, and rating. As a director may direct more than one film, the film object is implemented into the director table using a nesting technique. Show the implementation of the relationships described.
6. Discuss briefly the two mechanisms of encapsulation to implement methods or operations in an object-relational DBMS.

Chapter Solutions

1. We use a create-table statement with the attributes of ISBN, title, author, and publisher. The primary key is the ISBN attribute.

```
CREATE TABLE Book
  (isbn       VARCHAR2(10)      NOT NULL,
   title      VARCHAR2(100),
   author     VARCHAR2(100),
   publisher  VARCHAR2(50),
   PRIMARY KEY (isbn));
```

2. We assume the table Publisher already exists.

```
CREATE TABLE Book
  (isbn       VARCHAR2(10)      NOT NULL,
   title      VARCHAR2(100),
   author     VARCHAR2(100),
   publisher  VARCHAR2(50),
   PRIMARY KEY (isbn),
   FOREIGN KEY (publisher) REFERENCES Publisher
              (Name) ON DELETE NULLIFY);
```

3. For an ordered collection with only one data element (in this case the price), we can use varray.

```
CREATE OR REPLACE TYPE prices AS VARRAY(20) OF
NUMBER(12,2)
/
```

4. First, create the type and then follow this by creating the table.

```
CREATE OR REPLACE TYPE Book_Type AS OBJECT
  (isbn       VARCHAR2(10),
   title      VARCHAR2(100),
   author     VARCHAR2(100),
   publisher  VARCHAR2(50))
   /
```

```
CREATE TABLE Book OF Book_Type
  (isbn NOT NULL,
   PRIMARY KEY (isbn),
   FOREIGN KEY (publisher) REFERENCES Publisher(name)
   ON DELETE NULLIFY);
```

5. For a nested table, we have to create the object type followed by an object table before we can nest it to the table as an attribute.

```
CREATE OR REPLACE TYPE Film_T AS OBJECT
  (title      VARCHAR2(50),
   genre      VARCHAR2(10),
   year       NUMBER,
   rating     VARCHAR2(10))
/

CREATE OR REPLACE TYPE Film_Table_T AS TABLE OF
Film_T
/

CREATE OR REPLACE TYPE Director_T AS OBJECT
  (name             VARCHAR2(20),
   age              NUMBER,
   residence        VARCHAR2(20),
   filmography      Film_Table_T)
/

CREATE TABLE Director OF Director_T
  (name       NOT NULL,
   PRIMARY KEY (name))
  NESTED TABLE filmography STORE AS Film_tab;
```

6. The two mechanisms are encapsulation using stored procedures or functions with grants, and encapsulation using member procedures or functions.

   The first mechanism is based on pure RDBMS practice. It is a specific method for accessing the data that can be privileged to certain users by a grant mechanism. The second mechanism is based on an object-oriented model where the methods are encapsulated inside the class with the attributes. They are usually called member methods such as member procedures and member functions.

Chapter III

Using Object-Oriented Features

In Chapter II, we discussed the different features available in Oracle™ that can be used to implement an object-oriented model. We will use those features in this chapter. The discussion in this chapter will be categorized based on the relationship types.

There are three distinct relationship types that we have to consider in object-oriented modeling for implementation in object-relational databases: inheritance, association, and aggregation. Some manipulations will be needed in order to accommodate the features of these relationships.

Using Inheritance Relationships

The concept of inheritance, where an object or a relation inherits the attribute (and methods) of another object, is not supported in the older versions of Oracle™ (prior to Oracle™ 9). The implementation of an inheritance relationship is established using primary-key and foreign-key relationships (shared ID) in order to simulate the relationship between a superclass and its subclasses.

Union Inheritance Implementation

Figure 3.1 shows an inheritance relationship of union type. It declares that the union of a group of subclasses constitutes the entire membership of the superclass. In a union inheritance, we know that every object in the superclass is an object of at least one of the subclasses. In the example (see Figure 3.1), the union type does not preclude a member of a subclass from being a member of another subclass. For example, a person who is a staff member may also be a student at that university.

In order to simulate the union inheritance, Student and Staff will carry the primary key of the superclass, Person, in their relational tables. The primary key of the superclass becomes a foreign key in the subclasses. The foreign keys in the subclasses are also their primary keys. It becomes the main difference between the primary-key and foreign-key relationships in association and in inheritance. Thus, in Figure 3.1 it is noted that the primary key of Person is also the primary key of both Student and Staff. At the same time, the constraint of the primary-key and foreign-key relationship between the ID attributes in Student and Staff and the ID in Person is maintained in order to make sure that each student and staff is also a person. Thus, we have to specify the referential integrity constraint every time we want to manipulate the target of a foreign-key reference.

If we use the newer Oracle™ version, which supports inheritance using the “under” keyword, we can create Student and Staff subclasses under the superclass Person. The implementation is shown in Figure 3.3. Note that for union inheritance, we need to create one table each for the superclass and all the subclasses. As can be seen in the later sections, this union inheritance has a different way of implementation compared with other inheritance types. Using

Figure 3.1. Union inheritance

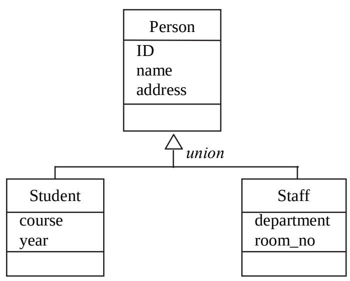


Figure 3.2. Implementation of union inheritance

```
CREATE TABLE Person
   (id         VARCHAR2(10) NOT NULL,
    name       VARCHAR2(20),
    address    VARCHAR2(35),
    PRIMARY KEY (id));

CREATE TABLE Student
   (id         VARCHAR2(10) NOT NULL,
    course     VARCHAR2(10),
    year       VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person ON DELETE CASCADE);

CREATE TABLE Staff
   (id         VARCHAR2(10) NOT NULL,
    department VARCHAR2(10),
    room_no    VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person ON DELETE CASCADE);
```

Figure 3.3. Implementation of union inheritance using "under"

```
CREATE OR REPLACE TYPE Person_T AS OBJECT
   (id         VARCHAR2(10),
    name       VARCHAR2(20),
    address    VARCHAR2(35)) NOT FINAL
/

CREATE TABLE Person OF Person_T
   (id         NOT NULL,
   PRIMARY KEY (id));

CREATE OR REPLACE TYPE Student_T UNDER Person_T
   (course     VARCHAR2(10),
    year       VARCHAR2(4))
/

CREATE TABLE Student OF Student_T
   (id         NOT NULL,
   PRIMARY KEY (id));

CREATE OR REPLACE TYPE Staff_T UNDER Person_T
   (department VARCHAR2(10),
    room_no    VARCHAR2(4))
   /

CREATE TABLE Staff OF Staff_T
   (id         NOT NULL,
    PRIMARY KEY (id));
```

the "under" keyword, normally we do not need to create separate tables for the subclasses because the table created for the superclass can also be used to store the instances of the subclasses. However, in this union type of inheritance, we need to allow a particular person to be both a student as well as a staff. If we are to store all instances into one superclass table, we will not be able to store the two records together as they will violate the primary-key constraints (i.e., two records with the same ID). Therefore, we need to create a separate table for each of the subclasses to allow the same person's record to appear in both the Student as well as Staff tables. We also need to create a table for Person to store persons who are neither staff members nor students.

Mutual-Exclusion Inheritance Implementation

Mutual-exclusion inheritance declares that a group of subclasses in an inheritance relationship is pairwise disjointed. An example of this type is shown in Figure 3.4. This example is called mutual exclusion because there is no manager who is also a worker, and vice versa. However, in this case there may be an employee who is neither a manager nor a worker.

The best way to handle mutual-exclusion inheritance without losing the semantics of the relationship is by adding to the superclass table an attribute that reflects the type of the subclasses or has the value null. For example (see Figure 3.4), in the table Employee, an attribute called emp_type is added. Thus, emp_type can take the values manager, worker, or null. There are no employees that can have two values for this attribute, such as a manager who is also a worker simultaneously (mutual exclusion). Figure 3.5 shows the implementation details. Note that we use the "check" keyword for the purpose of checking the value of an attribute in a set of values.

Figure 3.4. Mutual-exclusion inheritance

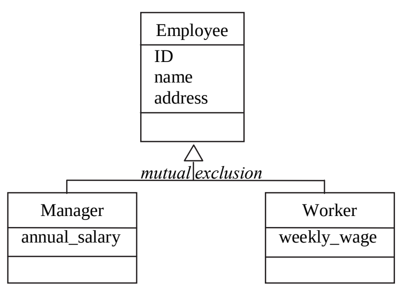


Figure 3.5. Implementation of mutual-exclusion inheritance

```
CREATE TABLE Employee
   (id         VARCHAR2(10) NOT NULL,
    name       VARCHAR2(20),
    address    VARCHAR2(35),
    emp_type   VARCHAR2(8)
      CHECK(emp_type IN ('Manager', 'Worker', NULL)),
    PRIMARY KEY (id));

CREATE TABLE Manager
   (id               VARCHAR2(10) NOT NULL,
    annual_salary    NUMBER,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Employee (id)
    ON DELETE CASCADE);

CREATE TABLE Worker
   (id               VARCHAR2(10) NOT NULL,
    weekly_wage      NUMBER,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Employee (id)
    ON DELETE CASCADE);
```

Figure 3.6. Implementation of mutual-exclusion inheritance using "under"

```
CREATE OR REPLACE TYPE Employee_T AS OBJECT
   (id         VARCHAR2(10),
    name       VARCHAR2(20),
    address    VARCHAR2(35),
    emp_type   VARCHAR2(8)) NOT FINAL
   /

CREATE TABLE Employee OF Employee_T
   (id   NOT NULL,
    emp_type CHECK (emp_type in ('Manager', 'Worker', 'NULL')),
    PRIMARY KEY (id));

CREATE OR REPLACE TYPE Manager_T UNDER Employee_T
   (annual_salary    NUMBER)
   /

CREATE OR REPLACE TYPE Worker_T UNDER Employee_T
   (weekly_wage      NUMBER)
   /
```

Using the newer Oracle™ version for the same example, we can create Manager and Worker subclasses under the superclass Employee (see Figure 3.6). Notice that in this type of inheritance, we create only one table for the superclass. We do not need subclass tables because an object can be a member of only one subclass. These subclasses are instantiations of the superclass. Also notice that although the table is created from the superclass table, Oracle™ maintains the integrity constraint between the subclass and the superclass table. We cannot delete the subclass while the superclass table still exists.

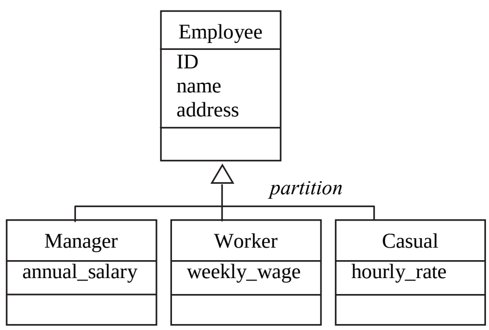
In this case, an employee can only be a manager, a worker, or neither. If an employee is neither a manager nor a worker, he or she is only an object of the superclass, Employee. If an employee is a manager, for example, he or she will be an object of the subclass Manager. Thus, the employee will have all of the attributes of the Manager type and all other attributes that are inherited from the Employee type. However, all of the subclass tables can be kept in the superclass table.

Partition Inheritance Implementation

Partition inheritance declares that a group of subclasses partitions a superclass. A partition requires that the partitioning sets be pairwise disjointed and that their union constitute the partitioned set. Therefore, a partition type can be said to be a combination of both union and mutual-exclusion types. Figure 3.7 shows an example of a partition type of inheritance. We use the example of an employee again, but here a new class, Casual, is added, and it is assumed that each member of the Employee class must belong to one and only one of the classes Manager, Worker, and Casual. For example, an employee cannot be both a manager and a casual.

Similar to the other types of inheritance, the best way to map the partition type of inheritance into tables is to have one table for each superclass and one for

Figure 3.7. Partition inheritance



each subclass. Like the mutual-exclusion type, a new attribute emp_type is added to the superclass table. The difference is that this new attribute has a constraint, which is "not null." This will ensure that each superclass object belongs to a particular subclass type. It also ensures that no superclass object belongs to more than one subclass. Figure 3.8 shows an example of the implementation of partition inheritance. Notice that the attribute emp_type is also needed in the Employee table with the "not null" constraint.

The newer Oracle™ version can also accommodate this inheritance type. It is very similar to the implementation in the mutual-exclusion type. The only difference is the constraint of emp_type in the Employee table as is shown in Figure 3.9.

Multiple Inheritance Implementation

The last type of inheritance relationship is called multiple inheritance. Figure 3.10 gives an example of multiple inheritance. A Tutor class can be said to be inheriting from overlapping classes because basically a tutor can be a student who is also a staff member.

The best way to handle this inheritance from overlapping classes is to use one table for each superclass and one table for the subclass. Figure 3.11 gives an example of a multiple inheritance implementation. A Tutor class can be said to be inheriting from overlapping classes Student and Staff.

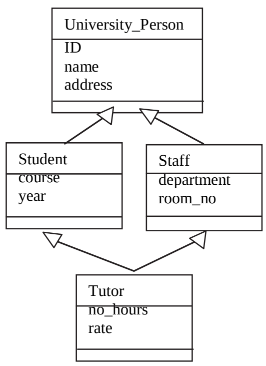
Figure 3.8. Implementation of partition inheritance

```
CREATE TABLE Employee
   (id         VARCHAR2(10) NOT NULL,
    name       VARCHAR2(20),
    address    VARCHAR2(35),
    emp_type   VARCHAR2(8) NOT NULL
       CHECK(emp_type IN ('Manager', 'Worker' ,'Casual')),
    PRIMARY KEY (id));

CREATE TABLE Manager same as in mutual exclusive inheritance
CREATE TABLE Worker  same as in mutual exclusive inheritance

CREATE TABLE Casual
   (id               VARCHAR2(10) NOT NULL,
    hourly_rate      NUMBER,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Employee (id) ON DELETE CASCADE);
```

Figure 3.9. Implementation of partition inheritance relationship using "under"

```
CREATE OR REPLACE TYPE Employee_T AS OBJECT
   (id          VARCHAR2(10),
    name        VARCHAR2(20),
    address     VARCHAR2(35),
    emp_type    VARCHAR2(8)) NOT FINAL
   /

CREATE TABLE Employee OF Employee_T
   (id NOT NULL,
    emp_type NOT NULL
    CHECK (emp_type in ('Manager', 'Worker', 'Casual')),
    PRIMARY KEY (id));

CREATE TYPE Manager_T      same as in mutual exclusive inheritance
CREATE TYPE Worker_T       same as in mutual exclusive inheritance

CREATE OR REPLACE TYPE Casual_T UNDER Employee_T
   (hourly_rate      NUMBER)
/
```

Figure 3.10. Multiple inheritance



At the time of this writing, the newer Oracle™ does not support multiple inheritance using the "under" keyword. This keyword is applicable only to the single inheritance type. However, this multiple inheritance concept is often simulated using other existing techniques. For example, we can use the "under" keyword to implement one inherited parent, and use an association type to link to the other parent. The drawback of using this technique is that only the parent type implemented using "under" can be inherited, and therefore we have to be careful when choosing which parent to inherit and which one to associate.

Figure 3.11. Implementation of multiple inheritance relationship

```
CREATE TABLE Person
   (id         VARCHAR2(10) NOT NULL,
    name       VARCHAR2(20),
    address    VARCHAR2(35),
    PRIMARY KEY (id));

CREATE TABLE Student
   (id         VARCHAR2(10) NOT NULL,
    course     VARCHAR2(10),
    year       VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person (id) ON DELETE CASCADE);

CREATE TABLE Staff
   (id         VARCHAR2(10) NOT NULL,
    department VARCHAR2(10),
    room_no    VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person (id) ON DELETE CASCADE);

CREATE TABLE Tutor
   (id         VARCHAR2(10) NOT NULL,
    no_hours   NUMBER,
    rate       NUMBER,
    PRIMARY KEY (id)
    FOREIGN KEY (id) REFERENCES Person (id) ON DELETE CASCADE);
```

Using Association Relationships

Relational data structures can be related to the concepts of sets through the fact that tuples are not in any particular order and duplicate tuples are not allowed. Therefore, the implementation of association relationships with a set semantic into object-relational tables is identical to the well-known transformation of many-to-many or one-to-many relationships from relational modeling to relational tables.

In relational modeling, many-to-many relationships are converted into tables in which the primary key is a composite key obtained from the participating entities. Should there be any attributes of the relationships, these will automatically be added to the tables that represent the many-to-many relationships. Likewise, in object modeling, if a class has a set relationship with another class and the inverse relationship is also a set, the transformation of such an association is identical to the many-to-many relationships' transformation from relational modeling to relational tables where a table is created to represent the set relationship. This transformation strategy also enforces that each element within a set cannot be duplicated, which is realized by the implementation of the composite primary key of the relationship tables.

In one-to-many relationships, as in relational modeling, the primary key of the one side is copied to the many side to become a foreign key. In other words, there is no special treatment necessary for the transformation of association relationships having a set semantic.

In Oracle™, there are two ways of implementing an association relationship: by primary-key and foreign-key relationships and by object references. Each of these methods will be described as follows.

Creating an Association Relationship by a Primary-Key and Foreign-Key Relationship

This first method is the traditional relational implementation of connecting two or more tables together. The placement of the foreign keys is based on the cardinality of the association relationship, whether it is one to one, one to many, or many to many. We will use the following object-oriented diagram to show the implementation of association relationships.

The first association between Student and Course is a many-to-many relationship. A third table needs to be created to keep the relationship between the two

Figure 3.12. Object-oriented diagram for association relationships

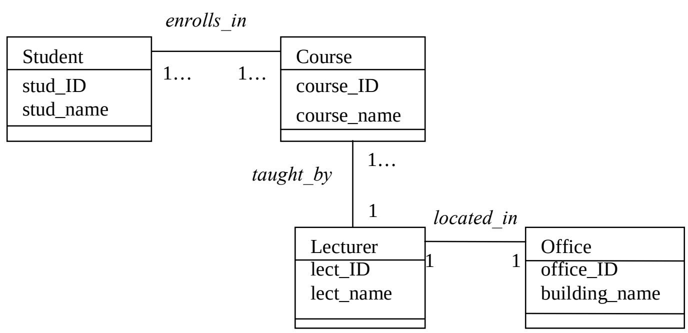


Figure 3.13. Implementation of many to many using a primary-key and foreign-key relationship

```
CREATE TABLE Course
   (course_id        VARCHAR2(10) NOT NULL,
    course_name      VARCHAR2(20),
    PRIMARY KEY (course_id));

CREATE TABLE Student
   (stud_id    VARCHAR2(10) NOT NULL,
    stud_name  VARCHAR2(20),
    PRIMARY KEY (stud_id));

CREATE TABLE Enrolls_in
   (course_id  VARCHAR2(10) NOT NULL,
    stud_id    VARCHAR2(10) NOT NULL,
    PRIMARY KEY (course_id, stud_id),
    FOREIGN KEY (course_id) REFERENCES Course (course_id)
    ON DELETE CASCADE,
    FOREIGN KEY (stud_id) REFERENCES Student (stud_id)
    ON DELETE CASCADE);
```

Figure 3.14. Implementation of one to many using a primary-key and foreign-key relationship

```
CREATE TABLE Lecturer
   (lect_id    VARCHAR2(10) NOT NULL,
    lect_name  VARCHAR2(20),
    PRIMARY KEY (lect_id));

CREATE TABLE Course
   (course_id        VARCHAR2(10) NOT NULL,
    course_name      VARCHAR2(20),
    lect_id          VARCHAR(10),
    PRIMARY KEY (course_id),
    FOREIGN KEY (lect_id) REFERENCES Lecturer (lect_id)
    ON DELETE CASCADE);
```

connected tables. This table will have the primary keys of the connected tables as its primary (composite) key. Each of the primary keys, which form the composite, is connected to the originated table through a primary-key and foreign-key relationship.

The second association is a one-to-many relationship between Lecturer and Course. In order to establish the association relationship in the implementation, the primary key of the one side, Lecture, becomes a foreign key of the table that holds the many side, Course.

Figure 3.15. Implementation of one-to-one using a primary-key and foreign-key relationship

```
CREATE TABLE Office
   (office_id       VARCHAR2(10) NOT NULL,
    building_name   VARCHAR2(20),
    PRIMARY KEY (office_id));

CREATE TABLE Lecturer
   (lect_id    VARCHAR2(10) NOT NULL,
    lect_name  VARCHAR2(20),
    office_id  VARCHAR2(10),
    PRIMARY KEY (lect_id),
    FOREIGN KEY (office_id) REFERENCES Office (office_id)
    ON DELETE CASCADE);
```

The third association is a one-to-one relationship between Lecturer and Office. In this type of relationship, one has to decide the *participation constraint* between the two connected tables (Elmasri & Navathe, 2000). There are two types of participation constraints, namely, *total* and *partial*. In the above example, every lecturer must be located in one particular office; thus, the participation of the lecturer in the relationship is total. On the other hand, one particular office may be vacant; no particular lecturer has been assigned the room. In this case, the participation of the office in the relationship is partial. In order to establish the association relationship in the implementation, the primary key of the table with partial participation, Office, becomes a foreign key of the table that holds the total participation, Lecturer.

Creating an Association Relationship by Object References

Another implementation method of association relationships in Oracle™ is using object references. Instead of connecting two tables through the values of the associated primary key and foreign key, this method allows one to directly connect two tables through the referencing attribute. Thus, the associated attribute that connects the two tables is not holding a value of the primary key of the other connected table, but a reference of where the connected table is actually stored.

Figure 3.16. Implementation of many to many using object references

```
CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
   /

CREATE OR REPLACE TYPE Course_T AS OBJECT
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(30))
   /

CREATE TABLE Student OF Person_T
   (person_id NOT NULL,
    PRIMARY KEY (person_id));

CREATE TABLE Course OF Course_T
   (course_id NOT NULL,
    PRIMARY KEY (course_id));

CREATE TABLE Enrolls_in
   (student     REF Person_T,
   course       REF Course_T);
```

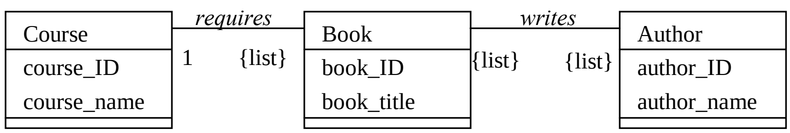
Figure 3.17. Implementation of one to many using object references

```
CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
   /

CREATE OR REPLACE TYPE Course_T AS OBJECT
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(30),
    course_lecturer  REF Person_T)
   /

CREATE TABLE Lecturer OF Person_T
   (person_id NOT NULL,
    PRIMARY KEY (person_id));

CREATE TABLE Course OF Course_T
   (course_id NOT NULL,
    PRIMARY KEY (course_id));
```

Figure 3.18. Implementation of one to one using object references

```
CREATE OR REPLACE TYPE Office_T AS OBJECT
   (office_id        VARCHAR2(10),
    building_name    VARCHAR2(20))
   /

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30),
    person_office    REF Office_T)
   /

CREATE TABLE Office OF Office_T
   (office_id NOT NULL,
    PRIMARY KEY (office_id));

CREATE TABLE Lecturer OF Person_T
   (person_id NOT NULL,
   PRIMARY KEY (person_id));
```

Figure 3.19. Association example using collection types



The following figures show the implementation of many-to-many, one-to-many, and one-to-one relationships of the example in Figure 3.17 using object references.

In some cases, we want to have additional semantics at the many side, for example, by incorporating an ordering semantic. To show more implementation examples of association relationships involving collection types, we will extend the example in Figure 3.12. The additional classes are shown in Figure 3.19. Every course will require a list of books as references. The class Book is also associated with a list of authors. Note that in this example we use the term *list* to represent an ordered collection as opposed to the earlier example of *set* for an unordered collection.

Figure 3.20. Implementation of one to list using object references

```
CREATE OR REPLACE TYPE Course_T AS OBJECT
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(30))
/

CREATE OR REPLACE TYPE Book_T AS OBJECT
   (book_id          VARCHAR2(10),
    book_title       VARCHAR2(30),
    course_book      REF Course_T)
/

CREATE TABLE Course OF Course_T
   (course_id NOT NULL,
    PRIMARY KEY (course_id));

CREATE TABLE Book OF Book_T
   (book_id NOT NULL,
    PRIMARY KEY (book_id));

CREATE TABLE Require
   (Book             REF   Book_T,
    Index_Book       NUMBER NOT NULL,
    Course           REF   Course_T);
```

The following figures show the implementation of one-to-list and list-to-list relationships of this example using object references. Note that we have the attribute Index_Book in table Require because we need the ordering semantic of the book associated with a specific course.

The main difference between a list implementation and the earlier many-to-many association is the need to add one index attribute (e.g., Index_Author in Figure 3.21). This index will maintain the ordering semantic within the list.

Primary Keys: Foreign Keys vs. Object References in an Association Relationship

An association relationship uses keys to provide a solid referential integrity constraint. As mentioned earlier, we can add constraints (cascade, restrict, and nullify) by either using the Oracle™ system-defined constraints or by triggers. With the referential integrity constraints, there will be an automatic check on the table that is being referenced before data manipulation is performed. On the

Figure 3.21. Implementation of list to list using object references

```
CREATE OR REPLACE TYPE Book_T AS OBJECT
   (book_id          VARCHAR2(10),
    book_title       VARCHAR2(30))
/

CREATE OR REPLACE TYPE Author_T AS OBJECT
   (author_id        VARCHAR2(10),
    author_name      VARCHAR2(30))
/

CREATE TABLE Book OF Book_T
   (book_id NOT NULL,
    PRIMARY KEY (book_id));

CREATE TABLE Author OF Author_T
   (author_id NOT NULL,
    PRIMARY KEY (author_id));

CREATE TABLE Write
   (Book             REF Book_T,
    Index_Book       NUMBER NOT NULL,
    Author           REF Author_T);

CREATE TABLE Written_By
   (Author           REF Author_T,
    Index_Author     NUMBER NOT NULL,
    Book             REF Book_T);
```

other hand, using the object reference ref, there is no referential integrity constraint performed. There is the possibility for an object reference to be *dangling* if the object it refers to has been accidentally deleted.

One suggestion to avoid this is by applying a foreign key to the object-reference concept. For example, recalling Figure 3.18, we can create a new version to add referential integrity into the object reference (see Figure 3.22).

Figure 3.22. Implementation of one to one using ref and references

```
CREATE TYPE Office_T -- same as in Figure 3.18
CREATE TYPE Person_T -- same as in Figure 3.18
CREATE TABLE Office -- same as in Figure 3.18

CREATE TABLE Lecturer OF Person_T
   (person_id  NOT NULL,
    PRIMARY KEY (person_id),
    FOREIGN KEY (person_office) REFERENCES Office
    ON DELETE CASCADE);
```

Using Aggregation Relationships

There are two techniques that can be used in Oracle™ in order to simulate the implementation of aggregations: the clustering technique and the nesting technique.

Implementing Existence-Dependent Aggregation using the Clustering Technique

In this section we use an example of a homogeneous aggregation relationship between Hard Disk (HD) and HD Controller (HD_Contr; see Figure 1.13). The Oracle™ implementation of this type of aggregation using the clustering technique is shown in Figure 3.23.

It is clear from the implementation that the clustering technique supports only an existence- dependent aggregation. It is not possible to have an HD controller (part object) that does not belong to an HD (whole object). This is enforced by the existence of the cluster key in all the part tables. Moreover, the example in Figure 3.23 also shows a nonexclusive aggregation type, where each part

Figure 3.23. Implementation of existence-dependent aggregation using the clustering technique

```
CREATE CLUSTER HD_Cluster
   (hd_id            VARCHAR2(10));

CREATE TABLE Hard_Disk
   (hd_id            VARCHAR2(10) NOT NULL,
    capacity         VARCHAR2(20),
    PRIMARY KEY (hd_id))
   CLUSTER HD_Cluster(hd_id);

CREATE TABLE HD_Contr
   (hd_id            VARCHAR2(10) NOT NULL,
    hd_contr_id      VARCHAR2(10) NOT NULL,
    description      VARCHAR2(25),
    PRIMARY KEY (hd_id, hd_contr_id),
    FOREIGN KEY (hd_id) REFERENCES Hard_Disk (hd_id))
   CLUSTER HD_Cluster(hd_id);

CREATE INDEX HD_Cluster_Index
ON CLUSTER HD_Cluster;
```

Figure 3.24. Implementation of exclusive aggregation using the clustering technique

```
CREATE TABLE Hard_Disk
   (hd_id            VARCHAR2(10) NOT NULL,
    capacity         VARCHAR2(20),
    PRIMARY KEY (hd_id))
   CLUSTER HD_Cluster(hd_id);

CREATE TABLE HD_Contr
   (hd_id            VARCHAR2(10) NOT NULL,
    hd_contr_id      VARCHAR2(10) NOT NULL,
    description      VARCHAR2(25),
    PRIMARY KEY (hd_contr_id),
    FOREIGN KEY (hd_id) REFERENCES Hard_Disk (hd_id))
   CLUSTER HD_Cluster(hd_id);
```

object can be owned by more than one whole object. For example, HD controller HDC1 may belong to HD1 as well as HD2.

Depending on the situation, the above nonexclusive type may not be desirable. We can enforce the aggregation-exclusive type by creating a single primary key for the part object and treating the cluster key as a foreign key rather than as part of the primary key. Figure 3.24 shows the implementation of the previous example as an exclusive type (the implementation of the cluster and the cluster index remain the same).

Each time a new record is inserted into the part table, HD_Contr, the value of the cluster key, hd_id, is searched for. If it is found, the new record will be added to the cluster. The rows of the whole table, Hard_Disk, and the rows of the part table, HD_Contr, are actually stored together physically (see Figure 3.25). The index is created in order to enhance the performance of the cluster storage.

Figure 3.25. Physical storage of the aggregation relationship using cluster

hd_id	capacity	hd_contr_id	description
HD11	2GB	Contr111	....
		Contr112	....
HD12	6GB	Contr121	....
		Contr122	....
		Contr123	....

Figure 3.26. Implementation of an aggregation relationship with multiple part objects

```
CREATE CLUSTER PC_Cluster
   (pc_id      VARCHAR2(10));

CREATE TABLE PC
   (pc_id      VARCHAR2(10) NOT NULL,
    type       VARCHAR2(20),
    PRIMARY KEY (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE TABLE Hard_Disk
   (pc_id      VARCHAR2(10) NOT NULL,
    hd_id      VARCHAR2(10) NOT NULL,
    capacity   VARCHAR2(20),
    PRIMARY KEY (pc_id, hd_id),
    FOREIGN KEY (pc_id) REFERENCES PC (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE TABLE Monitor
   (pc_id      VARCHAR2(10) NOT NULL,
    monitor_id VARCHAR2(10) NOT NULL,
    resolution VARCHAR2(25),
    PRIMARY KEY (pc_id, monitor_id),
    FOREIGN KEY (pc_id) REFERENCES PC (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE TABLE Keyboard
   (PC_id            VARCHAR2(10) NOT NULL,
    keyboard_id      VARCHAR2(10) NOT NULL,
    type             VARCHAR2(25),
    PRIMARY KEY (pc_id, keyboard_id),
    FOREIGN KEY (pc_id) REFERENCES PC (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE TABLE CPU
   (pc_id      VARCHAR2(10) NOT NULL,
    cpu_id     VARCHAR2(10) NOT NULL,
    speed      VARCHAR2(10),
    PRIMARY KEY (pc_id, cpu_id),
    FOREIGN KEY (pc_id) REFERENCES PC (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE INDEX PC_Cluster_Index
   ON CLUSTER PC_Cluster;
```

It is also possible to use the cluster method to implement an aggregation relationship between a whole object with a number of part objects. Figure 3.26 demonstrates the implementation of an aggregation between a PC with Hard_Disk, Monitor, Keyboard, and CPU (see Figure 1.8 in Chapter I).

Figure 3.27. Physical storage of multiple-aggregation relationships using cluster

whole id	whole attribute	part ID	part attribute
PC001	....	HardDisk1	....
		HardDisk1	....
		Monitor11	....
		Keyboard	....
		CPU11	....
PC002	....	HardDisk2	....
		Monitor21	
		Keyboard	
		CPU21	....

Figure 3.27 shows the physical storage of the multiple aggregation relationship between a PC with Hard_Disk, Monitor, Keyboard, and CPU.

Implementing Existence-Dependent Aggregation using the Nesting Technique

Another Oracle™ implementation technique for aggregation involves using nested tables. In this technique, similar to the clustering one, the part information is tightly coupled with the information of the whole object and it is implemented as a nested table. This actually enforces the aggregation existence-dependent type. If the data of the whole object is removed, all associated part objects will need to be removed as well. Moreover, the data in the part nested table is normally accessed through the whole object only. Because of this, this nested-table technique is suitable only for the implementation of the aggregation existence-dependent type.

Figure 3.28 describes the link between the whole and the part table in a nesting structure, whereas Figure 3.29 shows the implementation of the homogenous aggregation depicted in Figure 1.13 using the nested-table technique.

Note that there is neither the concept of a primary key nor the integrity constraint in the part nested table as shown in Figure 3.28. For example, if a particular HD controller is used by another HD from the whole table, then all the details of the HD controller will be written again as a separate record within the nested table.

Figure 3.28. Aggregation relationships using a nested table

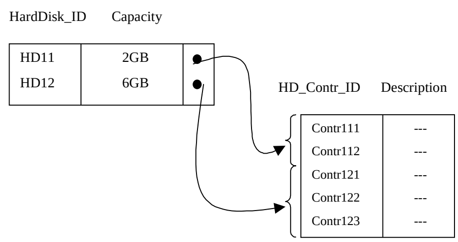


Figure 3.29. Implementation of aggregation relationships using nested tables

```
CREATE OR REPLACE TYPE HD_Contr AS OBJECT
   (hd_contr_id      VARCHAR2(10),
    description      VARCHAR2(30));
/

CREATE OR REPLACE TYPE HD_Contr_Table AS TABLE OF HD_Contr
/

CREATE TABLE Hard_Disk
   (hd_id      VARCHAR2(10) NOT NULL,
    capacity   VARCHAR2(20),
    controller HD_Contr_Table,
    PRIMARY KEY (hd_id))
  NESTED TABLE controller STORE AS HD_Contr_tab;
```

Oracle™ also facilitates multilevel nested tables and thus can be used for implementing a multilevel aggregation relationship. It is implemented by using the inner and outer table principle (see Figure 3.30). A PC is an aggregation of several HDs, and a HD is an aggregation of several HD controllers. In this case, the inner table is a nested table of HD controller, and the outer table is a nested table of HD. The implementation of this aggregation is shown in Figure 3.31.

Note in the implementation (see Figure 3.29 and Figure 3.31) that we do not create standard tables for the HD controller. We only need to define a HD controller type, and define it as a nested table later when we create the Hard Disk table (for Figure 3.29) and the PC table (for Figure 3.31). It is also shown that the information of the nested table is stored externally in a table called HD_Contr_tab. This is not a standard table; no additional constraints can be

Figure 3.30. Multilevel aggregation relationships using nested tables

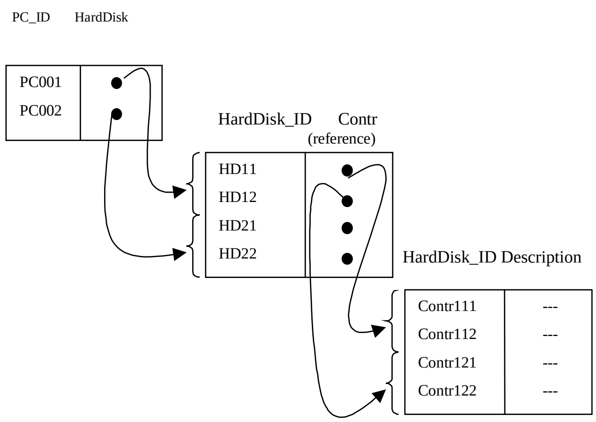


Figure 3.31. Implementation of multilevel aggregation relationships using nested tables

```
Example:

CREATE OR REPLACE TYPE HD_Contr AS OBJECT
   (hd_contr_id      VARCHAR2(10),
    description      VARCHAR2(30))
   /

CREATE OR REPLACE TYPE HD_Contr_Table AS TABLE OF HD_Contr
/

CREATE OR REPLACE TYPE Hard_Disk AS OBJECT
   (hd_id      VARCHAR2(10),
    capacity   VARCHAR2(20),
    controller HD_Contr_Table)
   /

CREATE OR REPLACE TYPE Hard_Disk_Table AS TABLE OF Hard_Disk
/

CREATE TABLE PC
   (pc_id      VARCHAR2(10) NOT NULL,
    hd         Hard_Disk_Table,
    PRIMARY KEY (pc_id))
   NESTED TABLE hd STORE AS HD_tab
      (NESTED TABLE controller STORE AS HD_Contr_tab);
```

attached to this table and no direct access can be performed to this table without going through the Hard Disk table.

Every whole object can own any part object in the nesting technique, even if that particular part has been owned by another whole object. The record of the HD_Contr object will simply be repeated every time a hard disk claims to own it. This shows a nonexclusive type of aggregation, where a particular part object can be shared by more than one whole object.

Because there is no standard table created for the HD controller, we cannot have a primary key for the table, which we usually employ to enforce an exclusive type of aggregation (see the previous clustering technique).

It is clear from the above sections on clustering and nesting techniques that these techniques are suitable only for the implementation of the existence-dependent type of aggregation. The clustering technique supports both nonexclusive and exclusive aggregation. However, the nesting technique supports only the nonexclusive type.

In the following section we will see how we can implement an existence-independent type of aggregation.

Implementing Existence-Independent Aggregation

To implement the existence-independent aggregation type in relational tables, an Aggregate table is created. This table maintains the part-of relationship between the whole table and the part tables. By having one Aggregate table, we avoid having a link from the whole to the part that is hard coded within one of the tables. In both the clustering and nesting techniques, the connection between whole and part is either hard coded within the whole table (in the nesting technique) or within the part tables (in the clustering technique). These techniques actually prevent us from creating independent part objects that exist but are not necessarily connected to a particular whole at any given time.

In the Aggregate table, only the relationships between the identifiers of the whole table and the part tables are stored. To maintain consistency in the Aggregate table, the identifiers across different part tables should be kept unique. If the number of the part tables is more than one, a new attribute type is used to distinguish the different types of the part tables.

Figure 3.32 shows an existence-independent aggregation, where lab is an aggregate of Computer, Printer, and Scanner. There are times when we have

Figure 3.32. Existence-independent type of aggregation

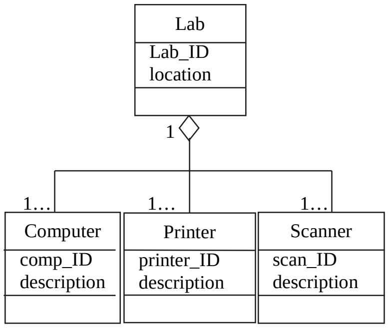
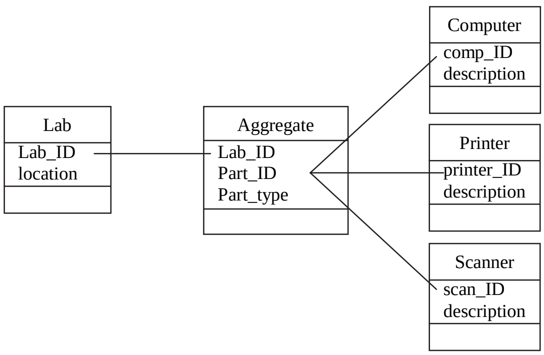


Figure 3.33. Existence-independent type of aggregation using the Aggregate table



new computers or printers that have not been allocated to any particular lab. We want to still be able to keep the record of the new parts even when no associated whole is established.

This situation cannot be implemented using either the clustering or the nested technique. In the nesting technique, we can only insert a new part record within the nested table if we have an existing whole record for it. In the clustering technique, the primary key of the whole serves as the cluster key; thus, it is not supposed to be null.

Figure 3.33 shows how an Aggregate table is created to store the relationship between Lab and Computer, Printer, and Scanner. The Aggregate table contains the primary key of the whole, which is lab_ID, and an attribute called part_ID, which is the primary key of either one of the part tables (comp_ID, printer_ID, or scan_ID). The last attribute is called part_type, which is the type

Figure 3.34. Implementation of the existence-independent type of aggregation

```
CREATE TABLE Lab
   (lab_id           VARCHAR2(10) NOT NULL,
    location         VARCHAR2(20),
    PRIMARY KEY (lab_id));

CREATE TABLE Computer
   (comp_id          VARCHAR2(10) NOT NULL,
    description      VARCHAR2(10),
    PRIMARY KEY (comp_id));

CREATE TABLE Printer
   (printer_id       VARCHAR2(10) NOT NULL,
    description      VARCHAR2(10),
    PRIMARY KEY (printer_id));

CREATE TABLE Scanner
   (scan_id          VARCHAR2(10) NOT NULL,
    description      VARCHAR2(10),
    PRIMARY KEY (scan_id));

CREATE TABLE Aggregate
   (lab_id           VARCHAR2(10) NOT NULL,
    part_id          VARCHAR2(10),
    part_type        VARCHAR2(20)
      CHECK (part_type in ('Computer', 'Printer', 'Scanner')),
    PRIMARY KEY (lab_id, part_id),
    FOREIGN KEY (lab_id) REFERENCES Lab(lab_id));
```

of the part_ID (computer, printer, or scanner). Figure 3.34 demonstrates the implementation of the above aggregation structure.

Figure 3.34 shows an implementation of the existence–independent, nonexclusive aggregation type. It is an existence-independent type because the new records of part tables, Computer, Printer, and Scanner, can be inserted without any associated record within the whole table, Lab. If a Lab record is removed from the Lab table, it will only be cascaded to the Aggregate table where the specific Lab record appears; however, it does not have to affect the records within the associated part tables. The above example is also a nonexclusive type because one particular part, such as a printer, can appear in the Aggregate table more than once and is associated with a different lab_ID. This is possible because both the lab_ID and part_ID are primary keys of the Aggregate table. If this situation is not desirable, then we can make the lab_ID a foreign key in the Aggregate table, and only the part_ID will be the primary key. This will enforce each part_ID to appear only once within the Aggregate table and be associated with one particular lab_ID only.

Case Study

The following course-manual authorship case study shows how we can implement an object-oriented model in Oracle™. The diagram shows two inheritance relationships. First is the union inheritance between an author and an industry-based author and an academic author. Second is the mutual-exclusion inheritance between an academic and a research staff and a teaching staff. There are association relationships between the author and course manual, as well as between the teaching staff and subject. There is also one aggregation relationship between the course manual and its chapters.

To implement the course-manual authorship object-oriented model into Oracle™, we will apply the following systematic steps: type and table. We have

Figure 3.35. Course-manual authorship case study

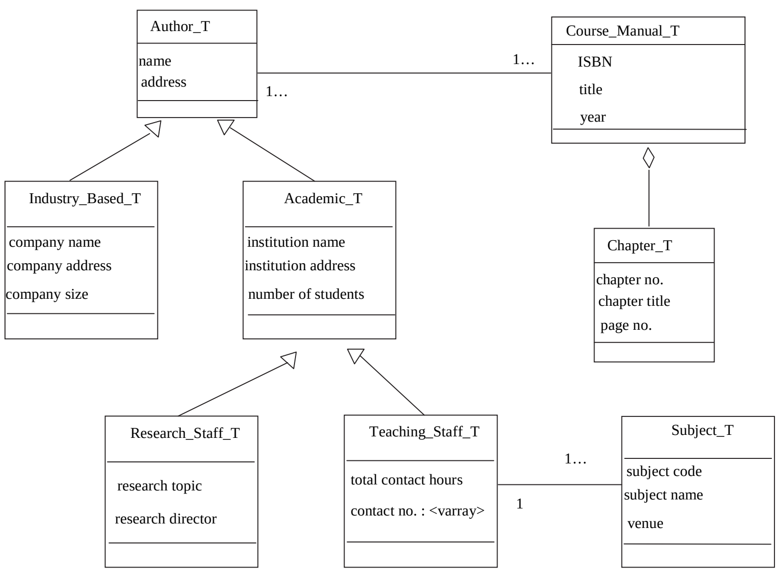


Figure 3.36. Implementation of the case study in Oracle™

```
CREATE OR REPLACE TYPE Author_T AS OBJECT
   (ao_id          VARCHAR2(3),
    name           VARCHAR2(10),
    address        VARCHAR2(20)) NOT FINAL
   /

CREATE OR REPLACE TYPE Industry_Based_T UNDER Author_T
   (c_name         VARCHAR2(10),
    c_address      VARCHAR2(20),
    c_size         VARCHAR2(10))
   /

CREATE OR REPLACE TYPE Academic_T UNDER Author_T
   (i_name         VARCHAR2(10),
    i_address      VARCHAR2(20),
    no_student     NUMBER,
    academic_type  VARCHAR2(20)) NOT FINAL
   /

CREATE OR REPLACE TYPE Research_Staff_T UNDER Academic_T
   (topic          VARCHAR2(20),
    director       VARCHAR2(10))
   /

CREATE OR REPLACE TYPE Contacts AS VARRAY(3) OF NUMBER
/

CREATE OR REPLACE TYPE Teaching_Staff_T UNDER Academic_T
   (total_hour     NUMBER,
    contact_no     Contacts)
   /

CREATE TABLE Author OF Author_T
    (ao_id NOT NULL,
     PRIMARY KEY (ao_id));

-- implementation of inheritance using an earlier version of Oracle
-- or traditional relational databases

CREATE OR REPLACE TYPE Author_T AS OBJECT
   (ao_id          VARCHAR2(3),
    name           VARCHAR2(10),
    address        VARCHAR2(20))
   /
```

Figure 3.36. (continued)

```
CREATE OR REPLACE TYPE Industry_Based_T AS OBJECT
  (ao_id          VARCHAR2(3),
   c_name         VARCHAR2(10),
   c_address            VARCHAR2(20),
   c_size         VARCHAR2(10))
  /
CREATE OR REPLACE TYPE Academic_T AS OBJECT
  (ao_id          VARCHAR2(3),
   i_name         VARCHAR2(10),
   i_address            VARCHAR2(20),
   no_student     NUMBER,
   academic_type  VARCHAR2(20))
  /
 CREATE OR REPLACE TYPE Research_Staff_T AS OBJECT
  (ao_id          VARCHAR2(3),
   topic          VARCHAR2(20),
   director       VARCHAR2(10))
  /
CREATE OR REPLACE TYPE Contacts AS VARRAY(3) OF NUMBER
 /
CREATE OR REPLACE TYPE Teaching_Staff_T AS OBJECT
  (ao_id          VARCHAR2(3),
   total_hour     NUMBER,
   contact_no     Contacts)
  /
 CREATE TABLE Author OF Author_T
    (ao_id NOT NULL,
     PRIMARY KEY (ao_id));

CREATE TABLE Industry_Based OF Industry_Based_T
    (ao_id NOT NULL,
     PRIMARY KEY (ao_id),
     FOREIGN KEY (ao_id) REFERENCES author (ao_id)
     ON DELETE CASCADE);

CREATE TABLE Academic OF Academic_T
    (ao_id NOT NULL,
     academic_type
     CHECK (academic_type IN ('Research', 'Teaching', NULL)),
     PRIMARY KEY (ao_id),
     FOREIGN KEY (ao_id) REFERENCES author (ao_id)
     ON DELETE CASCADE);

CREATE TABLE Research_Staff OF Research_Staff_T
    (ao_id NOT NULL,
     PRIMARY KEY (ao_id),
     FOREIGN KEY (ao_id) REFERENCES author (ao_id)
     ON DELETE CASCADE);
```

Figure 3.36. (continued)

```
CREATE TABLE Teaching_Staff OF Teaching_Staff_T
    (ao_id NOT NULL,
     PRIMARY KEY (ao_id),
     FOREIGN KEY (ao_id) REFERENCES author (ao_id)
     ON DELETE CASCADE);

 -- implementation of one-to-many association using ref

CREATE OR REPLACE TYPE Subject_T AS OBJECT
   (code           VARCHAR2(10),
    sub_name       VARCHAR2(20),
    venue          VARCHAR2(10),
    lecturer REF Teaching_Staff_T)
   /
 -- implementation of aggregation using a nesting technique

CREATE OR REPLACE TYPE Chapter_T AS OBJECT
    (c_no           NUMBER,
     c_title        VARCHAR2(20),
     page_no        NUMBER)
 /
CREATE OR REPLACE TYPE Chapter_Table_T AS TABLE OF Chapter_T
/

CREATE OR REPLACE TYPE Course_Manual_T AS OBJECT
    (isbn           VARCHAR2(10),
     title          VARCHAR2(20),
     year           NUMBER,
     chapter              Chapter_Table_T)
 /

 CREATE TABLE Course_Manual OF Course_Manual_T
    (isbn    NOT NULL,
     PRIMARY KEY (isbn));

 -- implementation of the Publish table

 CREATE TABLE Publish
    (author REF Author_T,
     course_manual REF Course_Manual_T);

CREATE TABLE Subject OF Subject_T
    (code NOT NULL,
     PRIMARY KEY (code));
```

to determine types and tables that we will need to implement the model. For this case, we need the types Author_T, Industry_T, Academic_T, Research_Staff_T, Teaching_Staff_T, and Subject_T.

For each of them, we will create the table respectively. We also need a type Contacts for the multiple-collection varray of the contact_no attribute in Teaching_Staff_T. Finally, we will need type Course_Manual_T and its table, and also Chapter_T type and Chapter_Table_T type if we decide to use the nested-table implementation in an aggregation relationship.

- **Inheritance relationship.** There are two inheritance relationships in the model. First is the inheritance between Author_T and the subclasses Industry_T and Academic_T. Second is the inheritance between Academic_T and its subclasses Research_Staff_T and Teaching_Staff_T. We will show two methods of implementing inheritance in our sample solution.
- **Association relationship.** There are two association relationships from this model. The first one is between Author_T and Course_Manual_T. If we use a nested table in implementing the aggregation relationship between Course_Manual_T and Chapter_T, we will be able to create a new table using the ref of Author_T and Course_Manual_T in it. The second association is the relationship between Teaching_Staff_T and Subject_T. As it is a one-to-many association, we will need to use the ref of the one side, in this case Teaching_Staff_T, in the many side, Subject_T.
- **Aggregation relationship.** There is one homogeneous aggregation relationship in this model. If we use a nested table, we have to create the type and type table for the part class, and the type and table for the whole class. If we use the clustering technique, we do not need the type, but we do need to create the cluster beforehand using the primary key of the whole class, Course_Manual, and then create an index after that.
- **Complete solution.** The complete solution is shown in Figure 3.36.

Summary

Object-oriented features such as object types, object identity, object references, and relationships are the new object-based features that have been introduced in Oracle™ to enrich traditional RDBMSs with object-oriented characteristics. Using these new features, complex relationships can be implemented, including different semantics of inheritance, associations among different collection types, and aggregation relationships. Although the latest object-oriented Oracle™ has incorporated various object-model features, it still maintains some basic concepts of the relational model such as data integrity and the simplicity of the implementation.

References

Elmasri, R., & Navathe, S. B. (2000). *Fundamentals of database systems* (3rd ed.). Addison Wesley.

Loney, K., & Koch, G. (2000). *Oracle™ 8i: The complete reference.* Osborne McGraw-Hill.

Loney, K., & Koch, G. (2002). *Oracle™ 9i: The complete reference.* Oracle Press.

ORACLE™ 8. (1997). *Oracle™ 8 product documentation library.* Redwood City, CA: Oracle Corporation.

Urman, S. (2000). *Oracle™ 8i advanced PL/SQL programming.* Oracle Press, Osborne.

Chapter Problems

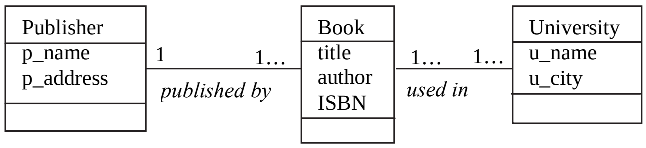
1. A university has a number of books listed as textbooks, each of which may be used by more than one university. A book is published by only one publisher, but one publisher can publish more than one book. Show the implementation of the association relationships above using object refer-

ences. Assume that there are three object types, that is, University, Book, and Publisher. Add any attribute where necessary.

2. The City College has just built a new computer laboratory. It has many PCs in it with their own IDs, capacities, and brands. Although these PCs are currently located in the new laboratory, they are removable to other laboratories or offices. Using the clustering technique, show the implementation of the aggregation relationship described.
3. The Victorian state government stores geographic data in the ranking of aggregation. Data of the state is an aggregation of the area data, and data of the area is an aggregation of the suburb data. For the first implementation, each level contains only an ID and a name as the attributes. Using a nested table, show the implementation of this case.
4. Saving supermarket is preparing many types of food hampers for the Christmas season. Each hamper has its own ID and price. It contains items that can be categorized into biscuit, confectionery, and deli products. Each category has its own ID, name, and price. These part items can be sold as a part of the hamper or sold separately. For this purpose, implement the aggregation relationships as described.
5. The Animal class has attributes ID, name, and description. It has inheritance to three other objects, that is, Fish, Bird, and Mammal. The Fish object has an attribute of its own, water_habitat. The Bird object has attributes color, sound, and fly. Mammal has attributes diet and size. Most of the animals can be allocated to these three objects. However, there is some problem when an animal like a whale is going to be inserted because it can be categorized into two different objects. Show the object-oriented diagram and the implementation for this inheritance relationship.
6. A researcher develops an object-based database for his collection of technical papers. The attributes for the Technical_Papers object are titles and authors. One object inherited from a technical paper is Conference_Paper, which basically contains papers taken from a conference. The attributes for this object are conference name, conference year, and conference venue. To make the database more detailed, he inserted other objects that inherit from Conference_Papers. One of them is OO_Conf_Papers, which contains all conference papers on object-oriented topics. It has its local attribute imp_type. Show the diagram and implementation of the inheritance relationship.

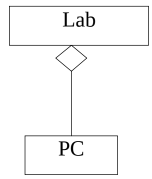
Chapter Solutions

1. The OO diagram for the case is shown below.



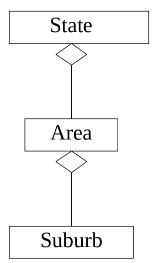

```
CREATE OR REPLACE TYPE Publisher_T AS OBJECT
   (p_id       VARCHAR2(3),
    p_name     VARCHAR2(20),
    p_address  VARCHAR2(50))
/

CREATE OR REPLACE TYPE Book_T AS OBJECT
   (b_id       VARCHAR2(3),
    title      VARCHAR2(50),
    author     VARCHAR2(20),
    isbn       VARCHAR2(10),
    published_by REF Publisher_T)
/
CREATE OR REPLACE TYPE University_T AS OBJECT
   (u_id       VARCHAR2(3),
    u_name     VARCHAR2(20),
    u_city     VARCHAR2(20))
/

CREATE TABLE Publisher OF Publisher_T
   (p_id NOT NULL,
    PRIMARY KEY (p_id));

CREATE TABLE Book OF Book_T
   (b_id NOT NULL,
    PRIMARY KEY (b_id));

CREATE TABLE University OF University_T
   (u_id NOT NULL,
    PRIMARY KEY (u_id));

CREATE TABLE Used_in
   (Book REF Book_T,
    University REF University_T);
```

2. The aggregation is shown below.



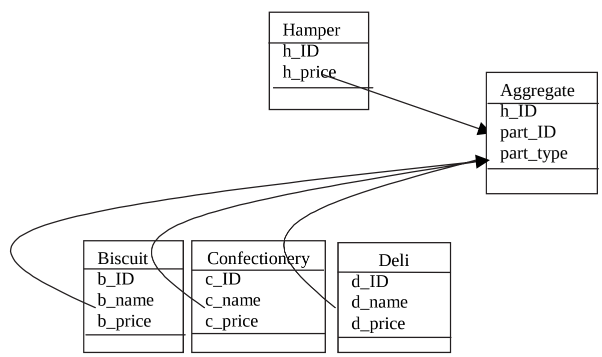

```
CREATE CLUSTER Lab_Cluster
   (lab_id     VARCHAR2(3));

CREATE TABLE Lab
   (lab_id     VARCHAR2(3)  NOT NULL,
    location   VARCHAR2(20),
    PRIMARY KEY (lab_id))
   CLUSTER Lab_Cluster(lab_id);

CREATE TABLE PC
   (lab_id     VARCHAR2(3)  NOT NULL,
    pc_id      VARCHAR2(3)  NOT NULL,
    capacity   VARCHAR2(10),
    brand      VARCHAR2(20),
    PRIMARY KEY (lab_id, pc_id),
    FOREIGN KEY (lab_id) REFERENCES Lab (lab_id))
   CLUSTER Lab_Cluster(lab_id);

CREATE INDEX Lab_Cluster_Index
   ON CLUSTER Lab_Cluster;
```

3. Using a nested table, we need to create the object from the lowest part object. For this case, it starts from suburb, then moves to area and then state.



```
CREATE OR REPLACE TYPE Suburb_T AS OBJECT
   (sb_id       VARCHAR2(3),
    sb_name     VARCHAR2(30))
/

CREATE OR REPLACE TYPE Suburb_Table AS TABLE OF
Suburb_T
/

CREATE OR REPLACE TYPE Area_T AS OBJECT
   (a_id        VARCHAR2(3),
    a_name      VARCHAR2(30),
    suburb      Suburb_Table)
/

CREATE OR REPLACE TYPE Area_Table AS TABLE OF Area_T
/

CREATE TABLE State
   (st_id       VARCHAR2(3) NOT NULL,
    st_name     VARCHAR2(30),
    areas       Area_Table,
    PRIMARY KEY (st_id))
   NESTED TABLE areas STORE AS Area_tab
     (NESTED TABLE suburb STORE AS Suburb_tab);
```

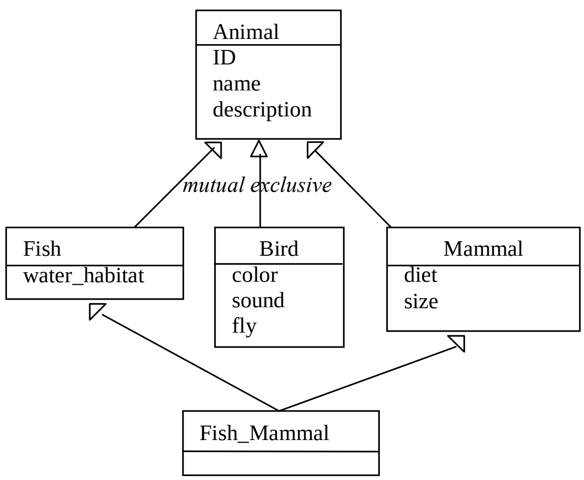
4. To implement the case, we need to create an Aggregate table that stores the whole and the part IDs as the primary keys. The figure below shows the implementation for the case. Part_ID in the Aggregate table is the primary key of each part table, and the part_type is the type of the part itself.




```
CREATE TABLE Hamper
   (h_id        VARCHAR2(3) NOT NULL,
```

```
     h_price    NUMBER,
     PRIMARY KEY (h_id));

CREATE TABLE Biscuit
    (b_id        VARCHAR2(3) NOT NULL,
     b_name      VARCHAR2(20),
     b_price     NUMBER,
     PRIMARY KEY (b_id));

CREATE TABLE Confectionery
    (c_id        VARCHAR2(3) NOT NULL,
     c_name      VARCHAR2(20),
     c_price     NUMBER,
     PRIMARY KEY (c_id));

CREATE TABLE Deli
    (d_id        VARCHAR2(3) NOT NULL,
     d_name      VARCHAR2(20),
     d_price     NUMBER,
     PRIMARY KEY (d_id));

CREATE TABLE Aggregate
    (h_id        VARCHAR2(3) NOT NULL,
     part_id     VARCHAR2(3) NOT NULL,
     part_type   VARCHAR2(20)CHECK (part_type IN
      ('biscuit', 'confectionery', 'deli')),
     PRIMARY KEY (h_id, part_id),
     FOREIGN KEY (h_id) REFERENCES hamper (h_id));
```

5. The diagram for the inheritance is as follows.

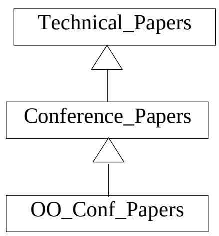


There is a multiple inheritance of class Fish_Mammal that can inherit attributes and methods from two classes, Fish and Mammal.

```
CREATE OR REPLACE TYPE Animal_T AS OBJECT
   (id              VARCHAR2(3),
    name            VARCHAR2(20),
    description     VARCHAR2(50),
    animal_type     VARCHAR2(10))
/

CREATE OR REPLACE TYPE Fish_T AS OBJECT
   (id              VARCHAR2(3),
    water_habitat   VARCHAR2(20))
/

CREATE OR REPLACE TYPE Bird_T AS OBJECT
   (id              VARCHAR2(3),
    color           VARCHAR2(20),
    sound           VARCHAR2(20),
    fly             VARCHAR2(10))
/

CREATE OR REPLACE TYPE Mammal_T AS OBJECT
   (id              VARCHAR2(3),
    diet            VARCHAR2(20),
    m_size          VARCHAR2(10))
/

CREATE OR REPLACE TYPE Fish_Mammal_T AS OBJECT
   (id              VARCHAR2(3),
    lungs_capacity  NUMBER)
/

CREATE TABLE Animal OF Animal_T
   (id NOT NULL,
    PRIMARY KEY (id));

CREATE TABLE Fish OF Fish_T
   (id NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Animal
    ON DELETE CASCADE);

CREATE TABLE Bird OF Bird_T
   (id NOT NULL,
    PRIMARY KEY (id),
```

```
     FOREIGN KEY (id) REFERENCES Animal
     ON DELETE CASCADE);

CREATE TABLE Mammal OF Mammal_T
    (id NOT NULL,
     PRIMARY KEY (id),
     FOREIGN KEY (id) REFERENCES Animal
     ON DELETE CASCADE);

CREATE TABLE Fish_Mammal OF Fish_Mammal_T
    (id NOT NULL,
     PRIMARY KEY (id),
     FOREIGN KEY (id) REFERENCES Animal
     ON DELETE CASCADE);
```

6. The diagram and the implementation of the inheritance case described can be solved by using the Oracle™ inheritance facility.




```
CREATE OR REPLACE TYPE Technical_Papers_T AS OBJECT
    (title      VARCHAR2(30),
     authors    VARCHAR2(20)) NOT FINAL
/

CREATE OR REPLACE TYPE Conference_Papers_T
UNDER Technical_Papers_T
    (conf_name        VARCHAR2(20),
     conf_year        NUMBER,
     conf_venue       VARCHAR2(10))
    NOT FINAL
/

CREATE OR REPLACE TYPE OO_Conf_Papers_T
UNDER Conference_Papers_T
    (imp_type         VARCHAR2(20));
/
```

Chapter IV

Object-Oriented Methods

We recall that an object-oriented model consists of two major aspects: the *static* and *dynamic*. The former covers the implementation of the data structure, which includes the object's attributes and relationships, whereas the latter is concerned with the object's operations, which is the implementation of object-oriented methods using SQL and PL/SQL.

The static and dynamic parts of an object model actually form a nonseparated unit since accesses to the attributes of an object must be done through the available methods. This raises the concept of encapsulation.

In the object-relational database environment, there are two possible mechanisms for implementing encapsulation.

- Encapsulation using stored procedures or functions and the grant mechanism
- Encapsulation using member procedures or functions

The first mechanism has been adopted mostly by pure RDB systems. It allows information hiding by managing the privileges of each method, as well as ensuring correctness and consistency of the database by providing specific methods for accessing the data.

The second mechanism, which is available in object-relational DBMSs such as Oracle™ 8 and above, is called the member procedure or function. This mechanism allows us to define object types with their associated procedures or functions together. Each of the two mechanisms will be described in the following sections.

Implementation of Encapsulation Using Stored Procedures or Functions and Grant Mechanisms

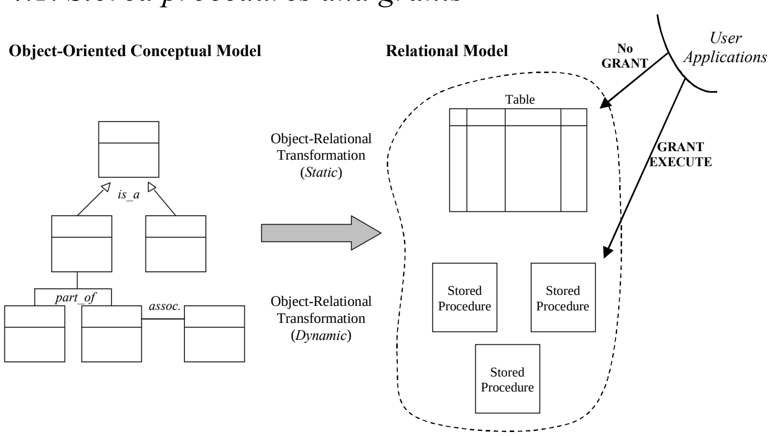
Encapsulation in the relational world is not common, although it may be implemented for the sake of security. We normally simulate encapsulation in RDBs through the use of grants. Figure 4.1 gives an illustration of the overall implementation of an object model into an object-relational system covering the static and dynamic transformation and the use of grants for encapsulation.

In the following sections, we especially consider two aspects for achieving encapsulation using this mechanism, namely, stored procedures or functions for storing generic methods, and grants for maintaining encapsulation.

Stored Procedures or Functions

Stored procedures or functions are PL/SQL programs that are stored in RDBs and subsequently can be invoked at any time. The benefit of stored procedures

Figure 4.1. Stored procedures and grants



or functions is well perceived in a client-server environment as a call to a stored procedure or function can be done in a single call, thereby minimizing network traffic. Another benefit, which is more relevant to our transformation business, is that methods of a class in an object-oriented model can be stored in stored procedures.

In the following sections, we will use the Customer_T object (see Figure 4.2) as a working example. This object has a number of methods. The detail implementation and the parameters will depend on whether the first approach (using the grant mechanism) or the second approach (using the member procedures or functions mechanism) is used.

Section 4.1 shows how to implement methods when a grant mechanism is used to simulate encapsulation in an object-relational database. Section 4.2 will show how the methods are implemented if the member procedures or functions mechanism is chosen.

The first code in Figure 4.3 shows how we implement the Add_Customer method. We will need all necessary attributes as the parameters of the method. Assume that table Customer of type Customer_T has already been created.

In the example of Add_Customer, the parameter types are written as "%type" rather than the usual data types such as number, char, and so forth. When %type is used, the procedure will copy whatever data types are used for the associated attributes in the specified table. For example, the parameter new_ID will use the data type of attribute ID in the Customer table, and so on.

The example in Figure 4.4 shows a stored procedure to update a customer's total bonus points. The method also handles an exception, where the customer

Figure 4.2. Two implementation techniques for Customer_T methods

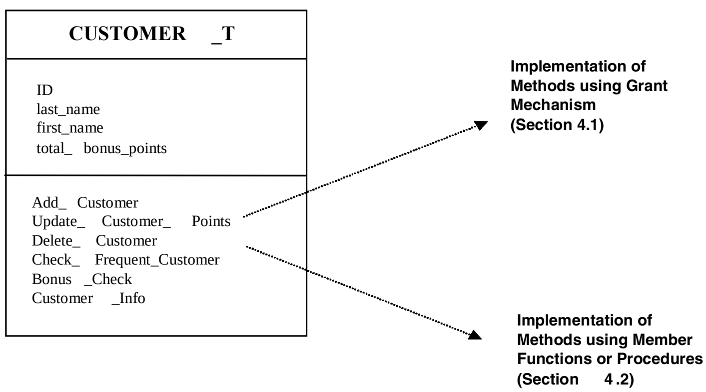


Figure 4.3. Stored procedures for Add_Customer

```
   CREATE OR REPLACE PROCEDURE Add_Customer(
      new_id            Customer.id%TYPE,
      new_last_name     Customer.last_name%TYPE,
      new_first_name    Customer.first_name%TYPE) AS

   BEGIN
      -- When inserting a new customer,
      -- the initial default value for total_bonus_points is 0.

      -- The Update_Customer_Points method can be used
      -- to modify the total_bonus_points.

      INSERT INTO Customer
         (id, last_name, first_name, total_bonus_points)
      VALUES (new_id, new_last_name, new_first_name, 0);
   END Add_Customer;
/
```

Figure 4.4. Stored procedures for Update_Customer_Points

```
General Syntax of Exception:

   EXCEPTION
      WHEN <Exception_name> THEN <statements>

Example:

CREATE OR REPLACE PROCEDURE Update_Customer_Points(
   new_id            Customer.id%TYPE,
   points            Customer.total_bonus_points%TYPE) AS

   old_bonus_points  NUMBER;

BEGIN
   SELECT total_bonus_points INTO old_bonus_points
   FROM Customer
   WHERE id = new_id;

   UPDATE Customer
   SET total_bonus_points = old_bonus_points + points
   WHERE id = new_id;

   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         INSERT INTO Customer (id, total_bonus_points)
         VALUES (new_id, points);

END Update_Customer_Points;
/
```

ID to be updated is not found in the Customer table, in which case a new customer record with the specified customer ID is created.

The example in Figure 4.5 demonstrates the use of the delete statement within a stored procedure. When the customer is not found in the database, a message will be displayed on the screen. Note that the statement "DBMS_output.put_line" is used for displaying results on the screen.

In order to process an SQL statement, Oracle™ allocates an area of memory known as the *context area*. The context area contains information necessary to complete the processing, including the number of rows processed by the statement, a pointer to the parsed representation of the statement, and in the case of a query, the *active set*, which is the set of rows returned by the query.

A *cursor* is a handle, or pointer, to the context area. Through the cursor, a PL/SQL program can control the context area and what happens to it as the statement is processed. The cursor declaration is placed before the procedure body.

The PL/SQL block in Figure 4.6 illustrates a cursor fetch loop, in which multiple rows of data are returned from a query. Notice that we are using a separate table, FreqClient, which has to be created first before we can execute the procedure.

The example in Figure 4.7 shows a stored procedure that produces the output to the screen rather than updating information in the database. A cursor is used

Figure 4.5. Stored procedures for Delete_Customer

```
    CREATE OR REPLACE PROCEDURE Delete_Customer(
       delete_id          Customer.id%TYPE,
       delete_last_name   Customer.last_name%TYPE) AS

    BEGIN
       DELETE FROM Customer
       WHERE id = delete_id
       AND last_name = delete_last_name;

       EXCEPTION
          WHEN NO_DATA_FOUND THEN
             DBMS_OUTPUT.PUT_LINE('Customer does not exist …');

    END Delete_Customer;
/
```

Figure 4.6. Stored procedures for Check_Frequent_Customer

```
General Syntax:

CREATE [OR REPLACE] PROCEDURE <procedure name> AS

CURSOR <cursor_name> IS
   SELECT <statement>;

BEGIN
   FOR <cursor variable> IN <cursor name> LOOP
      IF <condition>
         THEN <statement>
      [ELSEIF <condition>
         THEN <statement>]
      END IF;
   END LOOP;
END <procedure name>;

Example:

CREATE OR REPLACE PROCEDURE Check_Frequent_Customer AS
   -- Procedure to store those customers that have collected
   -- more than 100 points (frequent customer) into a separate
   -- table (FreqClient table)

CURSOR c_customer IS
   SELECT id, last_name, total_bonus_points
   FROM Customer;

BEGIN
   FOR v_customer_record IN c_customer LOOP
      IF (v_customer_record.total_bonus_points > 100) THEN
         INSERT INTO FreqClient
         VALUES
         (v_customer_record.id || ‘ ‘ ||
          v_customer_record.last_name || ‘ ‘ ||
         ' Frequent Customer! ');
      END IF;
   END LOOP;
END Check_Frequent_Customer;
/
```

to iterate each record in the database table. When the selection predicate is met, the record will be displayed on the screen.

Once a stored procedure is created, it is stored in the database. Hence, we can retrieve the stored procedure using a normal SQL select statement. For example, to retrieve the stored procedure Add_Customer, we can invoke the following select statement interactively.

Figure 4.7. Stored procedures for Bonus_Check

```
CREATE OR REPLACE PROCEDURE Bonus_Check(
minbonus NUMBER) AS

CURSOR c_bonus IS
   SELECT id, last_name, total_bonus_points
   FROM Customer
   WHERE total_bonus_points < minbonus;

BEGIN

   FOR v_bonus_record IN c_bonus LOOP
      DBMS_OUTPUT.PUT_LINE
      (v_bonus_record.id||' '||v_bonus_record.last_name||
       ' '||v_bonus_record.total_bonus_points);
   END LOOP;

END Bonus_Check;
/
```

Figure 4.8. Retrieving stored procedure

```
General Syntax:

SELECT line, text
FROM user_source
WHERE name = (stored procedure name)
[ORDER BY <attribute>];

Example:

SELECT line, text
FROM user_source
WHERE name = 'Add_Customer'
ORDER BY line;
```

Figure 4.9. Executing stored procedures

```
EXECUTE Add_Customer('92111', 'John', 'Done');
```

The result of the select statement in Figure 4.8 is a complete listing of the procedure Add_Customer. Each line begins with a line number. Users can also invoke a stored procedure and function through an execute command from SQL*Plus (Loney & Koch, 2000, 2002; ORACLE™ 8, 1997; Urman, 2000) as is shown with an example in Figure 4.9.

Figure 4.10. Stored functions for Customer_Info

```
   CREATE OR REPLACE FUNCTION Customer_Info(
      c_id                    Customer.id%TYPE,
      req_points              NUMBER)
      RETURN VARCHAR2 IS

      -- Returns 'Gold Point' if the Customer has completed all
      -- required bonus points,
      -- 'Silver Point' for over or equal to 75%,
      -- 'Bronze Point' for less than 75% and greater than 50%, and
      -- 'No Prize Yet' if the points are less than or equal to 50%.
      v_total_current_points        NUMBER;
      v_percent_completion          NUMBER;

   BEGIN

      SELECT total_bonus_points
      INTO v_total_current_points
      FROM Customer
      WHERE id = s_id;

      -- Calculate the current percentage.

      v_percent_completion :=
         v_total_current_points / req_points * 100;

      IF v_percent_completion = 100 THEN
         RETURN 'Gold Point';
      ELSIF v_percent_completion >= 75 THEN
         RETURN 'Silver Point';
      ELSIF v_percent_completion > 50 THEN
         RETURN 'Bronze Point';
      ELSE
         RETURN 'No Prize Yet';
      END IF;

   END Customer_Info;
/
```

Figure 4.11. Retrieving stored functions for Customer_Info

```
SELECT id, last_name, first_name, Customer_Info(id, 100)
FROM Customer;
```

Apart from stored procedures, we can also create a stored function (see Figure 4.10). The following example is a function that can be used to get information about a customer's bonus points.

With a stored function, we can display the output using a query as shown in Figure 4.11. The query will return a list of all customers in the Customer table together with their current bonus-point status. The required number of bonus points to get a gold point is 100.

Grant

Grant is often used in conjunction with stored procedures in RDBs, particularly in the context of data security. In Oracle™, one can restrict the database operations that users can perform by allowing them to access data only through procedures and functions (Loney & Koch, 2000, 2002; ORACLE™ 8, 1997; Urman, 2000). For example, one can grant users access to a procedure that updates one table, but not grant them access to the table itself. When a user invokes the procedure, the procedure executes with the privileges of the procedure's owner. Users who have only the privilege of executing the procedure (but not the privilege to query, update, or delete from the underlying tables) can invoke the procedure, but they cannot manipulate the table data in any other way (Loney & Koch; ORACLE™ 8; Urman).

In Oracle™ we can grant system, role, or object privileges to three different types mentioned below.

- *User*. The privilege is given to particular users, and the user can then exercise the privilege.
- *Role*. The privilege is given to particular roles, and the user who has been granted the role will be able to exercise the privilege.
- *Public*. The privilege is given to all users.

A grant on a system privilege is the grant to carry out a basic system operation such as create table, create procedure, and so forth. A grant on a role is the grant to access the information of the particular role. Finally, a grant on an object privilege is the grant to do a particular action to a particular object. Thus, for a grant on an object privilege, we need to declare the schema of the grant-object target. The general syntax for the grant statement is shown in Figure 4.12.

The use of grants to simulate object-oriented encapsulation is to grant users with no access to tables, and to grant users with execute accesses to the stored procedures where the methods are stored. Therefore, the tables are encapsulated with the stored procedures. For example, we want to grant a particular

Figure 4.12. Grant general syntax

```
GRANT [system privilege|role] TO [user|role|PUBLIC];

GRANT [object privilege] <object schema> TO [user|role|PUBLIC];
```

Figure 4.13. Grant object privilege to user

```
GRANT EXECUTE ON <procedure_name> TO <user>;
```

user with an object privilege to execute a stored procedure as shown in Figure 4.13.

Implementation of Encapsulation using Member Procedures or Functions

As mentioned previously, we can also implement object operations as member procedures or functions. The following example demonstrates the implementation of the Customer_T object together with its member procedures and functions. We reuse some of the routines defined in the previous section. Note the changes required for the implementation.

Figure 4.14. Object with member procedures and functions

```
    CREATE OR REPLACE TYPE Customer_T AS OBJECT
       (id                           VARCHAR2(10),
        last_name                    VARCHAR2(20),
        first_name                   VARCHAR2(20),
        total_bonus_points           NUMBER,

        MEMBER PROCEDURE
          Update_Customer_Points(c_points IN NUMBER),

        MEMBER FUNCTION
          Customer_Info(c_req_points IN NUMBER)
          RETURN VARCHAR2
       )
/
```

Figure 4.15. Method implementation of member procedures and functions

```
      CREATE OR REPLACE TYPE BODY Customer_T AS

         MEMBER PROCEDURE
         Update_Customer_Points(c_points IN NUMBER) IS

         BEGIN
            total_bonus_points := total_bonus_points + c_points;
         END Update_Customer_Points;

         MEMBER FUNCTION Customer_Info(s_req_points IN NUMBER)
            RETURN VARCHAR2 IS

         v_percent_completion    NUMBER;

         BEGIN
            -- Calculate the current percentage.

            v_percent_completion :=
               total_bonus_points / s_req_points * 100;

            IF v_percent_completion = 100 THEN
               RETURN 'Gold Point';
            ELSIF v_percent_completion >= 75 THEN
               RETURN 'Silver Point';
            ELSIF v_percent_completion > 50 THEN
               RETURN 'Bronze Point';
            ELSE
               RETURN 'No Prize Yet';
            END IF;

         END Customer_Info;

      END;
   /
```

Figure 4.16. Example of using self keyword

```
MEMBER PROCEDURE
   Update_Customer_Points(s_points IN NUMBER) IS

BEGIN
   self.total_bonus_points :=
      self.total_bonus_points + s_points;
   END Update_Customer_Points;
```

Figure 4.17. Syntax to call member procedures or functions

```
Object_name.member_procedure_name

Object_name.member_function_name
```

From the description in Figure 4.15, it is clear that the main difference between the normal procedures and functions and the member procedures and functions is the fact that we do not need to use the working object as a parameter in member procedures and functions. It is automatically referenced by the current working object, which eliminates the need to search for it first. Hence, the parameter new_ID, which is used to locate the current working object, is no longer necessary.

We can also use the keyword *self* to identify that the object we are referring is the current working object. For example, the above member procedure Update_Customer_Points can be written as follows.

In order to call or to use the above member functions or procedures, we need a reference to a particular object instance (i.e., the current working object). For example, in the above case, we need to instantiate a Customer_T object and use the object to execute the procedures and functions. The syntax for calling a member function or procedure is shown in Figure 4.16.

The procedure in Figure 4.17 shows how we can use the previous member procedures and functions. Declarations after "declare" can be an object, variables, or other declarations.

The example in Figure 4.18 shows how we can call member procedures and functions by first constructing a single object and then calling the methods that are applicable to that object. The result for the above procedure is shown after the code.

The example in Figure 4.18 demonstrates the use of member procedures and member functions using Option 1 in Figure 4.19.

In Option 2, the object is created from a record within a relational table. We call the table here Customer and it is used to store customer records. The following procedure shows how we apply the member procedures and functions as defined earlier for the Customer_T object to manipulate records from the Customer table.

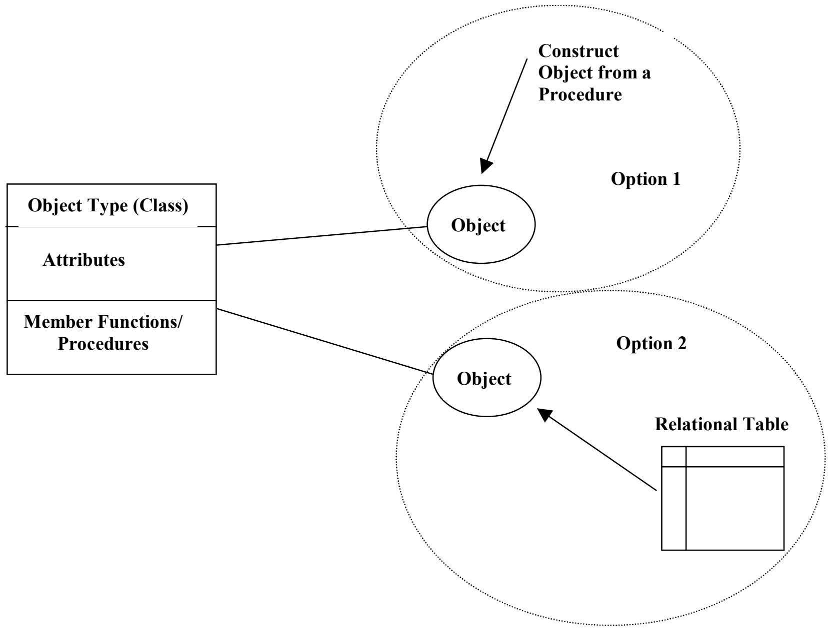
Figure 4.18. Member procedure or function call using an object

```
General Syntax:

DECLARE <declarations>
BEGIN
   <procedure body>
END;
/

Example:

DECLARE
   -- Construct a Customer object a_Customer.
   a_Customer Customer_T :=
      Customer_T('980790X', 'Smith', 'John', 50);


BEGIN
   -- Call procedure to update a_Customer total bonus points
   a_Customer.Update_Customer_Points(30);

   DBMS_OUTPUT.PUT_LINE
   ('New total points is '|| a_Customer.total_bonus_points);

   -- Call function to display the completion
   DBMS_OUTPUT.PUT_LINE (a_Customer.Customer_Info(100));

END;
/

New total points is 50
No Prize Yet
```

As mentioned previously, there are some differences between the implementation of stored procedures or functions and member procedures or functions.

- Stored procedures or functions are mainly used for pure relational systems where there is no member-object concept available. Obviously, member procedures and member functions are used for systems with an object-oriented feature, such as object-relational database systems.
- We do not need to use the working object as a parameter in member procedures or functions. It automatically refers to the current working object, which eliminates the need to search for it first.

Figure 4.19. Member procedure and function implementation options



- The %type cannot be applied to an attribute of an object type directly. It must be applied to an attribute of an instantiation of an object type (i.e., a table). Therefore, for member routines, we need to directly clarify the data type of each parameter.

Case Study

The Victorian tourism department stores the data of main tourist attractions in a database that can be accessed from every tourist information centre across the state. The database contains information about the name of the tourist destination, location, tourism type, and season. For each destination, the database provides the accommodations available around the area. The accommodation data includes the name, type, rate, address, and the contact details of the accommodation. Currently, the database is stored in a pure RDB with the E/R diagram shown next.

Figure 4.20. Member procedure and function call using a relational table

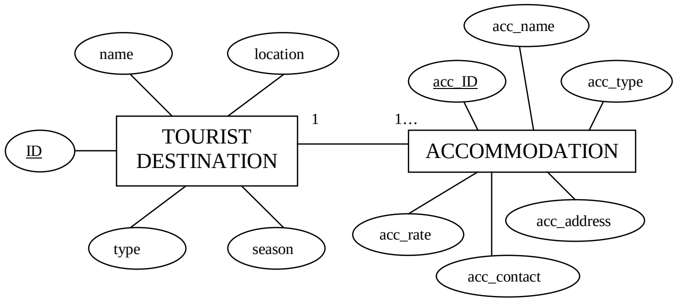
```
DECLARE

CURSOR c_customer IS
   SELECT id, last_name, first_name, total_bonus_points
   FROM Customer;

   -- Construct and initialise a_Customer object.
   a_Customer Customer_T := Customer_T(NULL,NULL,NULL,0);

BEGIN

   FOR v_customer_record IN c_customer LOOP

      -- Assign values to a_Customer object.
      a_Customer.id := v_customer_record.id;
      a_Customer.last_name := v_customer_record.last_name;
      a_Customer.first_name := v_customer_record.first_name;
      a_Customer.total_bonus_points:=
      v_customer_record.total_bonus_points;

      DBMS_OUTPUT.PUT_LINE
      (a_Customer.id||' '||a_Customer.last_name||' '||
      a_Customer.total_bonus_points);

      -- Call Update_Customer_Points to update a_Customer
      -- total points with another 30 points.
      a_Customer.Update_Customer_Credit(30);

      DBMS_OUTPUT.PUT_LINE
      ('The new total points is '||
      a_Customer.total_bonus_points);

      -- Call Customer_Info function to display whether a_Customer
      -- achieves a bonus prize. Gold Point is given for points
      -- equal to 100.
      DBMS_OUTPUT.PUT_LINE (a_Customer.Customer_Info(100));

   END LOOP;

END;
/
```

There are two query transactions that are frequently made by the users.

a. Given the ID, show the details of a tourist destination.

b. Given the accommodation ID, show its details including the name and the location of the tourist destination associated with the accommodation.

Figure 4.21. E/R diagram of the tourism-department case study



Due to the expansion of the database size, the department now wants to transform the database system into an object-relational system, with frequent procedures attached to the objects. The design of the object diagram is shown in Figure 4.22.

We need to show the implementation of the databases using both stored procedures of a pure relational system and member procedures of an object-relational system.

First, we create the stored procedures for tables Tourist_Destination and Accommodation. Assume that these tables already exist. The relational schema is shown in Figure 4.23 along with the stored procedures. Note that there is a foreign key of ID in the Accommodation table that references the attribute ID in the Tourist_Destination table.

The next step is to implement the member procedure. For this step, we start from the method declaration followed by the method implementation (see Figure 4.24).

Figure 4.22. Object diagram of the tourism-department case study

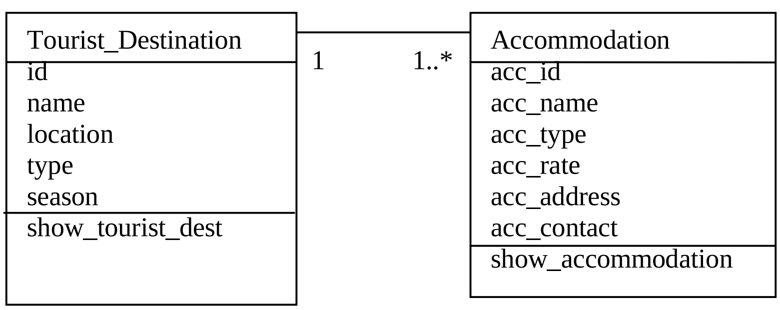


Figure 4.23. Stored-procedures implementation for the tourism-department case study

```
Relational Schemas
   Tourist_Destination (ID, name, location, type, season)
   Accommodation (acc_ID, acc_name, acc_type, acc_rate,
                 acc_address, acc_contact, ID)

Stored Procedures
CREATE OR REPLACE PROCEDURE Show_Tourist_Dest(
    new_id IN Tourist_Destination.id%TYPE) AS

   new_name Tourist_Destination.name%TYPE;
   new_location Tourist_Destination.location%TYPE;
   new_type Tourist_Destination.type%TYPE;

BEGIN

   SELECT name, location, type
   INTO new_name, new_location, new_type
   FROM Tourist_Destination
   WHERE id = new_id;

   DBMS_OUTPUT.PUT_LINE
      (new_name||' '||new_location||' '||new_type);

END Show_Tourist_Dest;
/

CREATE OR REPLACE PROCEDURE Show_Accommodation(
    new_id     IN Accommodation.id%TYPE) AS

   new_acc_name Accommodation.acc_name%TYPE;
   new_acc_address Accommodation.acc_address%TYPE;
   new_acc_contact Accommodation.acc_contact%TYPE;
   new_destination_name Tourist_Destination.name%TYPE;
   new_destination_location Tourist_Destination.location%TYPE;

BEGIN

   SELECT a.acc_name, a.acc_address, a.acc_contact, b.name, b.location
   INTO new_acc_name, new_acc_address, new_acc_contact,
   new_destination_name, new_destination_location
   FROM Accommodation a, Tourist_Destination b
   WHERE a.id = b.id
   AND b.id = new_id;

   DBMS_OUTPUT.PUT_LINE
   (new_acc_name||' '||new_acc_address||' '||new_acc_contact
   ||' '||new_destination_name||' '||new_destination_location)
   END LOOP;

END Show_Accommodation;
/
```

Figure 4.24. Member-methods implementation for the tourism-department case study

```
Methods Declaration
   CREATE OR REPLACE TYPE Tourist_Destination_T AS OBJECT
      (id            VARCHAR2(10),
       name          VARCHAR2(30),
       location      VARCHAR2(30),
       type          VARCHAR2(20),
       season        VARCHAR2(10),

       MEMBER PROCEDURE Show_Tourist_Dest)
   /

   CREATE TABLE Tourist_Destination OF
      Tourist_Destination_T
      (id NOT NULL,
       PRIMARY KEY (id));

   CREATE OR REPLACE TYPE Accommodation_T AS OBJECT
      (acc_id              VARCHAR2(10),
       acc_name            VARCHAR2(30),
       acc_type            VARCHAR2(30),
       acc_rate            NUMBER,
       acc_address         VARCHAR2(30),
       acc_contact         VARCHAR2(10),
       destination         REF Tourist_Destination_T,

       MEMBER PROCEDURE Show_Accommodation)
   /

   CREATE TABLE Accommodation OF Accommodation_T
      (acc_id NOT NULL,
       PRIMARY KEY (acc_id));


Methods Implementation
   CREATE OR REPLACE TYPE BODY Tourist_Destination_T AS

      MEMBER PROCEDURE Show_Tourist_Dest IS

      BEGIN
         DBMS_OUTPUT.PUT_LINE
           (self.name||' '||self.location||' '||self.type);
         END LOOP;
```

Figure 4.24. (continued)

```
   END Show_Tourist_Dest;
END;
/

CREATE OR REPLACE TYPE BODY Accommodation_T AS

   MEMBER PROCEDURE Show_Accommodation IS

      new_destination_name Tourist_Destination.name%TYPE;
      new_destination_location Tourist_Destination.location%TYPE;

   BEGIN

      SELECT name, location
      INTO new_destination_name, new_destination_location
      FROM Tourist_Destination
      WHERE destination.id = self.id;

   BEGIN
      DBMS_OUTPUT.PUT_LINE
      (self.acc_name||' '||self.acc_address||' '||self.acc_contact
      ||' '||new_destination_name||' '||new_destination_location);
   END Show_Accommodation;

END;
/
```

Summary

In a pure RDB system, packages such as stored procedures and functions are used to implement operations. With the additional grant mechanism, data security can be performed with stored procedures and functions so that only a certain user or role is privileged to access the system, role, and object. In ORDBMS, the concept of data security can be performed by having member methods. With the encapsulation feature in the object-oriented model, we can add member procedures and functions inside a class along with the member attributes. The declaration and implementation of member methods are separated as in other programming-language practices.

References

Loney, K., & Koch, G. (2000). *Oracle™ 8i: The complete reference.* Osborne McGraw-Hill.

Loney, K., & Koch, G. (2002). *Oracle™ 9i: The complete reference.* Oracle Press.

ORACLE™ 8. (1997). *Oracle™ 8 product documentation library.* Redwood City, CA: Oracle Corporation.

Urman, S. (2000). *Oracle™ 8i advanced PL/SQL programming.* Oracle Press, Osborne.

Chapter Problems

1. King Electronic is going to have its end-of-year 2005 sale. Every year the owner keeps the record of each item to be put on sale. The data is placed on an object-based database and table as follows.

Sale2005_T
item_code item_name quantity price

Sale2005			
Item_Code	**Item_Name**	**Quantity**	**Price**
SV101	VCR	20	150
SD101	DVD Player	20	225
SD102	DVD Player 2	10	350
ST101	TV 14"	15	400
ST102	TV 21"	20	700
ST103	TV 30"	10	1200
SP101	PS One	40	150
SP102	PS Two	20	450

a. Write a stored procedure to insert other sale items into the Sale2005 table.

b. Write a stored procedure to update the quantity of the item in the table every time an item is sold or added to the sale stock.

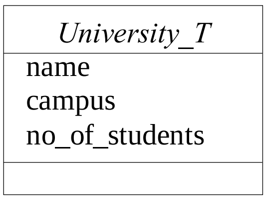
2. From Question 1, write statements to grant the following.

   a. Object privilege to execute Insert_Item to the user name Michael

   b. Object privilege to execute Update_Stock to the role Sales

   c. System privilege to create the user, type, and table to the role Admin.

3. The Victorian Department of Education and Training has records of every university in the state with all their details. For the purpose of accessing statistics quickly, the department develops an object University_T that contains the main information about the university.



University		
Name	**Campus**	**No_of_Students**
Melbourne University	Melbourne	28,000
Monash University	Berwick, Caulfield, Clayton, Gippsland, Peninsula	45,000
La Trobe University	Albury, Beechworth, Bundoora, Bendigo, Mildura,	22,000
Deakin University	Burwood, Geelong, Warrnambool	31,000
University of Ballarat	Ararat, Ballarat, Horsham	18,000
Royal Melb. Institute of Tech.	Bundoora, Brunswick, City	54,000
Swinburne University	Hawthorn, Lilydale, Prahran	10,000
Victoria University	City, Footscray, Sunbury, Sunshine, Werribee	50,000

Write a stored procedure to retrieve data from the University table so that the names of those universities with more than 25,000 students are shown on the screen.

4. Cinema Classic wants to develop an interactive Web site where customers can query the movies that are currently available. The Web developer uses the movie database that has been used by Cinema Classic. The database consists of the movie code, title, year, genre, directors, cast, and rating. The examples of the records on the database are shown below.

Movie						
Code	**Title**	**Year**	**Genre**	**Director**	**Cast**	**Rating**
G01	*Gone with the Wind*	1939	Drama	Cukor, Fleming	Leigh, Gable	PG
P07	*Psycho*	1960	Horror	Hitchcock	Perkins, Miles	MA
S23	*Star Wars*	1977	Sci_Fi	Lucas	Hamill, Ford	G

For this purpose, the Web developer wants to implement an object-relational database. In the object, he has a member function that can show the description of the ratings. The description of the ratings is shown in the table below. For example, calling the member function with parameter G will return the string "Suitable for all viewers." Write the implementation of the movie object and the body.

Rating	
Rating	**Description**
G	Suitable for all viewers
PG	Parental guidance recommended for children under 15 years of age
M	Mature, recommended for audiences 15 years and over
MA	Mature, accompanied by a parent or adult guardian
R	Restricted to adults, no one under 18 may view these
X	Restricted to adults, sexually explicit material

Chapter Solutions

1. a.
```
CREATE TYPE Sale2005_T AS OBJECT
    (item_code     VARCHAR2(10),
     item_name     VARCHAR2(30),
     quantity      NUMBER,
     price         NUMBER)
/

CREATE TABLE Sale2005 OF Sale2005_T
(item_code NOT NULL);

CREATE OR REPLACE PROCEDURE Insert_Item(
   new_item_code  IN Sale2005.item_code%TYPE,
   new_item_name  IN Sale2005.item_name%TYPE,
   new_quantity  IN Sale2005.quantity%TYPE,
   new_price  IN Sale2005.price%TYPE) AS

BEGIN
   INSERT INTO Sale2005
      (item_code, item_name, quantity, price)
   VALUES
      (new_item_code, new_item_name, new_quantity,
       new_price);
END Insert_Item;
/
```

b.
```
CREATE OR REPLACE PROCEDURE Update_Stock(
   sold_item_code IN Sale2005.item_code%TYPE,
   number_sold IN NUMBER) AS

   old_quantity   NUMBER;
   new_quantity   NUMBER;

BEGIN
   SELECT quantity INTO old_quantity
   FROM Sale2005
   WHERE item_code = sold_item_code

   FOR UPDATE OF quantity;
      new_quantity := old_quantity - number_sold;
      UPDATE Sale2005
         SET quantity = new_quantity
         WHERE item_code = sold_item_code;
END Update_Stock;
/
```

2. a. **GRANT EXECUTE ON** Insert_Item **TO** Michael;

   b. **GRANT EXECUTE ON** Update_Stock **TO** Sales;

   c. **GRANT CREATE USER, CREATE TABLE, CREATE TYPE TO** Admin;

3.
```
CREATE OR REPLACE PROCEDURE Above_25000 AS

CURSOR c_university IS
   SELECT name, no_of_students
   FROM University_T
   WHERE no_of_students > 25000;

BEGIN
   FOR v_uni_record IN c_university LOOP
      DBMS_OUTPUT.PUT_LINE
      (v_uni_record.name||‘
      ’||v_uni_record.no_of_students);
   END LOOP;
END Above_25000;
/
```

4.
```
CREATE OR REPLACE TYPE Movie_T AS OBJECT
    (code       VARCHAR2(5),
     title      VARCHAR2(40),
     year       NUMBER,
     genre      VARCHAR2(20),
     director   VARCHAR2(20),
     cast       VARCHAR2(50),
     rating     VARCHAR2(3),

   MEMBER FUNCTION Rating_Info
   (rating_code IN VARCHAR2)
      RETURN VARCHAR2)
/

CREATE OR REPLACE TYPE BODY Movie_T AS

   MEMBER FUNCTION Rating_Info
   (rating_code IN VARCHAR2)
      RETURN VARCHAR2 IS

   BEGIN
      IF rating_code = ‘G’ THEN
         RETURN ‘ Suitable for all viewers ’;
      ELSIF rating_code = ‘PG’ THEN
```

```
      RETURN 'Parental guidance recommended for
    children under 15 years of age';
    ELSIF rating_code = 'M' THEN
      RETURN 'Mature, recommended for audiences 15
    years and over';
    ELSIF rating_code = 'MA' THEN
      RETURN 'Mature, accompanied by a parent or
    adult guardian';
    ELSIF rating_code = 'R' THEN
      RETURN 'Restricted to adults, no one under
    18 may view these';
    ELSIF rating_code = 'X' THEN
      RETURN 'Restricted to adults, sexually
    explicit material';
    ELSE
      RETURN 'No rating';
    END IF;
  END Rating_Info;

END;
/
```

Chapter V

Generic Methods

Generic methods are the methods used to access the attributes of an object. The concept behind the need for generic methods is encapsulation, in which attributes associated with an object can be accessed directly only by the methods within the object itself. Therefore, each time an attribute is created within an object, we will need generic methods to access the attribute. This is the main difference between the standard relational techniques for implementing operations vs. the object-oriented methods. In relational databases, users normally can directly access attributes of a table by running SQL statements to update, delete, or insert. This may generate problems when certain attributes within an object have some constraints applied to them, and therefore the ad hoc access may violate these constraints.

As discussed in Section 1.3.1, generic methods are tightly related to the update and delete operations. Therefore, generic methods are associated with the concept of referential integrity. As in conventional relational systems, for each update and delete operation, there has to be an identified action to be carried out (i.e., cascade, restrict, and nullify). The transformation of object structures, including inheritance, aggregation, and association, involves primary-key and foreign-key association or object references, depending on the techniques used to represent the relationships. Therefore, for each object structure, actions for the update and delete operations have to be identified. In this section, we will show mainly the application of methods for updating and deletion, and the actions taken to maintain the referential integrity.

Additionally, the insert and retrieval operations, to some extent, also correlate with referential integrity. An insertion to a foreign key is known to match with the associated primary key of another table. A retrieval of a composite object will need to form join operations between foreign keys and their matching primary keys.

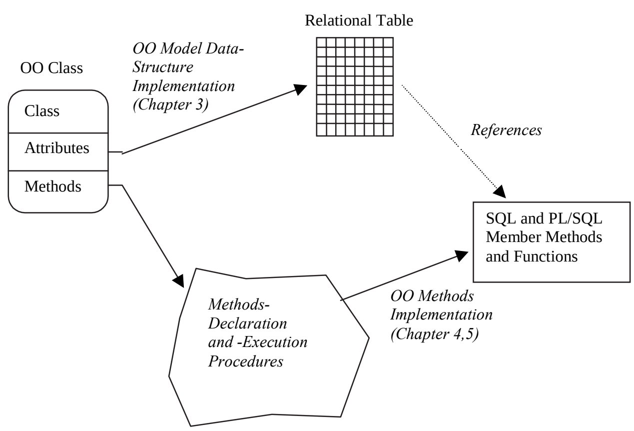
Figure 5.1 shows an overview of the implementation of generic methods. The process consists of several steps. First, given an object-oriented model, data-structure mapping is carried out. This basically applies the static transformation procedures (Chapter 3). Second, the key elements of each generic method (operations, parameters, constraints, etc.), which will be used as a basis for the implementation of the methods, are identified.

Implementation of Methods in Inheritance Hierarchies

There are some approaches that can be used for inheritance relationships implementation into tables. The usage of these approaches can be explored based on the types of inheritance: union inheritance, mutual-exclusion inheritance, partition inheritance, and multiple inheritance. In this section, we are going to see the implementation of generic methods in inheritance hierarchies.

Note that in Chapter III we described the two different ways of implementing inheritance in Oracle™. The first method uses a shared ID, which is mainly used

Figure 5.1. Generic methods



in earlier versions of Oracle™ (before Oracle™ 9) as well as other standard relational database systems that do not support inheritance. The second method uses the keyword under in Oracle™ 9 and above, which defines an inheritance hierarchy. In the following sections we will show the implementation of methods using both techniques.

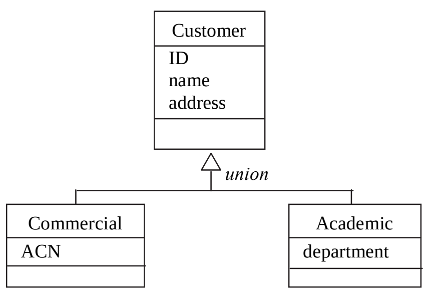
Implementation of Methods in Union Inheritance

We recall from Chapter III that the first technique for implementing union inheritance into relational tables is done by creating a separate table for the superclass and a table for each of the subclasses. A shared ID is used to preserve the identity of the objects across tables. Suppose Figure 5.2 is a union inheritance. Each class, together with its local attributes in the inheritance schema, is mapped into a table. This way of mapping is often called a *vertical division* since the list of attributes of a subclass is divided into several classes.

In the example, the declaration of attributes or properties belonging to an academic (i.e., ID, name, address, department) has to be divided into two tables, namely, table Customer and table Academic. This kind of declaration follows the way the class is declared. Since class Customer is already declared first, class Academic that inherits from class Customer merely adds a few more attributes. Inheritance provides a reuse mechanism whereby the definition of a new class is based on some existing classes. Consequently, the creation of new subclasses from scratch can be avoided; instead, some or all parts of the existing classes can be reused in the new classes.

Figure 5.3 shows the results of the implementation of this union inheritance including the methods. Note that we use member methods instead of an

Figure 5.2. Union inheritance



ordinary stored procedure or function to put into practice the object-oriented feature in Oracle™. The rules for member methods can be specified as follows.

- **Methods Declaration.** The declaration of the method is performed during object type creation. Thus, when we create a type, we have to know the name of the methods in it, whether it will be a procedure or a function, and also the parameters needed.
- **Methods Implementation.** After the type is created, we will need to specify the body of the type. In the body, the implementation details of the member methods are specified.

There are a few things to observe from the implementation of the union inheritance example as shown in Figure 5.3 and Figure 5.4.

First is that we provide not only the relational schemas, but also the SQL statements to create and manipulate the schemas. Some sample records are also provided to help readers visualise the implementation of union inheritance.

Second is that attribute ID of the subclass (e.g., class Commercial and Academic) is also a foreign key referencing to the superclass. This is to ensure that a subclass OID must exist as an OID in the superclass. Notice also that the

Figure 5.3. Implementation of union inheritance relationship

Relational Schemas

```
Customer (ID, name, address)
Commercial (ID, ACN)
Academic (ID, department)
```

Methods Declaration

```
CREATE OR REPLACE TYPE Customer_T AS OBJECT
    (id      VARCHAR2(10),
     name    VARCHAR2(30),
     address VARCHAR2(30),

     MEMBER PROCEDURE Insert_Customer(
       new_id IN VARCHAR2,
       new_name IN VARCHAR2,
       new_address IN VARCHAR2),

     MEMBER PROCEDURE Delete_Customer)
/
```

Figure 5.3. (continued)

```
CREATE TABLE Customer OF Customer_T
   (id NOT NULL,
    PRIMARY KEY (id));

CREATE OR REPLACE TYPE Commercial_T AS OBJECT
   (id      VARCHAR2(10),
    acn     VARCHAR2(30),

    MEMBER PROCEDURE Insert_Commercial(
      new_id IN VARCHAR2,
      new_name IN VARCHAR2,
      new_address IN VARCHAR2,
      new_acn IN VARCHAR2),

    MEMBER PROCEDURE Delete_Commercial)
/

CREATE TABLE Commercial OF Commercial_T
   (id NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Customer(id)
    ON DELETE CASCADE);

CREATE OR REPLACE TYPE Academic_T AS OBJECT
   (id            VARCHAR2(10),
    department    VARCHAR2(30),

    MEMBER PROCEDURE Insert_Academic(
      new_id IN VARCHAR2,
      new_name IN VARCHAR2,
      new_address IN VARCHAR2,
      new_department IN VARCHAR2),

    MEMBER PROCEDURE Delete_Academic)
/

CREATE TABLE Academic OF Academic_T
   (id NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Customer(id)
    ON DELETE CASCADE);
```

Methods Implementation

```
CREATE OR REPLACE TYPE BODY Customer_T AS

   MEMBER PROCEDURE Insert_Customer(
      new_id IN VARCHAR2,
      new_name IN VARCHAR2,
      new_address IN VARCHAR2) IS
```

Figure 5.3. (continued)

```
      BEGIN
         INSERT INTO Customer
         VALUES (new_id, new_name, new_address);
      END Insert_Customer;

      MEMBER PROCEDURE Delete_Customer IS

      BEGIN
         DELETE FROM Customer
         WHERE Customer.id = self.id;
      END Delete_Customer;

   END;
   /

   CREATE OR REPLACE TYPE BODY Commercial_T AS

      MEMBER PROCEDURE Insert_Commercial(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_acn IN VARCHAR2) IS

      BEGIN
         INSERT INTO Customer
         VALUES (new_id, new_name, new_address);
         INSERT INTO Commercial
         VALUES (new_id, new_acn);
      END Insert_Commercial;

      MEMBER PROCEDURE Delete_Commercial IS

      BEGIN
      DELETE FROM Commercial
         WHERE Commercial.id = self.id;
      DELETE FROM Customer
         WHERE
            (Customer.id = self.id) AND
            (Customer.id NOT IN
               (SELECT Academic.id
                FROM Academic
                WHERE Academic.id = self.id)) AND
            (Customer.id NOT IN
               (<selection of any other sibling sub-classes>);
      END Delete_Commercial;

   END;
   /

   CREATE OR REPLACE TYPE BODY Academic_T AS
```

Figure 5.3. (continued)

```
      MEMBER PROCEDURE Insert_Academic(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_department IN VARCHAR2) IS

      BEGIN
         INSERT INTO Customer
         VALUES (new_id, new_name, new_address);
         INSERT INTO Academic
         VALUES (new_id, new_department);
      END Insert_Academic;

      MEMBER PROCEDURE Delete_Academic IS

      BEGIN
      DELETE FROM Academic
         WHERE Academic.id = self.id;
      DELETE FROM Customer
         WHERE
            (Customer.id = self.id) AND
            (Customer.id NOT IN
               (SELECT Commercial.id
                FROM Commercial
                WHERE Commercial.id = self.id)) AND
            (Customer.id NOT IN
               (<selection of any other sibling sub-classes>);
      END Delete_Academic;

   END;
   /

Methods Execution Example

DECLARE
      -- Construct objects, initialize them to null
      a_customer Customer_T := Customer_T(NULL,NULL,NULL);
      a_commercial Commercial_T := Commercial_T(NULL,NULL);
      a_academic Academic_T := Academic_T(NULL, NULL);

   BEGIN
      -- Call member procedures to insert data into
      -- Customer, Commercial, and Academic tables.
      a_customer.Insert_Customer
            ('1', 'Myers Pty Ltd', Melbourne');
      a_commercial.Insert_Commercial
            ('2', 'Coles Pty Ltd', 'Sydney, '443-765');
      a_academic.Insert_Academic
            ('3' 'La Trobe Univ', 'Bundoora', 'Comp Sc.');

   END;
   /
```

Figure 5.4 Example of a union inheritance table

Sample Records:

Customer		
ID	**Name**	**Address**
1	Myer Pty Ltd.	Melbourne
2	Coles Pty Ltd.	Sydney
3	LaTrobe Univ.	Bundoora
4	Monash Univ.	Gippsland
5	RMIT Univ.	Melbourne
6	Victoria Univ.	Footscray
7	***Holmes Inst.***	***Melbourne***
8	***Federal Gov.***	***Canberra***

Commercial	
ID	**ACN**
1	123-423
2	443-765
7	***011-333***

Academic	
ID	**Department**
3	Comp. Sc.
4	Info. Tech
5	Comp. Sc.
6	Informatics
7	***Info. Studies***

referential integrity constraint is "delete cascade." These imply that when a superclass record is deleted, all matching subclass records are automatically deleted as well.

Third is that union inheritance allows a new object Customer, not belonging to any of the subclasses, to be inserted. Hence, we provide a member procedure Insert_Customer that can be used for this purpose, along with Insert_Commercial and Insert_Academic for subclass object insertion. Insert_Customer will not be made available in other types of inheritance, especially partition inheritance. This will be discussed later.

Fourth relates to insertion. For the insertion of subclass records, insertion to the superclass has to be made first as the primary key of the subclass table is also a foreign key of the superclass table. If this insertion order is not obeyed, the insertion operation will fail due to the referential integrity enforced by the notion of the foreign key in the subclass table. As we use the encapsulation method of insertion (see Figure 5.3), we can only insert a record into one subclass because the insertion to the superclass is done immediately before insertion to the subclass. Insertion to another subclass will be restricted because there will be duplication of the primary key in the superclass. Therefore, if we want to insert a record into more than one subclass, after the first subclass, we can only use

Figure 5.5. Simple insertion generic method

Customer:

```
INSERT INTO Customer
   VALUES (&new_id, &new_name, &new_address);
```

Commercial:

```
INSERT INTO Customer
   VALUES (&new_id, &new_name, &new_address);
INSERT INTO Commercial
   VALUES (&new_id, &new_acn);
```

Academic:

```
INSERT INTO Customer
   VALUES (&new_id, &new_name, &new_address);
   INSERT INTO Academic
   VALUES (&new_id, &new_department);
```

the usual generic method. Examples of the usual generic methods are shown in Figure 5.5. Notice that we use an ampersand in front of a user-defined variable.

Fifth is regarding deletion. Deleting customer records is straightforward, and because delete is cascaded, the deletion will automatically be carried out to the matching records in the subclasses. However, the deletion of subclass records (such as deleting an academic object) is rather complex as we cannot apply the same method as that for customer deletion. This is because the deleted subclass records may still exist in other sibling subclasses in which the matched superclass record should not be deleted. Therefore, we first delete the intended subclass record, and then we check whether this record does not exist in other sibling subclass tables. If it does not exist, we can delete the root record in the superclass table.

Sixth, update methods are not provided because the OID is immutable and an update to the OID is not permitted. The update of nonkey attributes is isolated to the relevant table only; hence, no complexity arises in an update.

Finally, the sample records show that customer Holmes Institute is a commercial as well as an academic customer, and customer Federal Government is neither a commercial customer nor an academic customer (both examples are printed in bold and italic). These two objects illustrate the fact that this is a union inheritance.

The implementation example mentioned in detail (see Figure 5.3) is made using the older Oracle™ version, which does not provide the inheritance feature. However, as mentioned previously, Oracle™ 9 and the newer version have provided an inheritance relationship (see Figure 5.6).

To accommodate union inheritance with the newer Oracle™ version, we create the tables for each type. However, we use a supertype table, Customer, only for data that do not belong to any of the subtype classes. In the sample records (see Figure 5.4), it will be the Federal Government. If we know that the data belongs to any subtype class, we can use subtype member methods straightaway. The weakness is that there will be repetition of a customer's common attributes in each of its subtype tables. This repetition is at a cost not only in insertion time, but also in storage space. Nevertheless, it has benefits compared with the previous Oracle™ version. We can insert into many subtype tables using their member methods without having to use a simple generic method

Figure 5.6. Implementation of union inheritance relationship in newer Oracle™

```
Methods Declaration
   CREATE OR REPLACE TYPE Customer_T AS OBJECT
      (id      VARCHAR2(10),
       name    VARCHAR2(30),
       address VARCHAR2(30),

       MEMBER PROCEDURE Insert_Customer(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2),

       MEMBER PROCEDURE Delete_Customer) NOT FINAL
   /

   CREATE TABLE Customer OF Customer_T
      (id NOT NULL,
       PRIMARY KEY (id));

   CREATE OR REPLACE TYPE Commercial_T UNDER Customer_T
      (acn     VARCHAR2(30),

       MEMBER PROCEDURE Insert_Commercial(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_acn IN VARCHAR2),
```

Figure 5.6 . (continued)

```
      MEMBER PROCEDURE Delete_Commercial)
   /

   CREATE TABLE Commercial OF Commercial_T
      (id NOT NULL,
       PRIMARY KEY (id));

   CREATE OR REPLACE TYPE Academic_T UNDER Customer_T
      (department    VARCHAR2(30),

       MEMBER PROCEDURE Insert_Academic(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_department IN VARCHAR2),

       MEMBER PROCEDURE Delete_Academic)
   /

   CREATE TABLE Academic OF Academic_T
      (id NOT NULL,
       PRIMARY KEY (id));

Methods Implementation
   CREATE OR REPLACE TYPE BODY Commercial_T AS

      MEMBER PROCEDURE Insert_Commercial(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_acn IN VARCHAR2) IS

      BEGIN
      INSERT INTO Commercial
         VALUES (new_id, new_name, new_address, new_acn);
      END Insert_Commercial;

      MEMBER PROCEDURE Delete_Commercial IS

      BEGIN
         DELETE FROM Commercial
         WHERE (Commercial.id = self.id);
      END Delete_Commercial;

   END;
   /
```

Figure 5.6. (continued)

```
CREATE OR REPLACE TYPE BODY Academic_T AS

   MEMBER PROCEDURE Insert_Academic(
      new_id IN VARCHAR2,
      new_name IN VARCHAR2,
      new_address IN VARCHAR2,
      new_department IN VARCHAR2)IS

   BEGIN
      INSERT INTO Academic
      VALUES (new_id, new_name, new_address, new_department);
   END Insert_Academic;

   MEMBER PROCEDURE Delete_Academic IS

   BEGIN
      DELETE FROM Academic
      WHERE (Academic.id = self.id);
   END DeleteAcademic;

END;
/
```

Methods Execution Example

```
DECLARE
      a_customer Customer_T := Customer_T(NULL,NULL,NULL);
      a_commercial Commercial_T := Commercial_T(NULL,NULL,NULL,N
      a_academic Academic_T := Academic_T(NULL,NULL,NULL, NULL);

   BEGIN
      a_customer.Insert_Customer
         ('8', 'Federal Gov', 'Canberra');
      a_commercial.Insert_Commercial
         ('7', 'Holmes Inst', 'Melbourne', '011-333');
      a_academic.Insert_Academic
         ('7', 'Holmes Inst', 'Melbourne', 'Info. Studies');
   END;
   /
```

because the insertion of each table is done without first inserting into the supertype table.

Implementation of Methods in Mutual-Exclusion Inheritance

Handling mutual-exclusion inheritance without losing the semantic of the relationship is achieved by adding an attribute that reflects the type of the subclasses to the superclass table. Instead of union inheritance, suppose Figure 5.2 is of mutual-exclusion inheritance. The table Customer will have an additional attribute called cust_type to ensure that every customer's record in the table has a definite type, either commercial or academic. There are no customers that can refer simultaneously to both a commercial and an academic customer. The transformation is shown in Figure 5.7. We also show a deletion example using both the OID and non-OID.

A number of observations can be made regarding the above transformation results. First, attribute cust_type in table Customer is added, and it includes a check constraint in which a check for whether the value of this attribute is either Commercial or Academic is carried out. Notice also that in the create-table statement, attribute cust_type does not have a "not null" constraint in order to allow a noncommercial or academic customer.

Second, the creation of subclass tables is identical to that in union inheritance. In other words, referential integrity constraints are still upheld in this inheritance.

Third, for insertion, an appropriate subclass name is inserted as attribute cust_type. In the case where a customer has no subtype (e.g., customer Federal Government of Canberra with ID 8), a null value is inserted. Notice also that the order of subclass-records insertion is nontrivial as is that of union inheritance.

Fourth, the deletion in mutual exclusion is much simpler than that of union inheritance due to the fact that a subclass object belongs to only one subclass. Deleting a subclass object can be done at once by deleting the root object in the superclass table. Since deletion is cascaded, the matching subclass records will be deleted automatically. Therefore, deletion in member procedures needs

Figure 5.7. Implementation of mutual-exclusion inheritance relationship

Relational Schemas

```
   Customer (ID, name, address, cust_type)
   Commercial (ID, ACN)
   Academic (ID, department)
```

Methods Declaration

```
   CREATE OR REPLACE TYPE Customer_T AS OBJECT
      (id            VARCHAR2(10),
       name          VARCHAR2(30),
       address       VARCHAR2(30),
       cust_type     VARCHAR2(15),

       MEMBER PROCEDURE Insert_Customer(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2),

       MEMBER PROCEDURE Delete_Customer_OID,

       MEMBER PROCEDURE Delete_Customer_non_OID(
         deleted_attribute IN VARCHAR2)
   /

   CREATE TABLE Customer OF Customer_T
      (id NOT NULL,
       cust_type CHECK (cust_type in ('Commercial', 'Academic', NULL)),
       PRIMARY KEY (id));

   CREATE OR REPLACE TYPE Commercial_T AS OBJECT
      (id      VARCHAR2(10),
       acn     VARCHAR2(30),

       MEMBER PROCEDURE Insert_Commercial(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_acn IN VARCHAR2),

       MEMBER PROCEDURE Delete_Commercial_OID,

       MEMBER PROCEDURE Delete_Commercial_non_OID(
         deleted_attribute IN VARCHAR2)
   /

   CREATE TABLE Commercial OF Commercial_T
      (id NOT NULL,
       PRIMARY KEY (id),
       FOREIGN KEY (id) REFERENCES Customer(id)
       ON DELETE CASCADE);

   CREATE OR REPLACE TYPE Academic_T AS OBJECT
      (id            VARCHAR2(10),
      department     VARCHAR2(30),
```

Figure 5.7. (continued)

```
      MEMBER PROCEDURE Insert_Academic(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_department IN VARCHAR2),

       MEMBER PROCEDURE Delete_Academic_OID,

       MEMBER PROCEDURE Delete_Academic_non_OID(
         deleted_attribute IN VARCHAR2))
   /

   CREATE TABLE Academic OF Academic_T
      (id NOT NULL,
       PRIMARY KEY (id),
       FOREIGN KEY (id) REFERENCES Customer(id)
       ON DELETE CASCADE);

Methods Implementation
   CREATE OR REPLACE TYPE BODY Customer_T AS

      MEMBER PROCEDURE Insert_Customer(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2) IS

      BEGIN
         INSERT INTO Customer
         VALUES (new_id, new_name, new_address, NULL);
      END Insert_Customer;

      MEMBER PROCEDURE Delete_Customer_OID IS

      BEGIN
         DELETE FROM Customer
         WHERE Customer.id = self.id
         AND Customer.cust_type IS NULL;
      END Delete_Customer_OID;

      MEMBER PROCEDURE Delete_Customer_non_OID(
         deleted_attribute IN VARCHAR2) IS

      BEGIN
         DELETE FROM Customer
         WHERE self.<attribute> = deleted_attribute
               AND self.cust_type IS NULL;
      END Delete_Customer_non_OID;

   END;
   /

   CREATE OR REPLACE TYPE BODY Commercial_T AS
```

Figure 5.7. (continued)

```
    MEMBER PROCEDURE Insert_Commercial(
       new_id IN VARCHAR2,
       new_name IN VARCHAR2,
       new_address IN VARCHAR2,
       new_acn IN VARCHAR2) IS

    BEGIN
       INSERT INTO Customer
       VALUES (new_id, new_name, new_address, 'Commercial');
       INSERT INTO Commercial
       VALUES (new_id, new_acn);
    END Insert_Commercial;

    MEMBER PROCEDURE Delete_Commercial_OID IS

    BEGIN
       DELETE FROM Customer
       WHERE Customer.id = self.id
       AND self.cust_type = 'Commercial';
    END Delete_Commercial_OID;

    MEMBER PROCEDURE Delete_Commercial_non_OID(
       deleted_attribute IN VARCHAR2) IS

    BEGIN
       DELETE FROM Customer
       WHERE Customer.id IN
          (SELECT Commercial.id
           FROM Commercial
           WHERE self.<attribute> = deleted_attribute)
           AND self.cust_type = 'Commercial';
    END Delete_Commercial_non_OID;

 END;
 /

 CREATE OR REPLACE TYPE BODY Academic_T AS

    MEMBER PROCEDURE Insert_Academic(
       new_id IN VARCHAR2,
       new_name IN VARCHAR2,
       new_address IN VARCHAR2,
       new_department IN VARCHAR2) IS

    BEGIN
       INSERT INTO Customer
       VALUES (new_id, new_name, new_address, 'Academic');
       INSERT INTO Academic
       VALUES (new_id, new_department);
    END Insert_Academic;

    MEMBER PROCEDURE Delete_Academic_OID IS
```

Figure 5.7. (continued)

```
    BEGIN
       DELETE FROM Customer
       WHERE Customer.id = self.id
       AND self.cust_type = 'Academic';
    END Delete_Academic_OID;

    MEMBER PROCEDURE Delete_Academic_non_OID(
       deleted_attribute IN VARCHAR2) IS

    BEGIN
       DELETE FROM Customer
       WHERE Customer.id IN
          (SELECT Academic.id
           FROM Academic
           WHERE self.<attribute> = deleted_attribute)
           AND self.cust_type = 'Academic';
    END Delete_Academic_non_OID;

 END;
 /
```

Figure 5.8. Mutual-exclusion inheritance table example

Sample Records:

Customer			
ID	**Name**	**Address**	**Cust_Type**
1	Myer Pty Ltd.	Melbourne	Commercial
2	Coles Pty Ltd.	Sydney	Commercial
3	La Trobe Univ.	Bundoora	Academic
4	Monash Univ.	Gippsland	Academic
5	RMIT Univ.	Melbourne	Academic
6	Victoria Univ.	Footscray	Academic
8	***Federal Gov.***	***Canberra***	

Commercial	
ID	**ACN**
1	123-423
2	443-765

Academic	
ID	**Department**
3	Comp. Sc.
4	Info. Tech
5	Comp. Sc.
6	Informatics

one "delete from" statement and varies the OID type to be deleted. Notice also that an optional predicate in which the type is checked appears in the "delete from" statements. This additional predicate is useful to ensure that the OID to be deleted is of the correct subtype. Other than this, the additional predicate imposes an unnecessary overhead. The decision about whether or not to use the additional predicate in the deletion is at the user's discretion.

Finally, in the sample records, as this is a mutual-exclusion example, customer Holmes Institute of ID 7 (see Figure 5.4) does not exist in Figure 5.8. This is because this customer is not mutually exclusive to a subclass type. A nontype customer with ID 8 still exists in the above example.

As in the union inheritance section, we also provide the example of implementation using the newer Oracle™ version (see Figure 5.9). Notice that we do not need subtype tables because the records will be kept in supertype table Customer only.

In this version, the data are kept only in the supertype table and there are no subtype tables created. During insertion, we will need to clarify the type of data that we want to insert. To retrieve the data, we cannot access the attribute of the subtype because there is no column for that attribute in the supertable. Therefore, to access them, we have to use object references such as value. These object references will be introduced in the next chapter.

Figure 5.9. Implementation of mutual-exclusion inheritance relationship in newer Oracle™

```
Methods Declaration

   CREATE OR REPLACE TYPE Customer_T AS OBJECT
      (id             VARCHAR2(10),
       name           VARCHAR2(30),
       address        VARCHAR2(30),
       cust_type      VARCHAR2(15),

      MEMBER PROCEDURE Insert_Customer(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2),

      MEMBER PROCEDURE Delete_Customer) NOT FINAL
   /

   CREATE TABLE Customer OF Customer_T
      (id NOT NULL,
       cust_type  CHECK  (cust_type  in  ('Commercial',  'Academic',
      NULL)),
       PRIMARY KEY (id));
```

Figure 5.9. (continued)

```
   Subtypes Commercial_T and Customer_T are the same as in union
   inheritance (Figure 5.6). However, unlike in union inheritance, no
   subtype table is needed.

Methods Implementation
   CREATE OR REPLACE TYPE BODY Customer_T AS

      MEMBER PROCEDURE Insert_Customer(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2) IS

      BEGIN
         INSERT INTO Customer
         VALUES (Customer_T(new_id, new_name, new_address, NULL));
      END Insert_Customer;

      MEMBER PROCEDURE Delete_Customer IS

      BEGIN
         DELETE FROM Customer
         WHERE Customer.id = self.id;
      END Delete_Customer;

   END;
   /

   CREATE OR REPLACE TYPE BODY Commercial_T AS

      MEMBER PROCEDURE Insert_Commercial(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_acn IN VARCHAR2) IS

      BEGIN
         INSERT INTO Customer
         VALUES
         (Commercial_T(new_id, new_name, new_address,
                     'Commercial', new_acn));
      END Insert_Commercial;

      MEMBER PROCEDURE Delete_Commercial IS

      BEGIN
         DELETE FROM Customer
         WHERE Customer.id = self.id;
      END Delete_Commercial;

   END;
   /

   CREATE OR REPLACE TYPE BODY Academic_T AS

      MEMBER PROCEDURE Insert_Academic(
         new_id IN VARCHAR2,
```

Figure 5.9. (continued)

```
      new_name IN VARCHAR2,
      new_address IN VARCHAR2,
      new_department IN VARCHAR2)IS

   BEGIN
      INSERT INTO Customer
      VALUES
      (Academic_T(new_id, new_name, new_address,
                'Academic', new_department));
   END Insert_Academic;

   MEMBER PROCEDURE DeleteAcademic IS

   BEGIN
      DELETE FROM Customer
      WHERE Customer.id = self.id;
   END DeleteAcademic;

END;
/
```

Implementation of Methods in Partition Inheritance

Similar to the other types of inheritance, mapping partition inheritance into tables in the previous Oracle™ version is done by having one table for each superclass and subclass. Like the mutual-exclusion type, an additional type attribute is added to the superclass table. The difference is that this type attribute has a "not null" constraint. This ensures that each superclass object belongs to a particular subclass type. It also ensures that no superclass object belongs to more than one subclass. Figure 5.10 shows an example of the transformation of partition inheritance.

A number of observations for the above example can be made. First, the relational schemas for partition inheritance are exactly the same as those for mutual-exclusion inheritance, where an additional cust_type attribute is added to the superclass table Customer.

Second, a "not null" constraint is added in the cust_type attribute during the table creation. Other than this, everything regarding the table creation for partition inheritance is identical to that of mutual exclusion. This includes the checking of attribute cust_type, and foreign-key referential integrity for the subclass tables Commercial and Academic.

Third, as a nonspecialized object does not exist in a partition inheritance, insertion into table Customer is not available. In other words, there is no customer object that does not belong to any subclasses. Subclass object insertion (insertion to subclass tables Commercial and Academic) is the same as that of mutual-exclusion inheritance. For practical reasons, it is better to use the encapsulation method of insertion because the insertion to the subclass table is done immediately after the insertion to the superclass table. In other words, there is no record that is inserted only into the superclass table without being inserted into the subclass as well.

Fourth, like insertion, deletion in partition inheritance is applicable to the deletion of subclass objects only (e.g., Commercial and Academic only). The deletion of pure customer objects is not available.

Finally, the sample records show that there is no customer record that does not exist in the subclass tables. Notice that customer Holmes Institute and customer Federal Government do not exist in the sample records due to the above reason.

An implementation example of partition inheritance using the newer Oracle™ version will not be shown here because it is very similar to the mutual-exclusion

Figure 5.10. Implementation of partition inheritance relationship

```
Relational Schemas

   Similar to the one in mutual exclusion (Figure 5.6)

Methods Declaration

   CREATE OR REPLACE TYPE Customer_T AS OBJECT
      (id           VARCHAR2(10),
       name         VARCHAR2(30),
       address      VARCHAR2(30),
       cust_type    VARCHAR2(15))
   /

   CREATE TABLE Customer OF Customer_T
      (id NOT NULL,
       cust_type NOT NULL
          CHECK (cust_type in ('Commercial', 'Academic')),
       PRIMARY KEY (id));

   The creation of commercial and academic subtypes and tables is the same
   as in mutual-exclusion inheritance (Figure 5.6).

Methods Implementation

   Methods implementation for commercial and academic subtypes is the same
   as in mutual-exclusion inheritance (Figure 5.6).
```

Figure 5.11. Partition inheritance table example

Sample Records:

Customer			
ID	**Name**	**Address**	**CustType**
1	Myer Pty Ltd.	Melbourne	Commercial
2	Coles Pty Ltd.	Sydney	Commercial
3	LaTrobe Univ.	Bundoora	Academic
4	Monash Univ.	Gippsland	Academic
5	RMIT Univ.	Melbourne	Academic
6	Victoria Univ.	Footscray	Academic

Commercial	
ID	**ACN**
1	123-423
2	443-765

Academic	
ID	**Department**
3	Comp. Sc.
4	Info. Tech
5	Comp. Sc.
6	Informatics

implementation in Figure 5.10. The only difference is the "not null" constraint for the cust_type attribute during the table creation.

Implementation of Methods in Multiple Inheritance

Mapping multiple inheritance to tables is similar to that of union inheritance; that is, no additional type attribute is necessary. We use the previous example and assume that a new class Private_Ed inherits from classes Commercial and Academic. As a result, a table for the new class is created. Figure 5.12 shows an example of transforming multiple inheritance.

The following are the observations relating to the multiple-inheritance examples.

First, the relational schemas are identical to those of union inheritance with the exception that here a new table is created to accommodate the subclass inheriting from multiple superclasses. In this case, table Private_Ed is created.

Second, as the relational schemas for the first three tables are the same as those in union inheritance, the table-creation statements for these tables are also the same. The new table has its own create-table statement. One thing to note is that the foreign key of this new table refers to table Customer only, although in fact it has references to tables Commercial and Academic. However, in the implementation, SQL allows one reference per foreign key.

Figure 5.12. Implementation of multiple inheritance relationship

```
Relational Schemas

   Customer (ID, name, address)
   Commercial (ID, ACN)
   Academic (ID, department)
   Private_Ed (ID, sponsor_board)

Methods Declaration

   The creation of customer, commercial, and academic types is the same as
   in union inheritance (Figure 5.3).

   CREATE OR REPLACE TYPE Private_Ed_T AS OBJECT
      (id           VARCHAR2 (10),
       sponsor_board VARCHAR2(30),

       MEMBER PROCEDURE Insert_Private_Ed(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_acn IN VARCHAR2,
         new_department IN VARCHAR2,
         new_sponsor_board IN VARCHAR2),

       MEMBER PROCEDURE Delete_Private_Ed)
   /

   CREATE TABLE Private_Ed OF Private_Ed_T
      (id NOT NULL,
       PRIMARY KEY (id),
       FOREIGN KEY (id) REFERENCES Customer (id) ON DELETE CASCADE);

Methods Implementation

   Methods implementation for customer, commercial, and academic is the
   same as in union inheritance (Figure 5.3).

   CREATE OR REPLACE TYPE BODY Private_Ed_T AS

      MEMBER PROCEDURE Insert_Private_Ed(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_acn IN VARCHAR2,
         new_department IN VARCHAR2,
         new_sponsor_board IN VARCHAR2) IS

      BEGIN
         INSERT INTO Customer
         VALUES (new_id, new_name, new_address);
         INSERT INTO Commercial
         VALUES (new_id, new_acn);
         INSERT INTO Academic
         VALUES (new_id, new_department);
         INSERT INTO Private_Ed
         VALUES (new_id, new_sponsor_board);
      END Insert_Private_Ed;
      MEMBER PROCEDURE Delete_Private_Ed IS

      BEGIN
         DELETE FROM Customer
         WHERE Customer.id = self.id;
      END Delete_Private_Ed;

   END;
/
```

Figure 5.13. Multiple inheritance table example

Sample Records:

Customer		
ID	**Name**	**Address**
1	Myer Pty Ltd.	Melbourne
2	Coles Pty Ltd.	Sydney
3	LaTrobe Univ.	Bundoora
4	Monash Univ.	Gippsland
5	RMIT Univ.	Melbourne
6	Victoria Univ.	Footscray
7	***Holmes Inst.***	***Melbourne***
8	Federal Gov.	Canberra

Commercial	
ID	**ACN**
1	123-423
2	443-765
7	***011-333***

Academic	
ID	**Department**
3	Comp. Sc.
4	Info. Tech
5	Comp. Sc.
6	Informatics
7	***Info. Studies***

Private_Ed	
ID	**Sponsor_Board**
7	***Pratt Brothers***

Third, the insertion of the first three tables is also identical to that of union inheritance. The new insertion is applied to the new table. Notice that the insertion order is top down from the least specialized class (table Customer) to the table Private_Ed.

Fourth, like creation and insertion, the deletion of the first three tables is the same as that of union inheritance. The deletion of records from the new table is simplified, however, so as to delete the root record only. The effect is that matching records of all tables underneath this root table will be deleted as well due to the foreign-key referential integrity constraint where deletion is cascaded.

Finally, the sample records show that customer Holmes Institute appears in the new table Private_Ed. It shows that Holmes Institute is a commercial customer as well as an academic customer. It also belongs to the category private educational institution, where private educational institution is both commercial and academic.

We do not provide the implementation of multiple inheritance using newer Oracle™ versions because at the time of this writing, there is still no support for multiple inheritance.

Implementation of Methods in Association Relationships

In this section, the object-oriented model as described in Chapter III, Figure 3.35, will be used. We are going to concentrate on the association relationships between Author_T and Course Manual_T, which represents a many-to-many association, and between Teaching_Staff_T and Subject_T, which depicts a one-to-many association. Figure 5.14 shows the many-to-many association relationship.

In this section, we focus on the implementation of association methods using the latest object-relational technology, namely, object references. Please note that the keyword ref is used to represent object references, as opposed to *references*, which is used to represent the traditional way of representing association using the foreign-key relationship.

Also note that we do not use the ao_ID and ISBN as the composite key in the Publish table. This is to distinguish between the two concepts of foreign-key references (using ID) and the object-references concept where internal references are used. One weakness of this practice is that there is no restriction of duplication of the same record as it would be restricted when we use primary keys in a pure relational system. For example (see Figure 5.15), we cannot insert the same record for the Author and Course_Manual tables, but we can insert duplication to the Publish table.

In the insertion example above, the select statements must return one row only. Thus, the specified attribute must be unique. It is recommended that a unique ID be used here. The insertion method for encapsulation is also not all straightforward from the simple generic method. We need to use variables in order to be able to insert them into the Publish table.

Figure 5.14. Many-to-many association

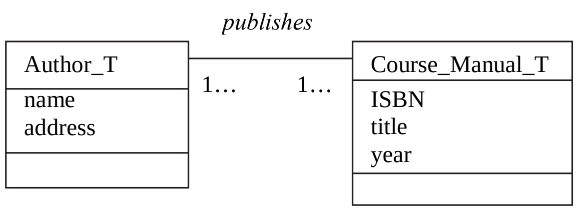


Figure 5.15. Implementation of a many-to-many association relationship

```
Relational Schemas

   Author (ao_ID, name, address)
   Course_Manual (ISBN, title, year)
   Publish (author, course_manual)

Methods Declaration

   CREATE OR REPLACE TYPE Author_T AS OBJECT
      (ao_id   VARCHAR2(3),
       name    VARCHAR2(10),
       address VARCHAR2(20),

       MEMBER PROCEDURE Insert_Author(
         new_ao_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2),

       MEMBER PROCEDURE Delete_Author)
   /

   CREATE TABLE Author OF Author_T
      (ao_id NOT NULL,
       PRIMARY KEY (ao_id));

   CREATE OR REPLACE TYPE Course_Manual_T AS OBJECT
      (isbn    VARCHAR2(10),
       title   VARCHAR2(20),
       year    NUMBER,

       MEMBER PROCEDURE Insert_Course_Manual(
         new_isbn IN VARCHAR2,
         new_title IN VARCHAR2,
         new_year IN NUMBER),

       MEMBER PROCEDURE Delete_Course_Manual)
   /

   CREATE TABLE Course_Manual OF Course_Manual_T
      (isbn NOT NULL,
       PRIMARY KEY (isbn));

   CREATE TABLE Publish
      (author REF Author_T,
       course_manual REF Course_Manual_T);

Methods Implementation

   CREATE OR REPLACE TYPE BODY Author_T AS

      MEMBER PROCEDURE Insert_Author(
         new_ao_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2) IS
```

Figure 5.15. (continued)

```
   BEGIN
      INSERT INTO Author
      VALUES (new_ao_id, new_name, new_address);
   END Insert_Author;

   MEMBER PROCEDURE Delete_Author IS

   BEGIN
      DELETE FROM Publish
      WHERE Publish.author IN
         (SELECT REF(a)
          FROM Author a
          WHERE a.ao_id = self.author_id);

      DELETE FROM Author
      WHERE Author.ao_id = self.author_id;

   END Delete_Author;

END;
/

CREATE OR REPLACE TYPE BODY Course_Manual_T AS

   MEMBER PROCEDURE Insert_Course_Manual(
      new_isbn IN VARCHAR2,
      new_title IN VARCHAR2,
      new_year IN NUMBER) IS

   BEGIN
      INSERT INTO Course_Manual
      VALUES (new_isbn, new_title, new_year);
   END Insert_Course_Manual;

   MEMBER PROCEDURE Delete_Course_Manual IS

   BEGIN
      DELETE FROM Publish
      WHERE Publish.course_manual IN
         (SELECT REF(b)
          FROM Course_Manual b
          WHERE b.isbn = self.course_id);

      DELETE FROM Course_Manual
      WHERE Course_Manual.isbn = self.course_id;

   END Delete_Course_Manual;

END;
/

For the Publish table, we do not have member procedures so we use
ordinary stored procedures.

CREATE OR REPLACE PROCEDURE Insert_Publish(
   new_ao_id IN VARCHAR2,
   new_isbn IN VARCHAR2) AS
```

Figure 5.15. (continued)

```
      author_temp REF Author_T;
      course_temp REF Course_Manual_T;

      BEGIN
         SELECT REF(a) INTO author_temp
         FROM Author a
         WHERE a.ao_id = new_ao_id;

         SELECT REF(b) INTO course_temp
         FROM Course_Manual b
         WHERE b.isbn = new_isbn;

         INSERT INTO Publish
         VALUES (author_temp, course_temp);
      END Insert_Publish;
   /

   CREATE OR REPLACE PROCEDURE Delete_Publish(
      deleted_ao_id IN VARCHAR2,
      deleted_isbn IN VARCHAR2) AS

      BEGIN
         DELETE FROM Publish
         WHERE
            Publish.author IN
            (SELECT REF(a)
             FROM Author a
             WHERE a.ao_id = deleted_ao_id) AND
            Publish.course_manual IN
            (SELECT REF(b)
             FROM Course_Manual b
             WHERE b.isbn = deleted_isbn);
      END Delete_Publish;
   /
```

Methods Execution Example

For this method we use member procedures for the Author and Course_Manual tables, and use stored procedure for the Publish table.

```
DECLARE
      a_author Author_T := Author_T(NULL,NULL,NULL);
      a_course_manual Course_Manual_T :=
         Course_Manual_T(NULL,NULL,NULL);

   BEGIN
      a_author.Insert_Author ('S2', 'Smith', 'Sydney');
      a_course_manual.Insert_Course_Manual
            ('1234', 'Database System', 2002);

   END;
   /

   EXECUTE Insert_Publish('S2', '1234');
```

Figure 5.16. Example of a many-to-many association-relationship table

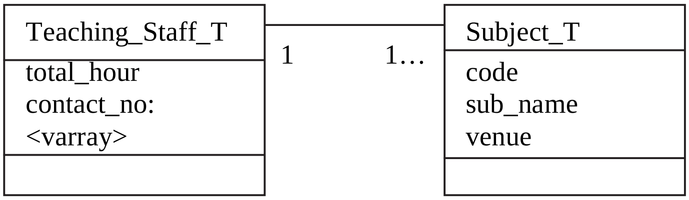
Sample Records:

Author		
ao_id	**name**	**address**
S2	Smith	Sydney

Course_Manual		
isbn	**title**	**year**
1234	Database System	2002

Publish	
ao_id	**isbn**
S2	1234

Figure 5.17. One-to-many association



The next example is the one-to-many association relationship (see Figure 5.17). The main difference in the implementation between this and the many-to-many association is the fact that the ref attribute that forms the object references is placed in the object that holds the many side. In the Figure 5.17 example, Subject_T will hold the object reference to Teaching_Staff_T. The implementation of this association is shown in Figure 5.18.

Implementation of Methods in Aggregation Relationships

In this section, we will concentrate on the aggregation relationship between Course_Manual_T and Chapter_T of the object-oriented model described in Figure 5.19.

As mentioned previously, there are two ways of implementing aggregation relationships in Oracle™: the clustering technique and the nesting technique. The associated methods to be implemented for each aggregation relationship will be dependent on the technique used to represent the aggregation link.

Figure 5.18. Implementation of a one-to-many association relationship

Relational Schemas

```
Teaching_Staff (ao_ID, total_hour, contact_no)
Subject (code, sub_name, venue, lecturer)
```

Methods Declaration

```
CREATE OR REPLACE TYPE Contacts AS VARRAY(3) OF NUMBER
/

CREATE OR REPLACE TYPE Teaching_Staff_T AS OBJECT
   (ao_id              VARCHAR2(3),
    total_hour         NUMBER,
    contact_no         CONTACTS,

    MEMBER PROCEDURE Insert_Teaching_Staff(
      new_ao_id IN VARCHAR2,
      new_ttl_hour IN NUMBER,
      new_number1 IN NUMBER,
      new_number2 IN NUMBER,
      new_number3 IN NUMBER),

    MEMBER PROCEDURE Delete_Teaching_Staff)
/

CREATE OR REPLACE TYPE Subject_T AS OBJECT
   (code          VARCHAR2(10),
    sub_name      VARCHAR2(20),
    venue         VARCHAR2(10),
    lecturer REF teaching_staff_T,

   MEMBER PROCEDURE Insert_Subject(
      new_code IN VARCHAR2,
      new_sub_name IN VARCHAR2,
      new_venue IN VARCHAR2,
      teach_ao_id IN VARCHAR2),

   MEMBER PROCEDURE Delete_Subject)
/

CREATE TABLE Teaching_Staff OF Teaching_Staff_T
   (ao_id NOT NULL,
    PRIMARY KEY (ao_id));

CREATE TABLE Subject OF Subject_T
   (code NOT NULL,
    PRIMARY KEY (code));
```

Methods Implementation

```
CREATE OR REPLACE TYPE BODY Teaching_Staff_T AS
```

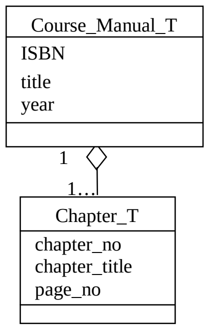
Figure 5.18. (continued)

```
   MEMBER PROCEDURE Insert_Teaching_Staff(
      new_ao_id IN VARCHAR2,
      new_ttl_hour IN NUMBER,
      new_number1 IN NUMBER,
      new_number2 IN NUMBER,
      new_number3 IN NUMBER) IS

   BEGIN
      INSERT INTO Teaching_Staff
      VALUES (new_ao_id, new_ttl_hour, Contacts
             (new_number1, new_number2, new_number3));
   END Insert_Teaching_Staff;

   MEMBER PROCEDURE Delete_Teaching_Staff IS

   BEGIN
      DELETE FROM Subject b
      WHERE b.lecturer.ao_id = self.ao_id;
      DELETE FROM Teaching_Staff a
      WHERE a.ao_id = self.ao_id;
   END Delete_Teaching_Staff;

END;
/

CREATE OR REPLACE TYPE BODY Subject_T AS

   MEMBER PROCEDURE Insert_Subject(
      new_code IN VARCHAR2,
      new_sub_name IN VARCHAR2,
      new_venue IN VARCHAR2,
      teach_ao_id IN VARCHAR2) IS

      new_lecturer REF Teaching_Staff_T;

      BEGIN
         SELECT REF(a) INTO new_lecturer
         FROM Teaching_Staff a
         WHERE ao_id = teach_ao_id;

         INSERT INTO Subject
         VALUES (new_code, new_sub_name, new_venue,
                new_lecturer);
      END Insert_Subject;

   MEMBER PROCEDURE Delete_Subject IS
```

Figure 5.18. (continued)

```
        BEGIN
           DELETE FROM Subject b
           WHERE b.code = self.code;
        END Delete_Subject;

END;
/
```

Figure 5.19. Aggregation relationship



Implementation of Methods in Aggregation Relationships Using the Clustering Technique

In the clustering technique, the primary key of the whole table is specified as the cluster key. This key will be the one that groups the part tables together. Physically, this key is stored only once, and connected to it will be all the part records that are associated with it. The implementation of Figure 5.19 using the clustering technique is described in Figure 5.20.

The relational schemas for the clustering technique show that the cluster key, ISBN (the primary key of the whole table), will be carried by each of the part tables. If we have more than one part table in the example, then each of them will have ISBN as one of the attributes. Note that c_no by itself is not a primary key (not unique) for the Chapter table; however, c_no combined with ISBN is unique within the Chapter table.

The generic method of implementation in relational tables using the clustering technique is not different from that of the standard insert, delete, and update procedures. We can simply ignore the cluster when we perform the data manipulation as a cluster key is an internal structure needed to make data storage and retrieval more efficient when a particular model is implemented.

Implementation of Methods in Aggregation Relationships Using the Nesting Technique

Figure 5.22 shows the nested-table technique. The attribute chapter within the Course_Manual table is an object reference that is referencing to a nested table called Chapter. We cannot place the attribute of Chapter (e.g., c_no) in Course_Manual as we usually do in a foreign-key relationship. This is because c_no is not a primary key of Chapter. There may be course manuals with the same chapter numbers but entirely different contents. In addition, the link between Course_Manual and Chapter is established through object references

Figure 5.20. Implementation of an aggregation relationship using the clustering technique

Relational Schemas

```
Course_Manual (ISBN, title, year)
Chapter (ISBN, c_no, c_title, c_page_no)
```

SQL Create Statements

The following create statements show how we create the cluster, tables, and index. It has been explained in Section 3.3.

```
CREATE CLUSTER CM_Cluster
   (isbn    VARCHAR2(10));

CREATE TABLE Course_Manual
   (isbn    VARCHAR2(10) NOT NULL,
    title   VARCHAR2(20),
    year    NUMBER,
    PRIMARY KEY (isbn))
   CLUSTER CM_Cluster(isbn);

CREATE TABLE Chapter
   (isbn          VARCHAR2(10) NOT NULL,
    c_no          VARCHAR2(10) NOT NULL,
    c_title       VARCHAR2(25),
```

Figure 5.20. (continued)

```
      c_page_no     NUMBER,
      PRIMARY KEY (isbn, c_no),
      FOREIGN KEY (isbn) REFERENCES Course_Manual(isbn))
     CLUSTER CM_Cluster(isbn);

  CREATE INDEX CM_Cluster_Index
     ON CLUSTER CM_Cluster;

Methods Implementation

  CREATE OR REPLACE PROCEDURE Insert_Course_Manual(
     new_isbn IN VARCHAR2,
     new_title IN VARCHAR2,
     new_year IN NUMBER) AS

     BEGIN
        INSERT INTO Course_Manual
        VALUES (new_isbn, new_title, new_year);
     END Insert_Course_Manual;
  /

  CREATE OR REPLACE PROCEDURE Insert_Chapter(
     new_isbn IN VARCHAR2,
     new_c_no IN NUMBER,
     new_c_title IN VARCHAR2,
     new_c_page_no IN NUMBER) AS

     BEGIN
        INSERT INTO CHAPTER
        VALUES (new_isbn, new_c_no, new_c_title,
               new_c_page_no);
     END Insert_Chapter;

  CREATE OR REPLACE PROCEDURE Delete_Course_Manual(
     deleted_isbn IN VARCHAR2) AS

     BEGIN
        DELETE FROM Course_Manual
        WHERE isbn = deleted_isbn;
     END Delete_Course_Manual;
  /

  CREATE OR REPLACE PROCEDURE Delete_Chapter(
     deleted_c_no IN NUMBER) AS

     BEGIN
        DELETE FROM Chapter
        WHERE c_no = deleted_c_no;
     END Delete_Chapter;
  /
```

Figure 5.20. (continued)

```
Methods Execution Example

As using the clustering technique, we do not have member procedures; to
execute the procedures we just use execute statements.

   EXECUTE Insert_Course_Manual ('1234', 'Database System', 2002);

   EXECUTE Insert_Chapter ('1234', 1, 'Introduction', 10);

   EXECUTE Insert_Chapter ('1234', 2, 'Database Concepts', 30);
```

Figure 5.21. Clustering aggregation-relationship table example

Sample Records:

Course_Manual		
ISBN	**Title**	**Year**
1234	Database System	2002

Chapter			
ISBN	**C_No**	**C_Title**	**Page_No**
1234	1	Introduction	20
1234	2	Database Concepts	50

(the internal reference of each individual chapter) rather than through the attribute value. Note also that there is no primary key in the Chapter table.

From the implementation above, note that we cannot insert new chapters without an associating course manual. This enforces the existence-dependent concept, where the existence of each part object is dependent on its associated whole object.

The keyword *the* in the above insert statement is used to represent the nested table Chapter. Since Chapter is not a standard table, we cannot use its name in order to populate it. The use of *the* (we can also use *table* instead) also ensures that each record within the nested table has an associated record from the whole table, in this case the Course_Manual table. Note also that the select statement must return one row only; otherwise, the query will return an error. To avoid this problem, we usually use a primary key as the selection attribute to ensure uniqueness.

Figure 5.23 describes the relationship between the whole table and its nested table (part table).

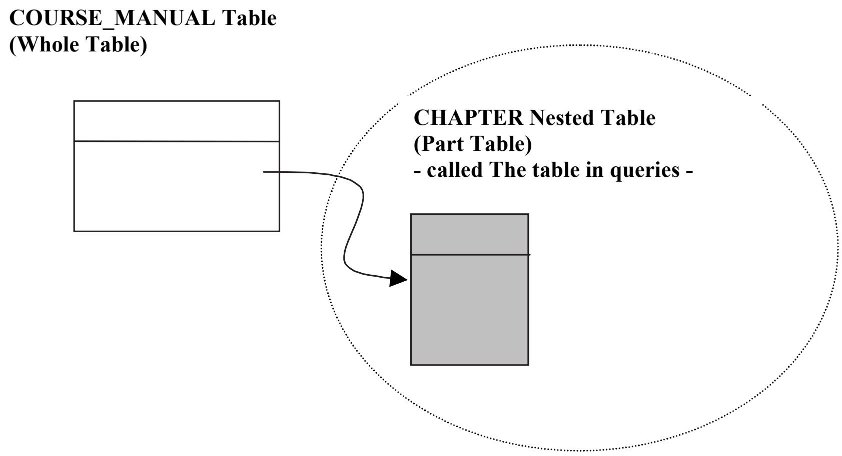
Figure 5.22. Implementation of an aggregation relationship using the nesting technique

```
Relational Schemas

   Course_Manual (ISBN, title, year, chapter)
   Chapter (c_no, c_title, page_no)

Methods Declaration

   CREATE OR REPLACE TYPE Chapter_T AS OBJECT
      (c_no          NUMBER,
       c_title       VARCHAR2(20),
       c_page_no     NUMBER)
   /

   CREATE OR REPLACE TYPE Chapter_Table_T AS TABLE OF Chapter_T
   /

   CREATE OR REPLACE TYPE Course_Manual_T AS OBJECT
      (isbn          VARCHAR2(10),
       title         VARCHAR2(20),
       year          NUMBER,
       chapter       Chapter_Table_T,

       MEMBER PROCEDURE Insert_Course_Manual(
         new_isbn IN VARCHAR2,
         new_title IN VARCHAR2,
         new_year IN NUMBER,
         new_c_no IN NUMBER,
         new_c_title IN VARCHAR2,
         new_c_page_no IN NUMBER),

      MEMBER PROCEDURE Delete_Course_Manual,

      MEMBER PROCEDURE Insert_Chapter(
         new_isbn IN VARCHAR2,
         new_c_no IN NUMBER,
         new_c_title IN VARCHAR2,
         new_c_page_no IN NUMBER),

      MEMBER PROCEDURE Delete_Chapter)

   /

   CREATE TABLE Course_Manual OF Course_Manual_T
      (isbn NOT NULL,
       PRIMARY KEY (isbn))
      NESTED TABLE chapter STORE AS chapter_tab;

Methods Implementation

   CREATE OR REPLACE TYPE BODY Course_Manual_T AS
```

Figure 5.22. (continued)

```
   MEMBER PROCEDURE Insert_Course_Manual(
      new_isbn IN VARCHAR2,
      new_title IN VARCHAR2,
      new_year IN NUMBER,
      new_c_no IN NUMBER,
      new_c_title IN VARCHAR2,
      new_c_page_no IN NUMBER) IS

   BEGIN
      INSERT INTO Course_Manual
      VALUES (new_isbn, new_title, new_year,
            Chapter_Table_T(Chapter_T(new_c_no, new_c_title,
            new_c_page_no)));
   END Insert_Course_Manual;

   MEMBER PROCEDURE Delete_Course_Manual

   BEGIN
      DELETE FROM Course_Manual a
      WHERE a.isbn = self.isbn;
   END Delete_Course_Manual;

   MEMBER PROCEDURE Insert_Chapter(
      new_isbn IN VARCHAR2,
      new_c_no IN NUMBER,
      new_c_title IN VARCHAR2,
      new_c_page_no IN NUMBER) IS

   BEGIN
      INSERT INTO THE
         (SELECT c.chapter
          FROM Course_Manual c
          WHERE c.isbn = new_isbn)
      VALUES (new_c_no, new_c_title, new_c_page_no);
   END Insert_Chapter;

   MEMBER PROCEDURE Delete_Chapter IS

   BEGIN
      DELETE FROM THE
         (SELECT c.chapter
          FROM Course_Manual c
          WHERE c.isbn = self.isbn);
   END Delete_Chapter;

END;
/
```

Figure 5.23. "The" table



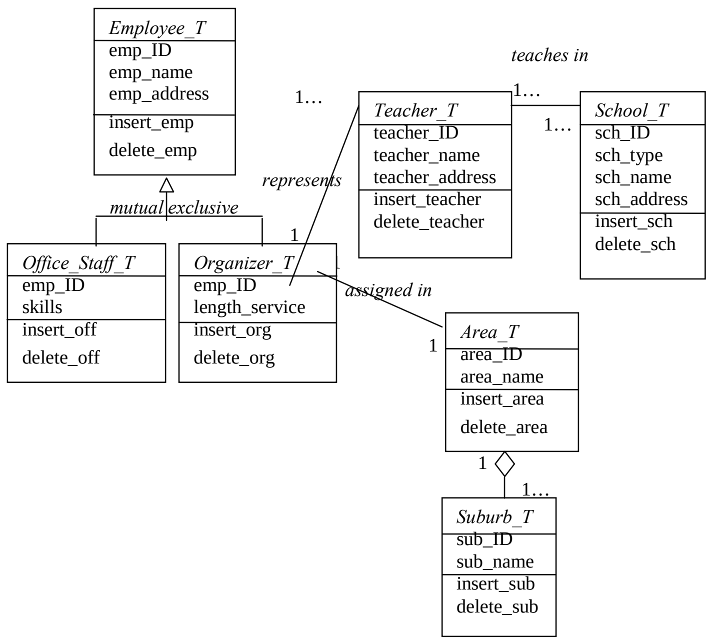
Case Study

Recall the AEU case study in Chapter 1. The union wants to add generic methods in its object-relational database. However, only objects with frequent changes will have member procedures attached to them. Figure 5.24 shows the partition of the AEU database diagram with the object attributes and object methods.

To implement the object-oriented model, we will follow the systematic steps that follow.

- Type and table. For this case, we need the types Employee_T, Office_Staff_T, Organizer_T, Teacher_T, and School_T. For each of them, we will create the table respectively. For this case study, we will use a nested table; thus, we need to add type Area_T and its table, and also the Suburb_T type and Suburb_Table_T for the aggregation relationship.
- Inheritance relationship. There is one mutual-exclusion inheritance relationship between Employee_T and its subclasses. We have to add another attribute in the Employee_T class, emp_type, to perform the mutual-exclusive feature.

Figure 5.24. AEU case study with generic-method implementation



- Association relationship. There are three association relationships from this model. First is the one-to-many association between Organizer_T and Teacher_T. A ref of the one side, Organizer_T, is needed in the many side. Next, the association is many to many between Teacher_T and School_T. For this association relationship, we need to add a table to keep the ref to both classes. Finally, there is a one-to-one association between Organizer_T and Area_T. We will use the object reference of Organizer_T in Area_T because Area_T has total participation.
- Aggregation relationship. There is one homogeneous aggregation relationship in this model. We will use a nested table, so we have to create the type and type table for the part class, and the type and table for the whole class.
- Complete solution. The complete solution is shown in Figure 5.25.

Figure 5.25. Implementation of the case study in Oracle™

```
Methods Declaration

   CREATE OR REPLACE TYPE Employee_T AS OBJECT
      (emp_id        VARCHAR2(10),
       emp_name      VARCHAR2(30),
       emp_address   VARCHAR2(30),
       emp_type      VARCHAR2(15),

       MEMBER PROCEDURE Insert_Emp(
         new_emp_id IN VARCHAR2,
         new_emp_name IN VARCHAR2,
         new_emp_address IN VARCHAR2),

       MEMBER PROCEDURE Delete_Emp) NOT FINAL
   /

   CREATE TABLE Employee OF Employee_T
      (emp_id NOT NULL,
       emp_type CHECK (emp_type IN ('Office Staff', 'Organizer', NULL)),
       PRIMARY KEY (emp_id));

   CREATE OR REPLACE TYPE Office_Staff_T UNDER Employee_T
      (skills  VARCHAR2(50),

       MEMBER PROCEDURE Insert_Off(
         new_emp_id IN VARCHAR2,
         new_emp_name IN VARCHAR2,
         new_emp_address IN VARCHAR2,
         new_skills IN VARCHAR2),

      MEMBER PROCEDURE Delete_Off)
   /

   CREATE OR REPLACE TYPE Organizer_T UNDER Employee_T
      (length_service      VARCHAR2(10),

       MEMBER PROCEDURE Insert_Org(
         new_emp_id IN VARCHAR2,
         new_emp_name IN VARCHAR2,
         new_emp_address IN VARCHAR2,
         new_length_service IN VARCHAR2),

       MEMBER PROCEDURE Delete_Org)
      /

   CREATE OR REPLACE TYPE Teacher_T AS OBJECT
      (teacher_id          VARCHAR2(10),
       teacher_name        VARCHAR2(20),
       teacher_address     VARCHAR2(10),
       representation REF Organizer_T,

       MEMBER PROCEDURE Insert_Teacher(
         new_teacher_id IN VARCHAR2,
         new_teacher_name IN VARCHAR2,
         new_teacher_address IN VARCHAR2,
         representation_emp_id IN VARCHAR2),
```

Figure 5.25. (continued)

```
    MEMBER PROCEDURE Delete_Teacher)
/

CREATE TABLE Teacher OF Teacher_T
   (teacher_id NOT NULL,
    PRIMARY KEY (teacher_id));

CREATE OR REPLACE TYPE School_T AS OBJECT
   (sch_id        VARCHAR2(10),
    sch_name      VARCHAR2(20),
    sch_address   VARCHAR2(30),
    sch_type      VARCHAR2(15),

    MEMBER PROCEDURE Insert_Sch(
      new_sch_id IN VARCHAR2,
      new_sch_name IN VARCHAR2,
      new_sch_address IN VARCHAR2,
      new_sch_type IN VARCHAR2),

   MEMBER PROCEDURE Delete_Sch)
/

CREATE TABLE School OF School_T
   (sch_id NOT NULL,
    sch_type CHECK (sch_type IN ('Primary', 'Secondary', 'TechC')),
    PRIMARY KEY (sch_id));

CREATE TABLE Teach_In
   (teacher REF Teacher_T,
    school REF School_T);


CREATE OR REPLACE TYPE Suburb_T AS OBJECT
   (sub_id        VARCHAR2(10),
    sub_name      VARCHAR2(20))

/

CREATE OR REPLACE TYPE Suburb_Table_T AS TABLE OF Suburb_T
/

CREATE OR REPLACE TYPE Area_T AS OBJECT
   (area_id       VARCHAR2(10),
    area_name     VARCHAR2(20),
    suburb        Suburb_Table_T,
    assigned_org REF Organizer_T,

    MEMBER PROCEDURE Insert_Area(
      new_area_id IN VARCHAR2,
      new_area_name IN VARCHAR2,
      new_sub_id IN VARCHAR2,
      new_sub_name IN VARCHAR2,
      assigned_org_id IN VARCHAR2),
```

Figure 5.25. (continued)

```
        MEMBER PROCEDURE Delete_Area,

        MEMBER PROCEDURE Insert_Suburb(
          new_area_id IN VARCHAR2,
          new_sub_id IN VARCHAR2,
          new_sub_name IN VARCHAR2),

        MEMBER PROCEDURE Delete_Suburb
    /

    CREATE TABLE Area OF Area_T
       (area_id NOT NULL,
        PRIMARY KEY (area_id))
       NESTED TABLE suburb STORE AS suburb_tab;

Methods Implementation
    CREATE OR REPLACE TYPE BODY Employee_T AS

       MEMBER PROCEDURE Insert_Emp(
          new_emp_id IN VARCHAR2,
          new_emp_name IN VARCHAR2,
          new_emp_address IN VARCHAR2) IS

       BEGIN
          INSERT INTO Employee
          VALUES (new_emp_id, new_emp_name, new_emp_address, NULL);
       END Insert_Emp;

       MEMBER PROCEDURE Delete_Emp IS

       BEGIN
          DELETE FROM Employee
          WHERE Employee.emp_id = self.emp_id
          AND Employee.emp_type IS NULL;
       END Delete_Emp;

    END;
    /

    CREATE OR REPLACE TYPE BODY Office_Staff_T AS

       MEMBER PROCEDURE Insert_Off(
          new_emp_id IN VARCHAR2,
          new_emp_name IN VARCHAR2,
          new_emp_address IN VARCHAR2,
          new_skills IN VARCHAR2) IS

       BEGIN
          INSERT INTO Employee
          VALUES Office_Staff_T(new_emp_id, new_emp_name, new_emp_address,
                'Office_Staff', new skills);
       END Insert_Off;

       MEMBER PROCEDURE Delete_Off IS

       BEGIN
```

Figure 5.25. (continued)

```
        DELETE FROM Employee
        WHERE Employee.emp_id = self.emp_id;
     END Delete_Off;

     END;
/

 CREATE OR REPLACE TYPE BODY Organizer_T AS

     MEMBER PROCEDURE Insert_Org(
        new_emp_id IN VARCHAR2,
        new_emp_name IN VARCHAR2,
        new_emp_address IN VARCHAR2,
        new_length_service IN VARCHAR2) IS

     BEGIN
        INSERT INTO Employee
        VALUES (new_emp_id, new_emp_name,
               new_emp_address, 'Organizer', new_length_service);
     END Insert_Org;

     MEMBER PROCEDURE Delete_Org IS

     BEGIN
        DELETE FROM Employee
        WHERE Employee.emp_id = self.emp_id;
     END Delete_Org;

END;
/

 CREATE OR REPLACE TYPE BODY Teacher_T AS

     MEMBER PROCEDURE Insert_Teacher(
        new_teacher_id IN VARCHAR2,
        new_teacher_name IN VARCHAR2,
        new_teacher_address IN VARCHAR2,
        representation_emp_id IN VARCHAR2) IS

     new_organizer REF Organizer_T;

     BEGIN
        SELECT REF(a) INTO new_organizer
        FROM Organizer a
        WHERE emp_id = representation_emp_id;

        INSERT INTO Teacher
        VALUES (new_teacher_id, new_teacher_name,
               new_teacher_address, new_organizer);
     END Insert_Teacher;
```

Figure 5.25. (continued)

```
   MEMBER PROCEDURE Delete_Teacher IS

      BEGIN
         DELETE FROM Teacher
         WHERE Teacher.teacher_id = self.teacher_id;
      END Delete_Teacher;

END;
/

CREATE OR REPLACE TYPE BODY School_T AS

   MEMBER PROCEDURE Insert_Sch(
      new_sch_id IN VARCHAR2,
      new_sch_name IN VARCHAR2,
      new_sch_address IN VARCHAR2,
      new_sch_type IN VARCHAR2) IS

   BEGIN
      INSERT INTO School
      VALUES (new_sch_id, new_sch_name, new_sch_address, new_sch_type);
   END Insert_Sch;

   MEMBER PROCEDURE Delete_Sch IS

   BEGIN
      DELETE FROM Teach_In
      WHERE Teach_In.school IN
         (SELECT REF(a)
          FROM School a
          WHERE a.sch_id = self.sch_id);

      DELETE FROM School
      WHERE School.sch_id = self.sch_id;
   END Delete_Sch;

END;
/

CREATE OR REPLACE PROCEDURE Insert_Teach_In(
   new_teacher_id IN VARCHAR2,
   new_sch_id IN VARCHAR2) AS

   teacher_temp REF Teacher_T;
   school_temp REF School_T;

   BEGIN
      SELECT REF(a) INTO teacher_temp
      FROM Teacher a
      WHERE a.teacher_id = new_teacher_id;

      SELECT REF(b) INTO school_temp
      FROM School b
      WHERE b.sch_id = new_sch_id;

      INSERT INTO Teach_In
      VALUES (teacher_temp, school_temp);
```

Figure 5.25. (continued)

```
   END Insert_Teach_In;
/

CREATE OR REPLACE PROCEDURE Delete_Teach_In(
   deleted_teacher_id IN VARCHAR2,
   deleted_sch_id IN VARCHAR2) AS

   BEGIN
      DELETE FROM Teach_In
      WHERE
         Teach_In.teacher IN
         (SELECT REF(a)
          FROM Teacher a
          WHERE a.teacher_id = deleted_teacher_id) AND
         Teach_In.school IN
         (SELECT REF(b)
          FROM School b
          WHERE b.sch_id = deleted_sch_id);
   END Delete_Teach_In;
/

CREATE OR REPLACE TYPE BODY Area_T AS

   MEMBER PROCEDURE Insert_Area(
      new_area_id IN VARCHAR2,
      new_area_name IN VARCHAR2,
      new_sub_id IN VARCHAR2,
      new_sub_name IN VARCHAR2,
      assigned_org_id IN VARCHAR2) IS

   new_organizer REF Organizer_T;

   BEGIN
      SELECT REF(a) INTO new_organizer
      FROM Organizer a
      WHERE emp_id = assigned_org_id;

      INSERT INTO Area
      VALUES (new_area_id, new_area_name,
            Suburb_Table_T(Suburb_T(new_sub_id,
            new_sub_name)),new_organizer);
   END Insert_Area;

   MEMBER PROCEDURE Delete_Area IS

   BEGIN
      DELETE FROM Area a
      WHERE a.area_id = self.area_id;
   END Delete_Area;
```

Figure 5.25. (continued)

```
    MEMBER PROCEDURE Insert_Suburb(
       new_area_id IN VARCHAR2,
       new_sub_id IN VARCHAR2,
       new_sub_name IN VARCHAR2) IS

    BEGIN
       INSERT INTO THE
          (SELECT a.Suburb
           FROM Area a
           WHERE a.area_id = new_area_id)
       VALUES (new_sub_id, new_sub_name);
    END Insert_Suburb;

    MEMBER PROCEDURE Delete_Suburb IS

    BEGIN
       DELETE FROM THE
          (SELECT a.Suburb
           FROM Area a
           WHERE a.area_id = self.area_id);
    END Delete_Suburb;

 END;
 /
```

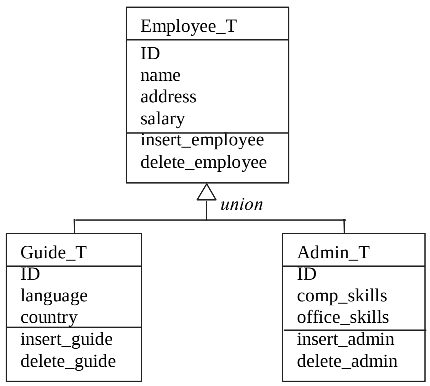
Summary

One type of dynamic aspect implemented in an object-relational system is the generic method. Generic methods are basically simple methods that are needed for operations such as retrieval, update, deletion, and insertion. For these methods, the concept of referential integrity is crucial and thus they need to be considered and designed accurately before implementation. In addition, different types of object structures or relationships (inheritance, association, and aggregation) will result in different types of generic-method implementations as well.

Chapter Problems

1. Giant Travel is a well-known travel agency that operates guided tours. With offices around the world, they maintain accurate and detailed

employee data. The employee data are kept in an object Employee_T and can be divided into two child objects: Guide_T and Admin_T. An employee can be categorized as a guide or an administration staff, but he or she can also be both. This is important because in the peak season, an administration worker might be needed to guide the tours and vice versa. The objects and the attributes are shown below.

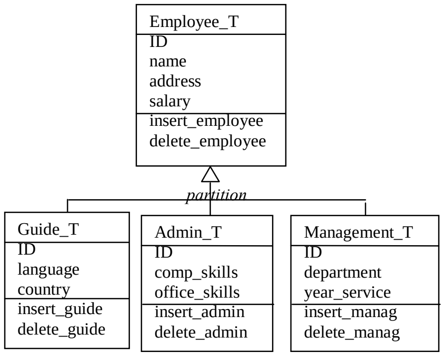


Assume that the tables for each object have been created; write the implementation of insertion into and deletion from tables Employee and Guide.

2. Continuing the case of Giant Travel in Question 1, management now wants employees' roles to become more specialized based on their major potentials. Since one employee can be only a guide or an administration staff, for that purpose, another attribute emp_type must be added to the Employee_T object. However, there are some records in the Employee table that are not categorized into the Guide or Admin object, that is, the managers.

   Assume that the tables for each object have been created; write the implementation of insertion into and deletion from tables Employee and Admin.

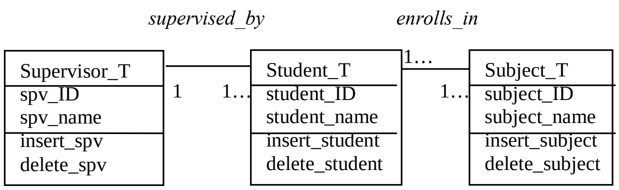
3. Continuing the case of Giant Travel in the previous questions, management now wants to create another object under the Employee_T object named Management_T, which obviously contains all the data of the managers. All employees must be categorized in one, and only one, child type.



Assume that the tables for each object have been created; write the implementation of insertion into and deletion from tables Employee and Management. Note that we want to delete the managers' records from the finance department.

4. The following figure shows the relationship among objects Supervisor_T, Student_T, and Subject_T in a university. A student can take many subjects, and a subject can be taken by many students. For every subject a student takes, there is a mark given.

   In another relationship, a student can be supervised by only one supervisor, but a supervisor can supervise many students. Assume that objects have been created and the tables from these objects are shown.



Supervisor	
Spv_ID	**Spv_Name**
1001	Steve Donaldson
1003	Erin Goldsmith
1007	Tony Wibowo

Student	
Student_ID	**Student_Name**
11013876	Robert Tan
11014832	Julio Fernandez
11014990	Colin Brown

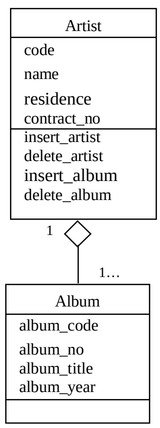
Subject	
Subject_ID	**Subject_Name**
CSE31DB	Database System
CSE31UIE	User Interface Engineering
CSE42ADB	Advanced Database

Enrolls_In		
Student_ID	**Subject_Code**	**Mark**
11013876	CSE31DB	86
11013876	CSE31UIE	90
11014832	CSE31ADB	78
11014990	CSE31DB	74
11014990	CSE31UIE	70

a. Write generic methods to insert into and delete from table Enrolls_In.

b. Write generic member methods to insert into and delete from table Supervisor.

5. Village Records' database keeps its artists as objects. Every artist has recorded and released at least one album. This album is kept as a nesting table inside the artist object.



Assume that the tables are created already.

a. Write the member procedures to insert into and delete from the nested table Album.

b. Write the stored procedure that takes one parameter, the artist's name, and shows the album name and years that he or she has recorded.

6. If the implementation for the aggregation relationship of Village Records in the previous question is done by using the clustering technique, and assuming that the cluster, tables, and index have been created, complete the following.

a. Write the stored procedures to insert into and delete from the whole table Artist.

b. Write the stored procedures to insert into and delete from the part table Album.

Chapter Solutions

1.
```
CREATE OR REPLACE TYPE BODY Employee_T AS

   MEMBER PROCEDURE Insert_Employee(
      new_id IN VARCHAR2,
      new_name IN VARCHAR2,
      new_address IN VARCHAR2,
      new_salary IN NUMBER) IS

   BEGIN
   INSERT INTO Employee
      VALUES (new_id, new_name, new_address, new_salary);
   END Insert_Employee;

   MEMBER PROCEDURE Delete_Employee IS

   BEGIN
      DELETE FROM Employee
      WHERE Employee.id = self.id;
   END Delete_Employee;

END;
```

```
/

CREATE OR REPLACE TYPE BODY Guide_T AS

   MEMBER PROCEDURE Insert_Guide(
      new_id IN VARCHAR2,
      new_name IN VARCHAR2,
      new_address IN VARCHAR2,
      new_salary IN VARCHAR2,
      new_language IN VARCHAR2,
      new_country IN VARCHAR2) IS

   BEGIN
   INSERT INTO Guide
      VALUES (new_id, new_name, new_address, new_salary,
               new_language, new_country);
   END Insert_Guide;

   MEMBER PROCEDURE Delete_Guide IS

   BEGIN
   DELETE FROM Guide
      WHERE Guide.id = self.id;
   END Delete_Guide;

END;
/
```

2.

```
CREATE OR REPLACE TYPE BODY Employee_T AS

   MEMBER PROCEDURE Insert_Employee(
      new_id IN VARCHAR2,
      new_name IN VARCHAR2,
      new_address IN VARCHAR2,
      new_salary IN NUMBER) IS

   BEGIN
   INSERT INTO Employee
      VALUES (new_id, new_name, new_address, new_salary,
      NULL);
   END Insert_Employee;

   MEMBER PROCEDURE Delete_Employee IS



   BEGIN
```

```
      DELETE FROM Employee
         WHERE Employee.ID = self.id
         AND Employee.employee_type IS NULL;
      END Delete_Employee;

   END;
   /

   CREATE OR REPLACE TYPE BODY Admin_T AS

      MEMBER PROCEDURE Insert_Admin(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_salary IN VARCHAR2,
         new_comp_skills IN VARCHAR2,
         new_office_skills IN VARCHAR2) IS

      BEGIN
      INSERT INTO Employee
         VALUES (new_id, new_name, new_address, new_salary,
         'Admin');
         INSERT INTO Admin
         VALUES (new_id, new_comp_skills, new_office_skills);
      END Insert_Admin;

      MEMBER PROCEDURE Delete_Admin IS

      BEGIN
      DELETE FROM Employee
         WHERE Employee.id = self.id
         AND Employee.employee_type = 'Admin';
      END Delete_Admin;

      END;
      /
```

3.
```
CREATE OR REPLACE TYPE BODY Management_T AS

      MEMBER PROCEDURE Insert_Manag(
         new_id IN VARCHAR2,
         new_name IN VARCHAR2,
         new_address IN VARCHAR2,
         new_salary IN VARCHAR2,
         new_department IN VARCHAR2,
         new_year_service IN VARCHAR2) IS
```

```
    BEGIN
       INSERT INTO Management
       VALUES (new_id, new_name, new_address, new_salary,
                new_department, new_year_service);
    END Insert_Manag;

    MEMBER PROCEDURE Delete_Manag IS

    BEGIN
    DELETE FROM Management
       WHERE Management.id = self.id;
    END Delete_Manag;

END;
/
```

4. a.
```
CREATE OR REPLACE PROCEDURE Insert_Enrolls_In(

   new_student_id IN VARCHAR2,
   new_subject_id IN VARCHAR2) AS

   student_temp REF Student_T;
   subject_temp REF Subject_T;

   BEGIN
      SELECT REF(a) INTO student_temp
      FROM Student a
      WHERE a.student_id = new_student_id;

      SELECT REF(b) INTO subject_temp
      FROM Subject b
      WHERE b.subject_id = new_subject_id;

      INSERT INTO Enrolls_In
      VALUES (student_temp, subject_temp);
   END Insert_Enrolls_In;
/

CREATE OR REPLACE PROCEDURE Delete_Enrolls_In(
   deleted_student_id IN VARCHAR2,
   deleted_subject_id IN VARCHAR2) AS

   BEGIN
      DELETE FROM Enrolls_In
      WHERE
         Enrolls_In.student IN
```

```
            (SELECT REF(a)
             FROM Student a
             WHERE a.student_id = deleted_student_id) AND
            Enrolls_In.subject IN
            (SELECT REF(b)
             FROM Subject b
             WHERE b.subject_id = deleted_subject_id);
      END Delete_Enrolls_In;
   /
```

b.
```
   CREATE OR REPLACE TYPE BODY Supervisor_T AS

      MEMBER PROCEDURE insert_spv(
         new_spv_id IN VARCHAR2,
         new_spv_name IN NUMBER) IS

      BEGIN
         INSERT INTO Supervisor
         VALUES (new_spv_id, new_spv_name);
      END insert_spv;

      MEMBER PROCEDURE delete_spv IS

      BEGIN
         — Supervised_by is the ref of Supervisor in the
         Student_T object.
         DELETE FROM Student b
         WHERE b.supervised_by.spv_id = self.spv_id;
         DELETE FROM Supervisor a
         WHERE a.spv_id = self.spv_id;
      END delete_spv;

   END;
   /
```

5. a.
```
   CREATE OR REPLACE TYPE BODY Artist_T AS

      MEMBER PROCEDURE Insert_Album(
         new_code IN VARCHAR2,
         new_album_code IN VARCHAR2,
         new_album_no IN NUMBER,
         new_album_title IN VARCHAR2,
         new_album_year IN NUMBER),

      BEGIN

      INSERT INTO THE
```

```
           (SELECT a.album
            FROM Artist a
            WHERE a.code = new_code)
        VALUES (new_album_code, new_album_no,
               new_album_title, new_album_year);
    END Insert_Album;

    — This procedure deletes all albums of the artist.

    MEMBER PROCEDURE Delete_Album IS

    BEGIN
        DELETE FROM THE
           (SELECT a.Album
            FROM Artist a
            WHERE a.code = self.code);
    END Delete_Album;

END;
/
```

b.
```
CREATE OR REPLACE PROCEDURE Show_Album(
    new_artist_name IN VACRHAR2) AS

CURSOR c_album IS
    SELECT album_title, album_year
FROM THE
      (SELECT album
       FROM Artist
       WHERE artist_name = new_artist_name);

BEGIN
    FOR v_curs IN c_album LOOP
        DBMS_OUTPUT.PUT_LINE
           (v_curs.album_title||' '||v_curs.album_year);
    END LOOP;
END Show_Album;
/
```

6. a.
```
CREATE OR REPLACE PROCEDURE Insert_Album(

    new_code IN VARCHAR2,
    new_album_code IN VARCHAR2,
    new_album_no IN NUMBER,
    new_album_title IN VARCHAR2,
    new_album_year IN VARCHAR2) AS
```

```
   BEGIN
      INSERT INTO Album
      VALUES (new_code, new_album_code, new_album_no,
             new_album_title, new_album_year);
   END Insert_Album;
/

CREATE OR REPLACE PROCEDURE Delete_Album(
   deleted_album_code IN NUMBER) AS

   BEGIN
      DELETE FROM Album
      WHERE album_code = deleted_album_code;
   END Delete_Album;
/
```

b.

```
CREATE OR REPLACE PROCEDURE Insert_Artist(
   new_code IN VARCHAR2,
   new_name IN VARCHAR2,
   new_residence IN VARCHAR2,
   new_contract_no IN VARCHAR2) AS


BEGIN
      INSERT INTO Artist
      VALUES  (new_code,  new_name,  new_residence,
   new_contract_no);
   END Insert_Artist;
/

CREATE OR REPLACE PROCEDURE Delete_Artist(
   deleted_code IN VARCHAR2) AS


BEGIN
      DELETE FROM Artist
      WHERE code = deleted_code;
   END Delete_Artist;
/
```

Chapter VI

User-Defined Queries

This chapter describes object-based user-defined queries in Oracle™. The queries will vary based on the hierarchy of the object model. We will show different categories of queries along the object-oriented relationships of inheritance, association, and aggregation.

These queries can be performed as ad hoc queries or implemented as methods. User-defined methods are methods whereby users define algorithms or the processes to be carried out by the methods. Since these methods involve operations specified by the users, they are called user-defined methods. As an example, we will use the case study of the authorship of the course manual in Chapter III as a working example for this chapter. Some queries discussed here are based on the DDL specified in Figure 3.36.

User-Defined Queries in Inheritance Hierarchies

In this section, different queries along inheritance hierarchies will be described. User-defined queries along inheritance hierarchies can be divided into two categories: *subclass queries* and *superclass queries*. Note that because there are two ways of implementing inheritance, using the shared ID between the primary key and foreign key or using the "under" keyword, we will show user-defined queries for both techniques in the following sections.

Subclass Query

User-defined queries in an inheritance hierarchy are queries that involve attributes of the class where the methods reside and attributes of their superclasses. Since a number of classes (at least two) are involved, a join operation to link all of these classes becomes necessary. The general format for the representation of user-defined queries in an inheritance hierarchy is as follows.

In the From clause, a list of tables is produced. These tables include all intermediate tables between a subclass ($table_1$) and a super-class ($table_n$). The *inheritance join expression* can be a join predicate to join all tables listed if the shared ID technique is used. Alternatively, if the latest Oracle™ is used, then it can be a *treat* expression to cast the selection from one class type to another within the inheritance hierarchy.

A subclass query is a query that retrieves information from the subclass(es), where the selection predicates are originated at the superclass. Figure 6.2 shows the flow of a query in a subclass query.

The query representation for a subclass query is shown in Figure 6.3, while Figure 6.4 shows the example of a subclass query and the results.

The subclass-query representation (see Figure 6.3) is applicable if we implement the superclass and subclass as two different tables. If we use the under features provided by Oracle™ 9 and above, we can use the treat keyword in the query. The general syntax of such a type of query is as follows.

Figure 6.1. User-defined inheritance query representation

```
SELECT <function or expression>
FROM <table1, table2, …, tablek, …, tablen>
WHERE <inheritance join expression>
AND <tablek.OID = &input_OID>
```

Figure 6.2. Subclass query flow

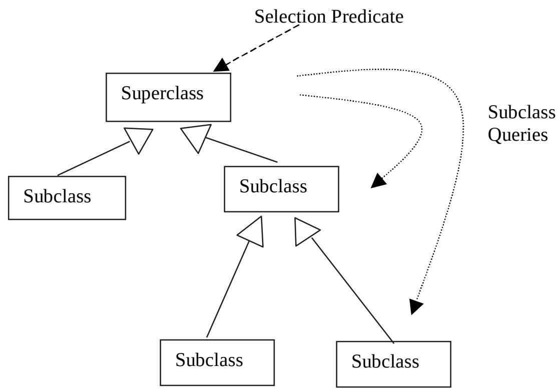


Figure 6.3. Subclass-query representation (using shared ID)

```
SELECT <sub-class attributes>
FROM <table1, table2,………,tablen>
WHERE <join predicates>
AND <tablen.attr = &input_super-class_selection_predicates>

where:   Table1, …, table n-1 are subclass tables, and
      tablen is a superclass table.
```

Superclass Query

A superclass query retrieves information from the superclass(es), where the selection predicates are originated at a subclass (see Figure 6.7).

The query representation for a superclass query using the shared ID is shown in Figure 6.8, while Figure 6.9 shows the example of a superclass query.

It is important to note that in inheritance queries, join operations have to be performed to link the superclass to the subclasses. When the hierarchy is deep, a number of join operations may be needed to perform a query. The fact that all of the join operations are carried out on primary keys of the tables makes the operations inexpensive in terms of performance cost.

Figure 6.4. Subclass-query example (using shared ID)

```
Example 1:
Find the contact number(s) of an author whose name is David Taniar.

  SELECT contact_no
  FROM Author a, Teaching_Staff b
  WHERE a.ao_id = b.ao_id
  AND a.name = 'David Taniar;

  The above query shows that we only need to join based on the common
shared ID, which is ao_ID. Since the contact number is a varray, the above
query will show the following result:
        CONTACT_NO
  ----------------------------
  CONTACTS (99059693, 94793060)
```

Figure 6.5. Query representation (using treat)

```
SELECT TREAT(VALUE(<alias>) AS <sub-type name>).<sub-class attribute>
FROM <table name>
WHERE <table.attr = &input_super-class_selection_predicates>;
```

Figure 6.6. Subclass-query example (using treat)

```
Example 2:
Find the institution of an author whose name is David Taniar.

   SELECT TREAT(VALUE(a) as Academic_T).i_name
   FROM Author a
   WHERE a.name = 'David Taniar';

  The above query shows the following result:
         i_name
  ----------------------------
  Monash University
```

With Oracle™ 9 and above, we can implement an inheritance relationship using one table only. The table will be of the supertype, in this case, Author. Thus, neither the subclass query nor the superclass query needs a join operation.

Another possibility of a superclass query in Oracle™ 9 and above is the use of "is of." This *type* of predicate tests object instances for the level of specializa-

Figure 6.7. Superclass-query flow

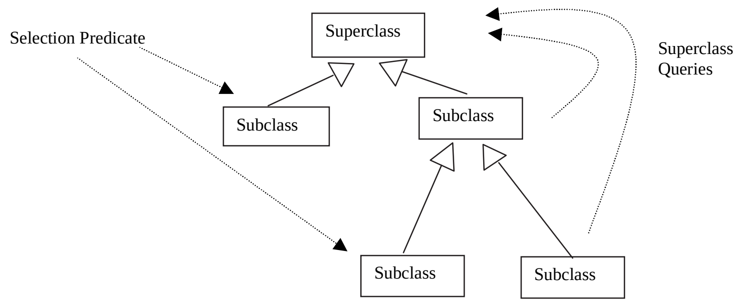


Figure 6.8. Superclass-query representation (using shared ID)

```
SELECT <super-class attributes>
FROM <table1, table2, …, tablen>
WHERE <join predicates>
AND <sub-class table.attr =
     &input_sub-class_selection_predicates>

where:    Table1, …, table n-1 are subclass tables, and
       tablen is a superclass table.
```

Figure 6.9. Superclass-query example

```
Example 3:
Find the details of the author(s) whose institution name is Monash
University.

  SELECT a.name, a.address
  FROM Author a, Academic b
  WHERE a.ao_id = b.ao_id
  AND b.i_name = 'Monash University';
```

Figure 6.10. Superclass-query example (using treat)

```
SELECT a.name, a.address
FROM Author a
WHERE TREAT(VALUE(a) as Academic).i_name = 'Monash University';
```

Figure 6.11. Superclass-query representation (using "is of")

```
SELECT <super-class attribute>
FROM <table name>
WHERE VALUE(<alias>) IS OF (Sub-class name);
```

Figure 6.12. Superclass-query example (using "is of")

```
Example 4:
Find the details of authors who belong to the industry-based type.

   SELECT a.name, a.address
   FROM Author a
   WHERE VALUE(a) IS OF (Industry_Based_T)
```

tion of its type along with any other further subclasses of the subclass. The general syntax of such a query is as follows.

User-Defined Queries in Association Relationships

In this section, different queries along association relationships will be described. They can be divided into two categories: *referencing queries* and *dereferencing queries*. Each of these types will be discussed in the following sections.

Referencing Query

A referencing query is a query from a class that holds the object reference (ref) to a class that is being referenced. The class that is being referenced is the class

Figure 6.13. Referencing-query flow

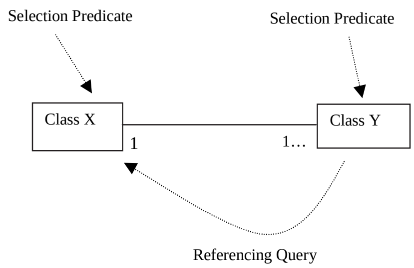


Figure 6.14. Referencing-query representation

```
SELECT <referencing class attributes>
FROM <referencing table>
WHERE <referencing table path expression>
[AND <class table.attr = &input_class_selection_predicates>]

where: The referencing table or class is the one that holds the many side in an
       association relationship.
```

Figure 6.15. Referencing-query Example 1

```
Example 5:
SELECT b.lecturer.total_hour
FROM Subject b
WHERE b.sub_name = 'Databases';
```

that holds the one side in a one-to-many relationship. Figure 6.13 depicts a referencing query.

The query representation for referencing a query is shown in Figure 6.14, while Figure 6.15 shows an example of this query type.

In the previous example, no join is performed. Rather, we are using object referencing from Teaching Staff to Subject through the lecturer attribute, which is of ref data type.

Example 6 (Figure 6.16) also shows a referencing type of query whereby a path traversal through the object references is used rather than the usual join operation. Without the facility of object references (ref) in Oracle™, we would

Figure 6.16. Referencing-query Example 2

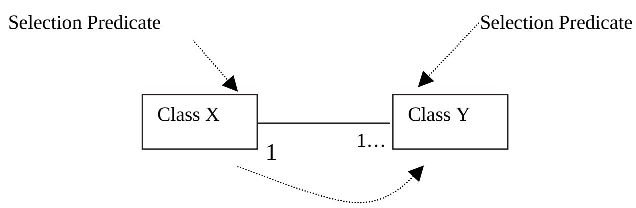
```
Example 6:
Display all subject details along with the teaching staff responsible for
the subject, showing only those subjects in which the teaching staff's
total contact hours is more than 5.

SELECT b.code, b.subname, b.venue, b.lecturer.name
FROM Subject b
WHERE b.lecturer.TotalHour > 5;
```

have to use a join operation between Subject and Teaching Staff to perform the above queries.

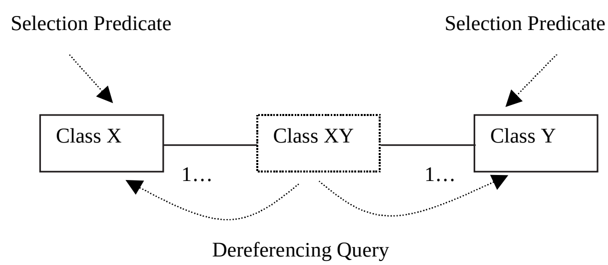
Dereferencing Query

A dereferencing query is a query from the referred class to a class that holds the object reference (ref). In a many-to-many relationship, both classes that are connected are the referred classes. Figure 6.17 below shows two types of dereferencing queries.

Figure 6.17. Dereferencing-query flow



a. Dereferencing query in a one-to-many relationship



b. Dereferencing query in a many-to-many relationship with a virtual class XY

Figure 6.18. Dereferencing-query representation

```
SELECT <class table attributes>
FROM <referring table>, <referred table>
WHERE <referencing join>
[AND <class table.attr = &input_class_selection_predicates>]

where:
       The referencing join takes the form of
          <referring class attribute = REF(referred class)>.
```

Figure 6.19. Dereferencing-query Example 1

```
Example 7:
In the relationship between Course Manual and Author, display all course
manuals written by John Smith.

  SELECT a.title
  FROM Course_Manual a, Author b, Publish c
  WHERE c.course_manual = REF(a)
  AND c.author = REF(b)
  AND b.name = 'John Smith';
```

The query representation for a dereferencing query is shown in Figure 6.18, while Figure 6.19 shows an example of this query type.

Similar to the previous dereferencing example, the above example also performs linking through object referencing rather than a join operation.

Note that in the previous example, both links are performed through object references. The Publish table holds two object references, one to Course_Manual_T and another one to Author_T. This situation is established in a many-to-many association relationship.

User-Defined Queries in Aggregation Hierarchies

In this section, we will describe different queries along aggregation hierarchies. These queries can be divided into two categories: *part queries* and *whole queries*. Each of the above types will be discussed in the following sections.

Figure 6.20. Dereferencing-query Example 2

```
Example 8:
In the relationship between Course Manual and Author, display all course
manuals along with the names of the author(s), showing only those authors
who live in Melbourne.

  SELECT a.title, b.name
  FROM Course_Manual a, Author b, Publish c
  WHERE c.course_manual = REF(a)
  AND c.author = REF(b)
  AND b.address LIKE '%Melbourne';
```

Figure 6.21. Dereferencing-query Example 3

```
Example 9:
In the relationship between Course Manual and Author, display all course
manuals along with the name(s) and address(es) of the author(s).

  SELECT a.title, b.name, b.address
  FROM Course_Manual a, Author b, Publish c
  WHERE c.course_manual = REF(a)
  AND c.author = REF(b);
```

Part Query

A part query is an aggregation-hierarchy query used to retrieve information of part classes, where the selection predicates are originated at the whole class. Figure 6.22 shows a part-query flow in a nesting technique.

The query representation for a part query is shown in Figure 6.23, while Figure 6.24 shows the example of a part query.

In the nesting technique, as mentioned in Chapter 5, we use the keyword "the" for querying the nested tables. Figure 6.24 shows that the selection predicate is located in the whole table Course_Manual.

Part queries can also appear in aggregation relationships implemented using the clustering technique. Figure 6.25 shows the query representation for a part query using the clustering technique, while Figure 6.26 shows an example of the query.

Note that when a clustering technique is used, the queries to access the data along the aggregation hierarchy are simply standard queries to join the whole table with its associated parts. However, the cluster index actually causes the queries to perform much better than those without it.

Figure 6.22. Part-query flow

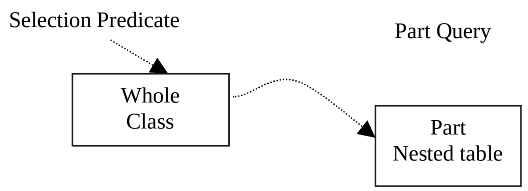


Figure 6.23. Part-query representation using the nesting technique

```
SELECT <"part" class attributes>
FROM THE ( SELECT "whole" class nested table attribute
           FROM <"whole" class table>
           WHERE <"whole" class table.attr =
                  &input_class_selection_predicates> )
```

Figure 6.24. Part-query example using the nesting technique

Example 10:
In the relationship between Course Manual and Chapter implemented using the nesting technique, display the chapter number and chapter title of a course-manual titled *Object-Relational Databases*.

```
SELECT c_no, c_title
FROM THE (SELECT a.chapter
          FROM Course_Manual a
          WHERE a.title = 'Object-Relational Databases');
```

Figure 6.25. Part-query representation using the clustering technique

```
SELECT <"part" class attributes>
FROM <table_1, table_2, …, table_n>
WHERE <join predicates>
AND <"whole" class table.attr = &input_class_selection_predicates>

where:   Table_1, …, table_n-1 are part-class tables,
       and table_n is a whole-class table.
```

Figure 6.26. Part-query example in the clustering technique

```
Example 11:
In the relationship between Course Manual and Chapter implemented using the
clustering technique, display the chapter number and chapter title of a
course manual titled Object-Relational Databases.

   SELECT a.c_no, a.c_title
   FROM Chapter a, Course_Manual b
   WHERE a.isbn = b.isbn
   AND b.title = 'Object-Relational Databases';
```

Whole Query

A whole query is the aggregation-hierarchy query to retrieve information from the whole class, where the selection predicates are originated at the part class. Figure 6.27 shows a whole-query flow in a nesting technique.

The technique we are using for solving a whole query in a nesting technique is called *unnesting*. It is because the nested table cannot be accessed except through the whole class, and yet we want to be able to access a selection predicate in the nested table (i.e., the part table). In this case, we need to unnest the nesting structure.

Figure 6.28 shows the query representation for a whole query using the nesting technique, while Figure 6.29 shows the example of the query.

Figure 6.29 shows how we can unnest a nested table structure in order to access its attributes directly. Figure 6.30 shows how we can run a query to retrieve the whole information within a nested structure.

Obviously, the above result is not very easy to interpret. In order to come up with a better display, we can also use the unnesting technique for the above

Figure 6.27. Whole-query flow

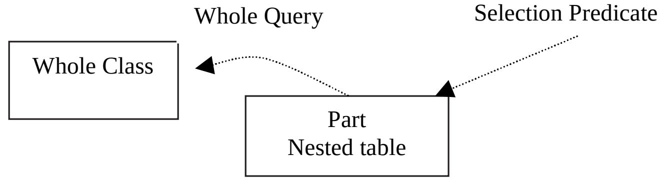


Figure 6.28. Whole-query representation using the nesting technique

```
SELECT <"whole" class attributes>
FROM < "whole" class table, TABLE("whole" class nested table
       attribute)>
WHERE <"part" class table.attr = &input_class_selection_predicates>
```

Figure 6.29. Whole-query example using the nesting technique

```
Example 12:
In the relationship between Course Manual and Chapter implemented using the
nesting technique, display the course-manual ISBN and course-manual title
that has an associated chapter-number 1 entitled "Introduction to Object-
Relational."

  SELECT a.isbn, a.title
  FROM Course_Manual a, TABLE(a.chapter) b
  WHERE b.c_no = 1
  AND b.c_title = 'Introduction to Object-Relational';
```

Figure 6.30. Query for whole information in the table using the nesting technique

```
Example 13:
In the relationship between Course Manual and Chapter implemented using the
nesting technique, display all course manuals together with their
associated chapters.

  SELECT *
  FROM Course_Manual;

     This example will give the following result:

  ISBN   TITLE            YEAR  CHAPTER(C_NO, C_TITLE, PAGE_NO)
  ----------------------------------------------------------------
  111xx  Databases   1993  CHAPTER_TABLE_T(CHAPTER_T(1, 'OODB', 1))
```

query. Nevertheless, the query will show a repetition of whole-table attributes if it has a number of parts (see Figure 6.31).

In the clustering technique, whole queries are implemented in a very similar manner as that of part queries. Figure 6.32 shows the query representation for a whole query using the nesting technique, while Figure 6.33 shows an example of the query.

Figure 6.31. Query for whole information using the nesting technique by unnesting

```
    SELECT a.isbn, a.title, a.year, b.c_no, b.c_title,
           b.page_no
    FROM Course_Manual a, TABLE(a.chapter) b;

The above query will give the following result:

ISBN        TITLE       YEAR        C_NO        C_TITLE          PAGE_NO
------------------------------------------------------------------------
111xx       Databases   1993        1           OODB             1
```

Figure 6.32. Whole-query representation using the clustering technique

```
SELECT <"whole" class attributes>
FROM <table1, table2, …, tablen>
WHERE <join predicates>
AND <"part" class table.attr = &input_class_selection_predicates>

where:    Table1 is a whole-class table,
       and table2, …, tablen are part-class tables.
```

Figure 6.33. Whole-query example using the clustering technique

```
Example 14:
In the relationship between Course Manual and Chapter implemented using the
clustering technique, display the course-manual ISBN and course-manual
title that has an associated chapter-number 1 entitled "Introduction to
Object-Relational."

SELECT a.isbn, a.title
FROM Course_Manual a, Chapter b
WHERE a.isbn = b.isbn
   AND b.c_no = 1
   AND b.c_title = 'Introduction to Object-Relational';
```

There is one limitation of the nesting-technique query that can be solved by using the clustering technique. With the nesting technique, during DML operation, the nested table locks the parent row. Thus, only one modification can be made to the particular nested table at a time. It shows that the part query in the nesting technique is not optimum compared with the clustering technique. Nevertheless, the whole query of the nesting technique can perform as good as in the clustering technique. It is shown in the following example.

Figure 6.34. Whole query from multiple part tables

```
Example 15:
Assume there is another nested table Preface under Course Manual. Display
the course-manual ISBN and course-manual title that has an associated
chapter-number 1 entitled "Introduction to Object-Relational" and a preface
entitled "Acknowledgement."

  SELECT a.isbn, a.title
  FROM Course_Manual a, TABLE(a.chapter) b, TABLE(a.preface) c
  WHERE b.c_no = 1
  AND b.c_title = 'Introduction to Object-Relational'
  AND c.p_title = 'Acknowledgement';
```

User-Defined Queries Using Multiple Collection Types

Oracle™ has also introduced collection types as alternative data types. They are other features of an object-oriented database that need to be adopted by RDBMSs. One of the types, which is the nested table, has been mentioned previously when we discussed aggregation relationships. In this section, we will discuss multiple collection types that can increase the power of ORDBMS application.

Varray Collection Type

One of the new collection types introduced by Oracle™ is an array type called varray. This type can be stored in database tables. When used as an attribute type in a table, varray is stored in line with the other attributes within the table.

Example 1 (see Section 6.1.1) demonstrates a subclass query that retrieves an array type of attribute. Retrieving the whole array can be done through SQL queries. The following example shows how we can retrieve information when the selection predicate is of the varray type.

It is not possible to access an individual element of an array type using an SQL query only. As shown above, we need to use a procedure whereby we can retrieve and manipulate the array elements. Furthermore, we have to make sure that during the insertion of the varray in the above example, there are three values to input. If there are only two contact numbers, the third value, null, should be inserted. It is needed to avoid error during the query process. The

Figure 6.35. Varray collection-type example

```
Example 16:
Find the details of authors whose contact numbers include 94793060.

   DECLARE

   CURSOR c_contact IS
      SELECT a.name, a.address, b.contact_no
      FROM Author a, Teaching_Staff b
      WHERE a.ao_id = b.ao_id;

   BEGIN

      FOR v_contactrec IN c_contact LOOP
         IF (v_contactrec.contact_no(1) = 94793060) OR
            (v_contactrec.contact_no(2) = 94793060) OR
            (v_contactrec.contact_no(3) = 94793060) THEN

            DBMS_OUTPUT.PUT_LINE('AuthorName:'||
            v_contactrec.name||'Author Address:'||
            v_contactrec.address);
         END IF;
      END LOOP;

   END;
   /
```

Figure 6.36. Example of a varray collection-type manipulation

```
 Example 17:
 Update one of the contact numbers of an author whose ao_ID is 123 from
 94793060 to 94793000.

 DECLARE
   Contacts      Teaching_Staff.contact_no%TYPE;

 BEGIN
   SELECT b.contact_no
   INTO contacts
   FROM Author a, Teaching_Staff b
   WHERE a.ao_id = b.ao_id
   AND a.ao_id = '123';

   FOR i IN 1..3 LOOP
      IF (contacts(i) = 94793060) THEN
          contacts(i) := 94793000;
      END IF;
      DBMS_OUTPUT.PUT_LINE ('New Contact Number '||i||
                         ':'||contacts(i));
   END LOOP;

 END;
/
```

following example shows how we can select a stored varray in a variable so that it can be manipulated.

Note that because the main purpose of this section is to demonstrate collection types, we will assume that each table that is needed for the examples has already been created. Whenever access to an inheritance hierarchy is used in the examples, we will assume the implementation method uses a shared ID. For example, in the defined cursor of Figure 6.35, we will need to use treat if we only implement one superclass table for the inheritance hierarchy (as shown in Figure 6.5).

Varray has several methods that can be used for accessing elements. Some of the methods are shown below.

First, Last	returns the index of the first (or last) element within the array
Next, Prior	returns the index of the next (or prior) element within an array, relative to a specified element
Exists	returns ”true” if the entry exists in the array
Count	returns the total number of elements within an array
Limit	returns the maximum number of elements of an array
Extend	adds elements to an array
Trim	removes elements from the end of an array
Delete	removes specified elements from an array

The following example shows how we can display the last element of an array using the "last" keyword for collection types. Note that the last element may not necessarily be the upper boundary of the varray. For example, we may define a varray of three elements, but since there are only two elements loaded in an array, the last element will be element number 2.

Nested-Table Collection Type

In Section 6.3, we have seen how we can manipulate a nested table using SQL queries as one of the methods for an aggregation relationship. Another way of manipulating a nested-table structure is by retrieving the whole nested table into

Figure 6.37. Varray collection-type method example

```
Example 18:
Find the last contact number of an author whose ao_ID is 123

DECLARE
   Contacts      Teaching_Staff.contact_no%TYPE;

BEGIN
   SELECT b.contact_no
   INTO contacts
   FROM Author a, Teaching_Staff b
   WHERE a.ao_id = b.ao_id
   AND a.ao_id = '123';

   DBMS_OUTPUT.PUT_LINE ('Last Contact No:'||
                        Contacts(contacts.LAST));

END;
/
```

a variable, and then manipulating the values within a procedure. The following example shows how we can manipulate a nested table.

Note that we use the method Last in the above example to check for the last record within the nested table. All the methods that are applicable for varray are applicable for the nested table except for Limit. This method will return null in a nested table because there is no explicit maximum size for a nested table.

Unlike varray that retains the ordering of its elements when stored, a nested table does not preserve its ordering in the database storage. This is because varray maintains its element in line within the main table, whereas a nested table is stored independently of the associated main table.

User-Defined Queries with Object References

So far we have seen how we can create association relationships with object references using REF. REF is not the only object references feature available. Oracle™ also provides other operators that will allow us to navigate object references. The operators include VALUE, DEREF, and IS DANGLING. We will consider each operator in the following section:

Figure 6.38. Example of a nested-table manipulation

```
Example 19:
Find the total number of chapters in a course manual published by an author
with ao_ID 123.

DECLARE
  v_chapters   Course_Manual.chapter%TYPE;

BEGIN

  SELECT a.chapter
  INTO v_chapters
  FROM Course_Manual a, Author b, Publish c
  WHERE c.course_manual = REF(a)
  AND c.author = REF(b)
  AND b.ao_id = '123';

  IF v_chapters IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE
    ('The number of chapters is:'||v_chapters.LAST);
  END IF;

END;
/
```

Figure 6.39. Value example

```
Example 20:
using value to compare the return value of a query

SELECT a.sub_name, a.venue
FROM Subject a, Teaching_Staff b
WHERE a.lecturer = REF(b)
AND VALUE(a) =
   (SELECT VALUE(c)
    FROM Subject c
    WHERE c.code = 'CSE42ADB');
```

VALUE

Value is used to retrieve the value of row objects. It is only applicable to object type, and thus it will be invalid to use for retrieving row tables. This operator might be useful to compare objects and find whether they have the same values.

Figure 6.40. Deref example

```
Example 21:
Retrieve the information about the teaching staff using a deref to Subject.

   SELECT DEREF(a.lecturer) FROM Subject a;

The above query returns the following results:

 DEREF(LECTURER)(AO_ID, TOTAL_HOUR, CONTACT_NO)
 --------------------------------------------------------------
 TEACHING_STAFF_T('p1', 20, CONTACTS(94675810, 93452341, NULL))
 TEACHING_STAFF_T('p5', 30, CONTACTS(92318406, 93510365, NULL))
 TEACHING_STAFF_T('p8', 35, CONTACTS(92638475, 92345678, NULL))
```

Figure 6.41. "Is dangling" example

```
Example 22:
Check whether or not there is any dangling reference from Subject to
Teaching Staff (notice that Subject has an attribute called lecturer, which
is of type ref).

   SELECT s.sub_name, s.venue
   FROM Subject s
   WHERE s.lecturer IS DANGLING;
```

Figure 6.42. Example of "is dangling"

```
Example 23:
Copy a subject into a new subject if the code is CSE42ADB and the venue is
ELT2.

     DECLARE
       S1        Subject_T;

     BEGIN
       SELECT VALUE(s) INTO S1
       FROM Subject s, Teaching_Staff t
       WHERE s.lecturer = REF(t);
       AND s.code = 'MAT42'
       AND s.venue = 'ELT2';
     END;
     /
```

DEREF

Deref is used to return the object of an object reference. Note that a Deref of a ref is the same as a value.

IS DANGLING

Whenever an object has an object reference (ref) pointing to it, this object is not supposed to be deleted. If it is deleted, the reference is said to be dangling or pointing to nothing. "Is dangling" is used to check whether or not a particular reference is pointing to an existing object.

Unfortunately, in the implementation of object references, there is no implicit referential integrity checking such as the one found in primary-key and foreign-key relationships. The ref operator does not automatically avoid any deletion of the referenced objects in the earlier version of Oracle™. However, new releases after Oracle™ 8 provide referential integrity checking with object references.

Figure 6.43. Object-table example

```
Example 24a:
Create an object table Author with all the attributes
as specified in Chapter 3 (Case Study).

   CREATE OR REPLACE TYPE Author_T AS OBJECT
      (ao_id      VARCHAR2(3),
       name       VARCHAR2(10),
       address    VARCHAR2(20))
   /

   CREATE TABLE Author OF Author_T
      (ao_id NOT NULL,
       PRIMARY KEY (ao_id));
```

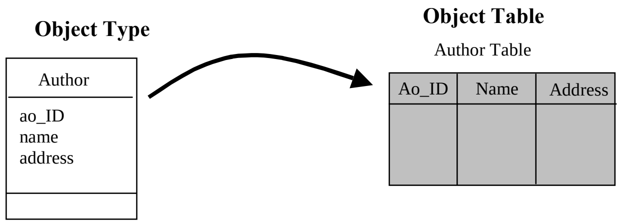


Figure 6.44. Object attribute

```
Example 24b:
Create an object attribute Author within the table Course_Manual.

  CREATE TABLE Course_Manual
     (isbn       VARCHAR2(10),
      title      VARCHAR2(20),
      year       NUMBER,
      course_author   AUTHOR_T);
```

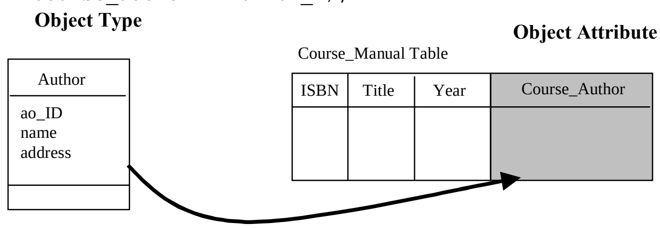


Figure 6.45. Object-attribute query example

```
Example 25:
Find all information about course-manual ISBN number 1268-9000.

   SELECT *
   FROM Course_Manual;

      The query will return the following result:

   ISBN          TITLE       YEAR        COURSE_AUTHOR(AOID, NAME, ADDRESS)
   ----------------------------------------------------------
   1268-9000     Parallel Database 1998  AUTHOR_T('123', 'D Taniar',
   'Clayton')
```

Object Table vs. Object Attribute

We have seen in most of our examples how to create and manipulate an object table. An object table (or often called a row object) is a database table created based on an object type. Thus, each row within the table actually represents the values of an object.

Another technique of making an object persistent in object-relational databases is by creating an object attribute (or often called a column object). An object attribute is actually an attribute of a relational table that is of object type.

Figure 6.46. Varray inside object-attribute example

```
Example 26:
Create a varray of object attributes Authors within the Course_Manual
table.

  CREATE OR REPLACE TYPE Authors AS VARRAY(3) OF Author_T
  /

  CREATE TABLE Course_Manual
    (isbn           VARCHAR2(10),
     title          VARCHAR2(20),
     year           NUMBER,
     course_author  Authors);

By running the following query to display the new contents of
Course_Manual, we will get results like it is shown in the following
display.

  SELECT *
  FROM Course_Manual;

 ISBN          TITLE                  YEAR        COURSE_AUTHOR(AOID,
 NAME, ADDRESS)
 ---------------------------------------------------------------------------
 ------------------------------------------------
 1268-9000     Parallel Database      1998        AUTHORS(AUTHOR_T('123',
 'D Taniar', 'Clayton'), AUTHOR_T('567', 'W Rahayu', 'Bundoora'))
```

The following examples 24a and 24b show the differences between an object table and object attribute.

Example 24b shows how we can have an attribute of an object type in our relational table. This notion of object attributes can also be used to link a table with an object, for example, to link Course_Manual and Author. However, when using object attributes, there is no table created for the object attribute. Therefore, we can only retrieve the author information through the Course_Manual table. If we need to be able to index and manipulate author information independently, then we need to create a separate table for Author_T and define a link between the tables.

Although we cannot manipulate the Author object attribute within the Course_Manual table, we still can display the value of the object attribute using a simple SQL query as shown below.

We can also have a collection of object attributes. In other words, we can have an attribute whose value is a collection of objects instead of just a single object. Example 26 shows how we can create a varray of Authors objects.

Figure 6.47. Index-organization table example

```
Example 27:
using index organization to implement an aggregation hierarchy between
Course_Manual_T and Chapter_T.

   CREATE TABLE Course_Manual
      (isbn       VARCHAR2(10) NOT NULL,
       title      VARCHAR2(20),
       year       NUMBER,
       PRIMARY KEY (isbn));

   CREATE TABLE Chapter
      (isbn       VARCHAR2(10) NOT NULL,
       c_no       VARCHAR2(10) NOT NULL,
       c_title    VARCHAR2(25),
       page_no    NUMBER,
       PRIMARY KEY (isbn, c_no),
       FOREIGN KEY (isbn) REFERENCES Course_Manual(isbn)) ORGANIZATION
      INDEX;
```

Clustering Technique vs. Index-Organization Table

We have introduced the use of clusters in the previous chapters, mainly in the context of the implementation of aggregation hierarchies. The clustering technique, as opposed to the nesting technique, is more of a physical mechanism in which the database engine will cluster together rows that are connected using the same cluster key.

While the clustering technique can be very useful in implementing aggregation hierarchies, Oracle™ actually supports another physical mechanism of clustering rows together called an *index-organization table*. It allows us to physically cluster and order a table based on its primary key. The main difference between clustering and index organization is that clustering allows multiple table clusters, whereas the index-organization table allows only a single table cluster.

This difference is the main reason why index-organization tables may not be suitable for the implementation of aggregation hierarchies. In most situations, aggregation hierarchies consist of many different parts connected to a whole object. However, if what we have is a homogenous aggregation, with one whole object and one part object, then the following index-organization structure can be used.

In the Chapter 3 case study, we have a homogenous aggregation between Course_Manual_T and Chapter_T. We will see here how we can also implement the aggregation hierarchy using index organization.

In example 27, each row of the Chapter table is physically stored together with the associated Course_Manual row as specified in the primary key of Chapter. This certainly increases performance in accessing the records of the tables whenever they need to be accessed together. However, in cases where we have homogenous aggregation with a possible future extension of the model, where we may extend the aggregation with one or more part objects, then the index-organization table may not be a suitable solution. When deciding which structure to use, we need to also carefully consider any possible future extension of the model. For example, in the above Course_Manual_T whole object, we may want to add the Preface_T object and Bibliography_T object as part objects. The aggregation hierarchy is no longer a homogenous aggregation.

Case Study

Recall the AEU case study in Chapters 1 and 5. The union now wants to add some user-defined methods for several queries that are often made. These queries will be implemented as member methods of the classes. The user-defined queries that will be implemented are listed as follows.

- Query to show the price, date of purchase, and the brand of a vehicle. It is a superclass query and will be implemented as a member method in the subclass Vehicle_T.
- Query to show the details of a property building. It is a subclass query and will be implemented as a member method in the superclass Property_T.
- Query to find the organizer's name and her or his address for a particular teacher. It is a referencing query and will be implemented as a member method in the class that holds the object reference, Teacher_T.
- Query to find the details of the union where a particular employee works. It is a referencing query and will be implemented as a member method in the class that holds the object reference, Employee_T.

Figure 6.48. AEU case study with user-defined method implementation

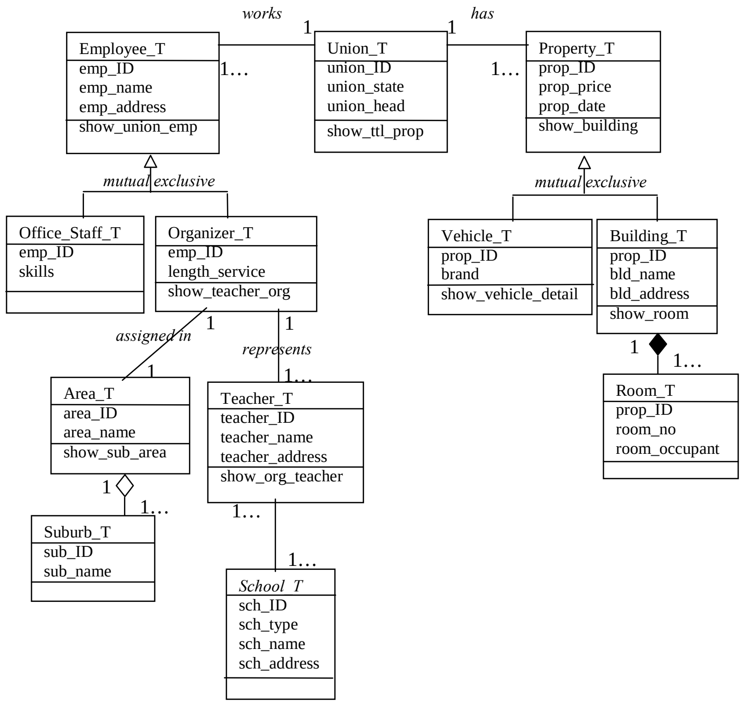


Figure 6.49. Implementation of the case study in Oracle™

```
Methods Declaration
   CREATE OR REPLACE TYPE Union_T AS OBJECT
      (union_id        VARCHAR2(10),
       union_state     VARCHAR2(20),
       union_head      VARCHAR2(30),

       MEMBER PROCEDURE show_ttl_prop)
   /

   CREATE TABLE Union_Table OF Union_T
      (union_id NOT NULL,
       PRIMARY KEY (union_id));

   CREATE OR REPLACE TYPE Employee_T AS OBJECT
      (emp_id          VARCHAR2(10),
       emp_name        VARCHAR2(30),
       emp_address     VARCHAR2(30),
       emp_type        VARCHAR2(15),
       work_in REF Union_T,

       MEMBER PROCEDURE show_union_emp) NOT FINAL
   /
```

Figure 6.49. (continued)

```
CREATE TABLE Employee OF Employee_T
   (emp_id NOT NULL,
    emp_type CHECK (emp_type IN
      ('Office Staff', 'Organizer', NULL)),
    PRIMARY KEY (emp_id));

CREATE OR REPLACE TYPE Office_Staff_T UNDER Employee_T
   (skills    VARCHAR2(50))
/

CREATE OR REPLACE TYPE Organizer_T UNDER Employee_T
   (length_service        VARCHAR2(10),

    MEMBER PROCEDURE show_teacher_org)
/

CREATE OR REPLACE TYPE Teacher_T AS OBJECT
   (teacher_id            VARCHAR2(10),
    teacher_name          VARCHAR2(20),
    teacher_address       VARCHAR2(10),
    representation REF Organizer_T,

    MEMBER PROCEDURE show_org_teacher)
/

CREATE TABLE Teacher OF Teacher_T
   (teacher_id NOT NULL,
    PRIMARY KEY (teacher_id));

CREATE OR REPLACE TYPE Schools_T AS OBJECT
   (sch_id          VARCHAR2(10),
    sch_name        VARCHAR2(20),
    sch_address     VARCHAR2(30),
    sch_type        VARCHAR2(15))
/

CREATE TABLE Schools OF Schools_T
   (sch_id NOT NULL,
    sch_type CHECK (sch_type IN ('Primary', 'Secondary', 'TAFE')),
    PRIMARY KEY (sch_id));

CREATE TABLE Teach_In
   (teacher REF Teacher_T,
    school REF Schools_T);

CREATE OR REPLACE TYPE Suburb_T AS OBJECT
   (sub_id    VARCHAR2(10),
    sub_name  VARCHAR2(20))
/

CREATE OR REPLACE TYPE Suburb_Table_T AS TABLE OF Suburb_T
/
```

Figure 6.49. (continued)

```
CREATE OR REPLACE TYPE Area_T AS OBJECT
   (area_id   VARCHAR2(10),
    area_name VARCHAR2(20),
    suburb    Suburb_Table_T,
    assigned_org REF Organizer_T,

    MEMBER PROCEDURE show_sub_area)
/

CREATE TABLE Area OF Area_T
   (area_id NOT NULL,
    PRIMARY KEY (area_id))
   NESTED TABLE suburb STORE AS suburb_tab;

CREATE OR REPLACE TYPE Property_T AS OBJECT
   (prop_id         VARCHAR2(10),
    prop_price      NUMBER,
    prop_date       DATE,
    prop_type       VARCHAR2(15),
    in_union REF Union_T,

    MEMBER PROCEDURE show_building)
/

CREATE TABLE Property OF Property_T
   (prop_id NOT NULL,
    prop_type CHECK (prop_type IN ('Vehicle', 'Building', NULL)),
    PRIMARY KEY (prop_id));

CREATE OR REPLACE TYPE Vehicle_T AS OBJECT
   (prop_id   VARCHAR2(10),
    brand     VARCHAR2(20),

    MEMBER PROCEDURE show_vehicle_detail)
/

CREATE TABLE Vehicle OF Vehicle_T
   (prop_id NOT NULL,
    PRIMARY KEY (prop_id),
    FOREIGN KEY (prop_id) REFERENCES Property(prop_id)
      ON DELETE CASCADE);

CREATE OR REPLACE TYPE Buildings_T AS OBJECT
   (prop_id         VARCHAR2(10),
    bld_name        VARCHAR2(20),
    bld_address     VARCHAR2(30),

    MEMBER PROCEDURE show_room)
/
```

Figure 6.49. (continued)

```
    CREATE CLUSTER Buildings_Cluster
      (prop_id         VARCHAR2(10));

    CREATE TABLE Buildings OF Buildings_T
      (prop_id NOT NULL,
       PRIMARY KEY (prop_id),
       FOREIGN KEY (prop_id) REFERENCES Property(prop_id))
      CLUSTER Buildings_Cluster(prop_id);

    CREATE OR REPLACE TYPE Room_T AS OBJECT
      (prop_id         VARCHAR2(10),
       room_no         VARCHAR2(10),
       room_occupant   VARCHAR2(30))
    /

    CREATE TABLE Room OF Room_T
      (prop_id NOT NULL,
       room_no NOT NULL,
       PRIMARY KEY (prop_id, room_no),
       FOREIGN KEY (prop_id) REFERENCES Property(prop_id))
      CLUSTER Buildings_Cluster(prop_id);

    CREATE INDEX Buildings_Cluster_Index
      ON CLUSTER Buildings_Cluster;

  Methods Implementation
    CREATE OR REPLACE TYPE BODY Union_T AS

      MEMBER PROCEDURE show_ttl_prop IS

        v_total  NUMBER;

      BEGIN
        SELECT SUM(b.prop_price) INTO v_total
        FROM Union_Table a, Property b
        WHERE b.in_union = REF(a)
        AND a.union_id = self.union_id;
      END show_ttl_prop;

    END;
    /
    CREATE OR REPLACE TYPE BODY Employee_T AS

      MEMBER PROCEDURE show_union_emp IS

        v_state  VARCHAR2(20);

      BEGIN
```

Figure 6.49. (continued)

```
      SELECT a.union_state INTO v_state
      FROM Union_Table a, Employee b
      WHERE b.work_in = REF(a)
      AND b.emp_id = self.emp_id;
    END show_union_emp;

  END;
  /

  CREATE OR REPLACE TYPE BODY Organizer_T AS

    MEMBER PROCEDURE show_teacher_org IS

    CURSOR c_show_teacher_org IS
      SELECT TREAT (VALUE(a) AS Organizer).emp_name, d.sch_name
      FROM Employee a, Teacher b, Teach_In c, Schools d
      WHERE b.representation = REF(a)
      AND c.teacher = REF(b)
      AND c.school = REF(d)
      AND a.emp_id = self.emp_id;

    BEGIN
      FOR v_show_teacher_org IN c_show_teacher_org LOOP
        DBMS_OUTPUT.PUT_LINE
           (v_show_teacher_org.emp_name||' '||
            v_show_teacher_org.sch_name);
      END LOOP;
    END show_teacher_org;

  END;
  /

  CREATE OR REPLACE TYPE BODY Teacher_T AS

    MEMBER PROCEDURE show_org_teacher IS

    CURSOR c_show_org_teacher IS
      SELECT TREAT (VALUE(a) AS Organizer).emp_name, TREAT (VALU
      Organizer).emp_address
      FROM Employee a, Teacher b
      WHERE b.representation = REF(a)
      AND b.teacher_id = self.emp_id;

    BEGIN
      FOR v_show_org_teacher IN c_show_org_teacher LOOP
        DBMS_OUTPUT.PUT_LINE
           (v_show_org_teacher.emp_name||' '||
            v_show_org_teacher.emp_address);
      END LOOP;
    END show_org_teacher;
```

Figure 6.49. (continued)

```
END;
/

CREATE OR REPLACE TYPE BODY Area_T AS

  MEMBER PROCEDURE show_sub_area IS

  CURSOR c_show_sub_area IS
    SELECT b.sub_id, b.sub_name
    FROM Area a, TABLE(a.suburb) b
    WHERE a.area_name = self.area_name

  BEGIN
    FOR v_show_sub_area IN c_show_sub_area LOOP
      DBMS_OUTPUT.PUT_LINE
        (v_show_sub_area.sub_id||' '||
         v_show_sub_area.sub_name);
    END LOOP;
  END show_sub_area;

END;
/

CREATE OR REPLACE TYPE BODY Property_T AS

  MEMBER PROCEDURE show_building IS

  CURSOR c_show_building IS
    SELECT b.bld_name, b.bld_address
    FROM Property a, Buildings b
    AND a.prop_id = b.prop_id;

  BEGIN
    FOR v_show_building IN c_show_building LOOP
      DBMS_OUTPUT.PUT_LINE
        (v_show_building.bld_name||' '||
         v_show_building.bld_address);
    END LOOP;
  END show_building;

END;
/

CREATE OR REPLACE TYPE BODY Vehicle_T AS

  MEMBER PROCEDURE show_price_date IS

  CURSOR c_show_price_date IS
    SELECT a.prop_price, a.prop_date
    FROM Property a, Vehicle b
```

Figure 6.49. (continued)

```
      WHERE a.prop_id = b.prop_id
      AND a.prop_id = self.prop_id;

    BEGIN
      FOR v_show_price_date IN c_show_price_date LOOP
        DBMS_OUPUT.PUT_LINE
           (v_show_price_date.prop_price||‘ ‘||
            v_show_price_date.prop_date||‘ ‘||
            v_show_price_date.brand);
      END LOOP;
    END show_vehicle_detail;

  END;
  /

  CREATE OR REPLACE TYPE BODY Buildings_T AS

    MEMBER PROCEDURE show_room IS

    CURSOR c_show_room IS
      SELECT room_no, room_occupant
      FROM Room
      WHERE prop_id = self.bld_id;

    BEGIN
      FOR v_show_room IN c_show_room LOOP
        DBMS_OUTPUT.PUT_LINE
           (v_show_room.room_no||‘ ‘||
            v_show_room.room_occupant);
      END LOOP;
    END show_room;

  END;
  /
```

- Query to show the name of the teachers that are represented by an organizer. It will also need to show the school where those teachers are working. It is a dereferencing, subclass query and will be implemented as a member method in the class that is referenced, Organizer_T.
- Query to show the total property value of a particular state union. It is a dereferencing query and will be implemented as a member method in the class that is referred, Union_T.
- Query to show all the suburb names for a particular area. It is a part query that will be implemented as a member method in the whole class Area_T.

- Query to show the details of an organizer who is in charge in a particular suburb. It is a whole query combined with a dereferencing query and superclass query at the same time. It will be implemented as a member method in the part class Suburb_T.
- Query to show the room number and its occupant given a building ID as the parameter. It is a part query that will be implemented as a member method in superclass Building_T.

Figure 6.48 shows the AEU database diagram with the attributes and methods. For simplicity, we ignore the generic methods implemented in Chapter 5.

For the implementation section, we will re-create the class and tables so that we can see the user-defined methods declarations. In this case, the declarations will not include the generic member methods as shown in the Chapter 5 case study. Figure 6.49 shows the whole implementation for the user-defined methods in this case study.

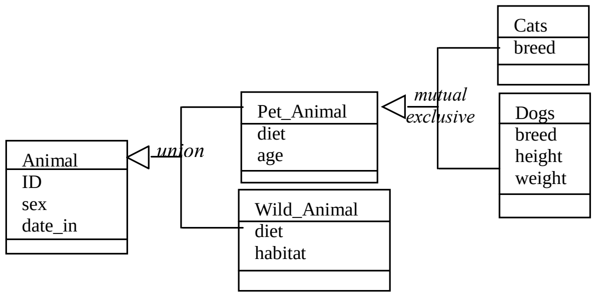
Summary

Another type of dynamic aspect in ORDBMSs is user-defined methods. While generic methods are used for the simple operations of retrieval, updating, deletion, and insertion, user-defined methods are used for performing defined algorithms specified by the users. For this method, issues to be considered include the structure of the relationships, the data types, and also the referencing methods implemented inside the classes.

Chapter Problems

1. The animal pound (AP) has always maintained records of every animal they have had. They keep the records in a hierarchical relationship. Some examples of the data kept in the tables are shown below.
   a. Create a superclass query to retrieve the date_in of all big dogs (height is more than 35 cm).

b. Create a sub- and superclass query to retrieve the age of the small dogs (height is less than 25 cm or weight is less than 10 kg).



Animal		
ID	**Sex**	**Date_In**
a243	M	7/12/01
a244	M	9/12/01
a245	F	3/1/02
a246	F	3/1/02
a247	F	3/1/02
a248	M	10/1/02

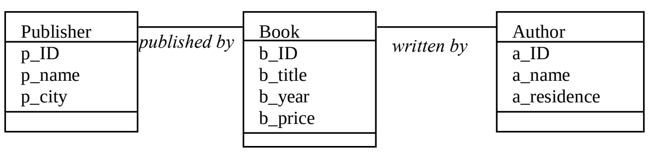
Pet Animal						
ID	**Diet**	Age	Pet_Type	Breed	Height	Weight
a243	meat	1	dogs	Labrador retriever	40	10
a244	meat	5	dogs	Pugs	30	8
a246	meat	7	dogs	German shepherd	60	25
a247	meat	2	dogs	Fox terrier	20	8
a248	grain	1	null			

2. *Ryan Bookstore* keeps a record of their books in three different object tables: Author, Book, and Publisher. The object diagram and sample of the records are shown below.

a. Create a referencing query to retrieve the name and the city of the publisher that publishes *Les Miserables*.

b. Create a dereferencing query to retrieve the title and the publishing year of the books published by Harper Collins, New York.

c. Create a dereferencing query to retrieve the titles, authors, and prices of the books that were published after 1985.



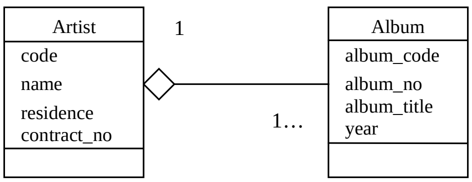
Publisher		
P_ID	**P_Name**	**P_City**
H01	Harper Collins	NYC
K02	Knopf	London
L02	Little, Brown, & Co	Boston
P02	Penguin Classic	London
P04	Penguin Australia	Sydney
S02	Simon and Schuster	NYC

Author		
A_ID	**A_Name**	**A_Residence**
A23	Allende, Isabel	Spain
B35	Bronte, Charlotte	UK
B36	Bronte, Emily	UK
C28	Courtenay, Bryce	Australia
H09	Hugo, Victor	France
K04	King, Stephen	USA

Book				
B_ID	**B_Title**	**B_Year**	**B_Price**	**Publish**
F123	*The Complete Story*	1980	16	P02
F342	*The Potato Factory*	1998	18	P04
F345	*Dreamcatcher*	2000	15	S02
F453	*Les Miserables*	1980	12	P02
F488	*The House of the Spirits*	1985	12	K02
F499	*Daughter of Fortune*	1999	19	H01
F560	*Solomon's Songs*	1999	19	P04

Written_By	
Book	**Author**
F123	B35
F123	B36
F342	C28
F345	K04
F453	H09
F488	A23
F499	A23
F560	C28

3. Village Records, as mentioned in the sample questions for Chapter 5, uses the following object diagram to keep their artist and album records, and they use the nesting technique for the implementation. Some of the records are shown below.
   a. Create a part query to retrieve the album number, title, and year of the artist Bryan King.
   b. Create a whole query to retrieve the names and the contract numbers of the artists who have recorded more than two albums.



Artist					
Code	**Name**	**Residence**	**Contract_No**	**Album**	
BK	Bryan King	Tamworth	13576345		→ 1
PA	Paige Alexander	Sydney	14534321		→ 2
R	Rogue	Melbourne	12093722		→ 3
TB	Tim Ball	Melbourne	12092834		→ 4
VQ	Valerie Quinton	Melbourne	12098546		→ 5

	Artist			
	Album_Code	Album_No	Album_Title	Year
1 →	BK1	1	Bryan King	1999
2 →	PA1	1	Paige Alexander	1999
	PA2	2	Paige	2001
3 →	R1	1	Rogue	2001
4 →	TB1	1	Tim Ball Vol 1	1997
	TB2	2	Tim Ball Vol 2	1999
	TB3	3	Tim Ball Vol 3	2002
5 →	VQ1	1	Valerie	2002

4. Village Records wants to expand their stock by selling videos of the artists. Therefore, they want to use a clustering technique instead of a nesting table. Below are the new tables.

Artist			
Code	**Name**	**Residence**	**Contract_No**
BK	Bryan King	Tamworth	13576345
PA	Paige Alexander	Sydney	14534321
R	Rogue	Melbourne	12093722
TB	Tim Ball	Melbourne	12092834
VQ	Valerie Quinton	Melbourne	12098546

Album				
Artist_Code	**Album_Code**	**Album_No**	**Album_Title**	**Year**
BK	BK1	1	Bryan King	1999
PA	PA1	1	Paige Alexander	1999
PA	PA2	2	Paige	2001
R	R1	1	Rogue	2001
TB	TB1	1	Tim Ball Vol 1	1997
TB	TB2	2	Tim Ball Vol 2	1999
VQ	VQ1	1	Valerie	2002

Video			
Artist_Code	**Video_Code**	**Video_No**	**Video_Title**
BK	VBK1	1	Bryan King in Concert
PA	VPA1	1	Paige
R	VR1	1	Rogue in Rod Laver Arena
R	VR2	2	Rogue World
TB	VTB1	1	Sydney Concert Tim Ball
TB	VTB2	2	Tim Acoustic

a. Create a part query to retrieve the album title and video of the artist with contract number 12093722.

b. Create a whole query to retrieve the details of the artists for whom there are both an album and a video in stock

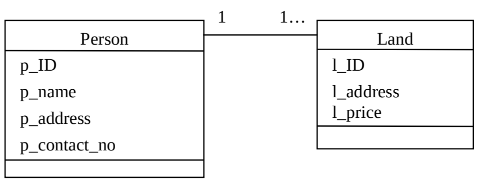
5. A real-estate agency keeps records of its tenants, which include the tenant_number, tenant_name, tenant_address, tenant_co_number, and ref_list. Ref_list is a type of varray of two references of the tenants.

a. Create the objects and tables of Tenant and Reference.

b. Create a procedure to show the details of the tenants who have at least one reference from a landlord.

6. Following Question 5 above, the real-estate company wants to extend its ref_list attribute of the tenant object. Ref_list is a varray of a reference object. The reference object has the attributes of reference_number, reference_name, relationship, reference_contact_no, and reference_date. Create the new object and table for the tenants and references.

7. Show the difference between an object table and object attribute by implementing the relation between two objects, Person and Land. A person can own more than one piece of land, but one piece of land can be owned by only one person. The details of these two objects are shown below.



Chapter Solutions

1. a.
```
SELECT a.id, a.date_in
FROM Animal a
WHERE TREAT(VALUE(a) AS dog_t).height > 35;
```
b.
```
SELECT p.id, p.age
FROM Pet_Animal p
WHERE TREAT(VALUE(p) AS dog_t).height < 25 OR
TREAT(VALUE(p) AS dog_t).weight < 10);
```

2. a.
```
SELECT b.pub_by.p_name, b.pub_by.p_city
FROM Book b
WHERE b.b_title = 'Les Miserables';
```
b.
```
SELECT b.b_title, b.b_year
FROM Book b
WHERE b.pub_by.p_id = 'P04';
```
c.
```
SELECT b.b_title, a.a_name, b.b_price
FROM Author a, Book b, Written_By w
```

```
WHERE w.author = REF(a)
AND w.book = REF(b)
AND b.b_year > 1985;
```

3. a.
```
SELECT album_no, album_title, year
FROM THE (SELECT album
          FROM Artist
          WHERE name = 'Bryan King');
```
   b.
```
SELECT DISTINCT a.name, a.contract_no
FROM Artist a, TABLE (a.album) b
WHERE b.album_no > 2;
```

4. a.
```
SELECT b.album_title, c.video_title
FROM Artist a, Album b, Video c
WHERE a.code = b.artist_code
AND a.code = c.artist_code
AND a.contract_no = 12093722;
```
   b.
```
SELECT *
FROM Artist
WHERE code IN (SELECT artist_code
               FROM Album)
AND code IN (SELECT artist_code
             FROM Video);
```

5. a.
```
CREATE OR REPLACE TYPE References AS VARRAY(2) OF
VARCHAR2(20)
/

CREATE OR REPLACE TYPE Tenants_T AS OBJECT
   (tenant_number        VARCHAR2(3),
    tenant_name          VARCHAR2(20),
    tenant_address       VARCHAR2(30),
    tenant_contact_no    NUMBER,
    ref_list             References)
/

CREATE TABLE Tenants OF Tenants_T
   (tenant_number NOT NULL,
    PRIMARY KEY (tenant_number));
```
   b.
```
DECLARE

CURSOR c_tenants IS
   SELECT tenant_number, tenant_name, tenant_address,
          tenant_contact_no, ref_list
   FROM Tenants;

BEGIN
```

```
    FOR v_tenants IN c_tenants LOOP
      IF (v_tenants.ref_list(1) = 'Landlord')   OR
      (v_tenants.ref_list(2) = 'Landlord') THEN
        DBMS_OUTPUT.PUT_LINE
            (v_tenants.tenant_number||''||
             v_tenants.tenant_name||''||
             v_tenants.tenant_address||''||
             v_tenants.tenant_contact_no);
      END IF;
    END LOOP;
  END;
  /
```

6.
```
CREATE OR REPLACE TYPE Reference_T AS OBJECT
    (reference_number           VARCHAR2(3),
     reference_name             VARCHAR2(20),
     relationship               VARCHAR2(20),
     reference_contact_no NUMBER,
     reference_date             DATE)
/

CREATE OR REPLACE TYPE References AS VARRAY(2) OF
Reference_T
/

CREATE OR REPLACE TYPE Tenants_T AS OBJECT
    (tenant_number        VARCHAR2(3),
     tenant_name          VARCHAR2(20),
     tenant_address       VARCHAR2(30),
     tenant_contact_no    NUMBER,
     ref_list             References)
/

CREATE TABLE Tenants OF Tenants_T
    (tenant_number  NOT NULL,
     PRIMARY KEY (tenant_number));
```

7. Object Table: Two tables are created from objects Person_T and Land_T. Therefore, we have to create the object first, followed by the tables. Notice that we use ref in connecting the two objects.

```
CREATE OR REPLACE TYPE Person_T AS OBJECT
    (p_id            VARCHAR2(3),
     p_name          VARCHAR2(10),
     p_address              VARCHAR2(20),
     p_contact_no    NUMBER)
/
```

```
CREATE OR REPLACE TYPE Land_T AS OBJECT
   (l_id           VARCHAR2(3),
    l_address           VARCHAR2(20),
    l_price        NUMBER,
    owner REF Person_T)
/

CREATE TABLE Person OF Person_T
   (p_id NOT NULL,
    PRIMARY KEY (p_id));

CREATE TABLE Land OF Land_T
   (l_id NOT NULL,
    PRIMARY KEY (l_id));
```

Object Attribute: We create only one table. In this case, as there is only one person who owns each piece of land, we create an object attribute of Person_T inside the Land table. Notice we are not using ref in connecting the object.

```
CREATE OR REPLACE TYPE Person_T AS OBJECT
   (p_id           VARCHAR2(3),
    p_name         VARCHAR2(10),
    p_address           VARCHAR2(20),
    p_contact_no   NUMBER)
/

CREATE OR REPLACE TYPE Land_T AS OBJECT
   (l_id           VARCHAR2(3),
    l_address           VARCHAR2(20),
    l_price        NUMBER,
    owner          Person_T)
/

CREATE TABLE Land OF Land_T
   (l_id NOT NULL,
    PRIMARY KEY (l_id));
```

Chapter VII

University Case Study

Our intention in the previous chapters was to give some understanding of the ORDB concept and its implementation using Oracle™. Examples, case studies, and questions based on these chapters have been relatively simplified in order to explain one concept at a time. However, in the real world, often we find far more complex cases that may involve the integration of every concept that we have already discussed. In this chapter, we will consider a bigger case study that uses most of the ORDB concepts.

In addition, we will also demonstrate the implementation of a big case study into one application that can be more user friendly. For this purpose, we will use a package that is also provided by Oracle™.

Problem Description

City University (CU) keeps an extensive database for daily operational purposes. The database includes information pertaining to the campuses, faculties, buildings, personnel, degrees, and subjects offered, and other data derived from them. Information Technology Services (ITS), responsible for maintaining the database system within the university, decided to use an ORDB and Oracle™ for the database implementation.

CU has eight campuses around the state of Victoria. The Campus database is linked to the Building and Person databases. Although each campus offers different degree courses and has different faculties, at this stage, there is no direct link from these data to the Campus table. Figure 7.1 shows the sample data for this table.

CU has five faculties, each of which is an aggregation of a different department, school, and research centre. Each of them is implemented as a separate object and has derived object tables. As we do not need to access the data of the departments, schools, and research centres directly for this database system, the data is implemented using a nested table. Figure 7.2 shows the sample for the Faculty table and its nested tables. Note that the attributes school_prof and dept_prof are themselves objects. Thus, they have their own attributes including name, contact, and year of inauguration. An attribute unit in the Research_Centre nested table will have more than one value and thus needs to be implemented using collection types.

Each campus has several buildings, each of which is an aggregation of different rooms such as offices, classrooms, and labs. The faculty can occupy many buildings. However, one building can only be allocated to one faculty. Note that there is an attribute bld_location, which is the location of the building on the particular campus map.

As mentioned previously, a building can be divided into offices, classrooms, and labs, each with its own attributes. Figure 7.4 shows the sample for the Office, Classroom, and Lab tables. Note that the attribute lab_equipment in Labs has to be implemented using collection types. For this aggregation, we are using the clustering technique instead of a nested table because there will be association relationships needed between the part table Office and another table to show the staff who occupies the office.

Figure 7.1. Campus table

Campus				
Campus_Location	**Campus_Address**	**Campus_Phone**	**Campus_Fax**	**Campus_Head**
Albury/Wodonga	Parkers Road Wodonga VIC 3690	61260583700	620260583777	John Hill
City	215 Franklin St. Melb VIC 3000	61392855100	6103 92855111	Michael A. O'Leary
Mildura	Benetook Ave. Mildura VIC 3502	61350223757	61350223646	Ron Broadhead

Figure 7.2. Faculty table and the nested tables

Faculty					
Fac_ID	**Fac_Name**	**Fac_Dean**	**Department**	**School**	**Research_Centre**
1	Health Sciences	S. Duckett	**Nested Tables**		
2	Humanity & Social Sc.	J. A. Salmond			
3	Law & Management	G. C. O'Brien			
4	Science, Tech. & Eng.	D. Finlay			
5	Regional Department	L. Kilmartin			

School (Nested Table)			
School_ID	**School_Name**	**School_Head**	**School_Prof**
1-1	Human Biosciences	Chris Handley	Chris Handley
1-2	Human Comm. Sciences	Elizabeth Lavender	Sheena Reilly, Alison Perry, Jan Branson

Department (Nested Table)			
Dept_ID	**Dept_Name**	**Dept_Head**	**Dept_Prof**
4-1	Agricultural Sciences	Mark Sandeman	
4-2	Biochemistry	Nick Hoogenraad	Nick Hoogenraad, Robin Anders, Claude Bernard, Bruce Stone

Research_Centre (Nested Table)			
RC_ID	**RC_Name**	**RC_Head**	**RC_Unit**
1-1	Australian Research Centre in Sex, Health & Society	Marian Pitts	SSAY Projects HIV Futures Australian Study of Health and Relationships
1-2	Australian Institute for Primary Care	Hal Swerissen	Centre for Dev. and Innovation in Health Centre for Quality in Health & Community Svc. Lincoln Gerontology Centre

Figure 7.3. Building table

Building					
Bld_ID	**Bld_Name**	**Bld_Location**	**Bld_Level**	**Campus_Location**	**Fac_ID**
BB1	Beth Gleeson	D5	4	Bundoora	4
BB2	Martin Building	F5	4	Bundoora	3
BB3	Thomas Cherry	D4	4	Bundoora	1
BB4	Physical Science 1	D5	3	Bundoora	4

Figure 7.4. Office, Classroom, and Lab tables

Office		
Bld_ID	**Off_No**	**Off_Phone**
BB4	BG207	94791118
BB4	BS208	94792393

Classroom		
Bld_ID	**Class_No**	**Class_Capacity**
BB3	TCLT	50
BB3	TC01	30

Lab			
Bld_ID	**Lab_No**	**Lab_Capacity**	**Lab_Equipment**
BB1	BG113	25	25 PC, 1 Printer
BB1	BG114	20	21 PC

Figure 7.5. Degree table

Degree				
Deg_ID	**Deg_Name**	**Deg_Length**	**Deg_Prereq**	**Fac_ID**
D100	Bachelor of Comp. Sci	3	Year 12 or equivalent	4
D101	Master of Comp. Sci	2	Bach of Comp. Sci	4

Figure 7.6. Person table

Person							
Pers_ID	**Pers_ Surname**	**Pers_ Fname**	**Pers_ Title**	**Pers_Address**	**Pers_Phone**	**Pers_ Postcode**	**Campus_ Location**
01234234	Grant	Felix	Mr	2 Boadle Rd Bundoora VIC	0398548753	3083	Bundoora
10008895	Xin	Harry	Mr	6 Kelley St Kew VIC	0398875542	3088	Bundoora
10002935	Jones	Felicity	Ms	14 Rennie St Thornbury VIC	0398722001	3071	Bundoora

Every faculty offers students a number of degrees. The information about the degree is stored in the Degree table (see Figure 7.5). Obviously, one particular degree can be offered by only one faculty.

One substantial part of the database is the personnel data. The university personnel can be categorized into two major types: staff and student. A staff can be categorized in more detail into administrator, technician, lecturer, and tutor. A lecturer can further be categorized into senior lecturer and associate lecturer. A tutor, on the other hand, can also be a student and, thus, has to be implemented in a multiple inheritance relationship.

While Figure 7.6 shows the Person table, Figure 7.7 shows the tables for its subclasses. Empty fields show that the attribute can be null.

Figure 7.7. Person's subclass tables

Staff			
Pers_ID	**Bld_ID**	**Off_No**	**Staff_Type**
10008895	BB1	BG212	Lecturer
10002935	BB4	BG210	Admin

Student	
Pers_ID	**Year**
01234234	2000
01958652	2000

Admin			
Pers_ID	**Admin_Title**	**Comp_Skills**	**Office_Skills**
10002935	Office Manager		Managerial
10008957	Receptionist	MS Office	Customer Service, Phone

Technician		
Pers_ID	**Tech_Title**	**Tech_Skills**
10005825	Network Officer	UNIX, NT
10015826	Photocopy Technician	Electrician

Lecturer		
Pers_ID	**Area**	**Lect_Type**
10008895	Software Engineering	Associate
10000255	Business Information	Senior

Senior Lecturer			
Pers_ID	**No_Phd**	**No_Master**	**No_Honours**
10000255	2	5	7
10000258		1	5

Associate Lecturer		
Pers_ID	**No_Honours**	**Year_Join**
10008895	2	1999
10006935		2001

Tutor		
Pers_ID	**No_Hours**	**Rate**
01234234	10	20.00
01958652	30	35.00

Figure 7.8. Subject table

Subject				
Subj_ID	**Subj_Name**	**Subj_Credit**	**Subj_Prereq**	**Pers_ID**
CSE21NET	Networking	10	CSE11IS	10008895
CSE42ADB	Advanced Database	15	CSE21DB	10006935

Figure 7.9. Enrolls_In and Takes tables

Enrolls_In	
Student	**Degree**
01234234	D101
10012568	D101

Takes		
Student	**Subject**	**Marks**
01234234	CSE42ADB	70
10012568	CSE42ADB	80

The Student_T class is linked to the Degree_T class. One student can take more than one degree at a time. The Student_T class is also linked to another class, Subject_T. It contains the information about the subject ID, subject name, subject credit, subject prerequisite, and its description. On the other hand, the Subject_T class is linked to the Lecturer_T class, which obviously shows the lecturer in charge of the subject. Figure 7.8 shows the Subject table

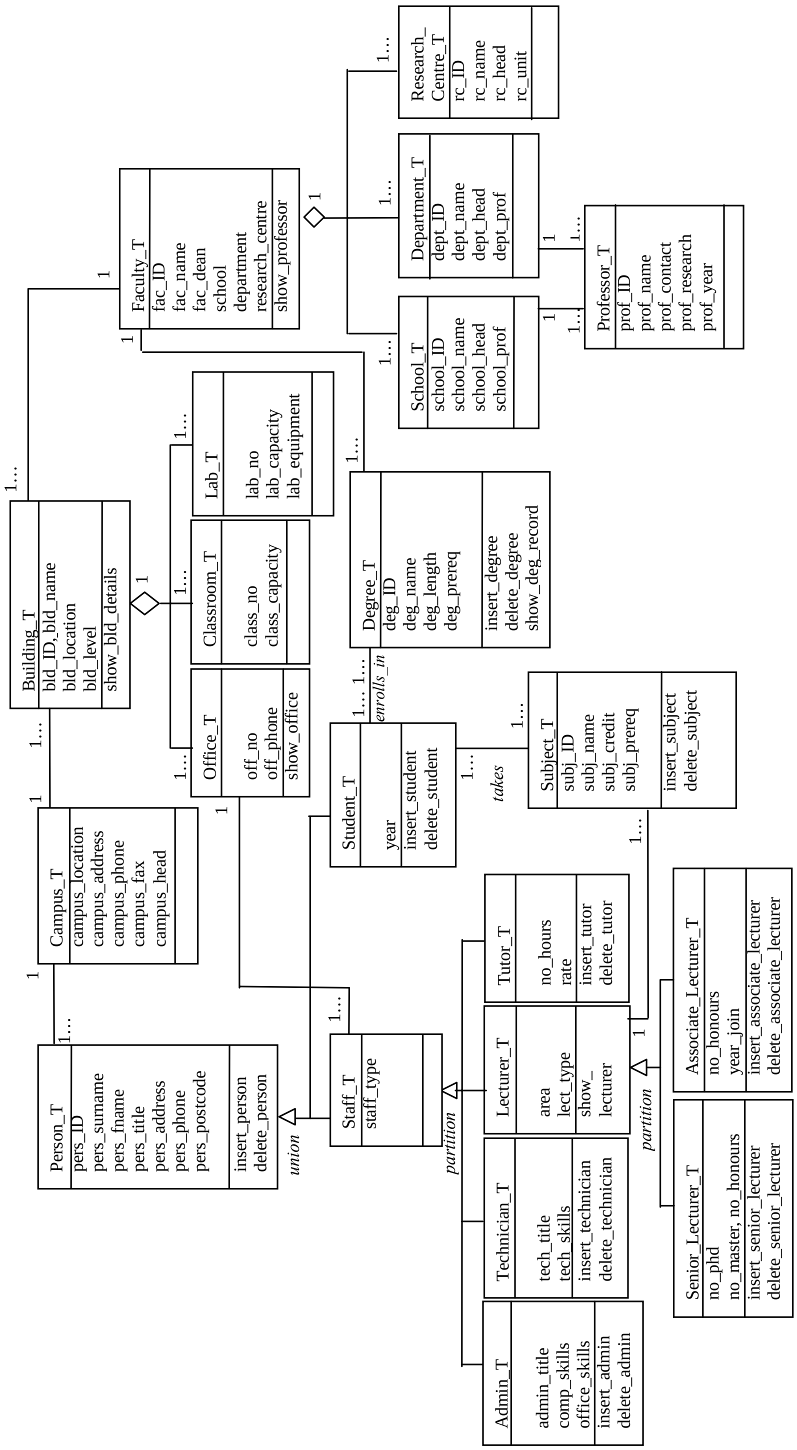
Figure 7.9 shows the tables associated with the Student table: respectively, the Enrolls_In table that is formed by the association to the Degree table, and the Takes table that is formed by the association to the Subject table. Note that the tables do not exactly store only the ID, for example, student_ID in the Enrolls_In table. The whole object with the particular ID is being referenced because of the implementation of object references.

ITS implements the generic methods inside the classes, which will need a lot of updates. They include Subject_T, Degree_T, and all the classes derived from Person_T. There are also generic stored procedures for insertion and deletion into tables that are not derived from objects, that is, table Enrolls_In and table Takes.

Beside the generic methods, there are some user-defined queries that are frequently made for this database. These user-defined queries will be implemented as user-defined methods, listed below.

- Method to show the names and the heads of the schools, departments, and research centres of a faculty. This method is implemented in Faculty_T.
- Method to insert the data of a building into a new table, namely, Building_Details. This method will be implemented in Building_T.
- Method to display the details of the offices and their occupants. This method will be implemented in the Office_T class.
- Method to save into a new table, namely, Degree_Records, which will store the degree details and the number of students enrolled in it. This method will be implemented in the Degree_T class.
- Method to show the details of the lecturer that will be implemented in the Lecturer_T class

Figure 7.10. Object-oriented diagram of CU



Problem Solution

The first thing to do in solving this problem is to design the database. We provide the design in an object-oriented diagram (see Figure 7.10). Note that the diagram does not indicate the number of tables that we need to create. We have to also consider the cardinality of the relationships before determining the number of tables. The diagram shows two aggregation relationships. We use the clustering technique for the Building_T-class aggregation because there is an association relationship needed to the part class, in this case, the Office_T class to the Lecturer_T class. On the other side, we will use the nested technique for the Faculty_T class.

To ensure a clearer step-by-step development, the solution will be implemented for one class at a time. It starts with the object creation, then progresses to the table creation and then, where applicable, the method creation. Note that the table for the many-to-many relationship will be implemented along with the implementation of the second class.

Campus_T Table

The implementation of the Campus_T class and the table derived from the class is shown below. There are no generic methods needed for this class because insertion or deletion of a campus database is not a frequent operation.

```
Relational Schemas
   Faculty (campus_location, campus_address,
   campus_phone,
            campus_fax, campus_head)

Class and Table Declaration
   CREATE OR REPLACE TYPE Campus_T AS OBJECT
      (campus_location     VARCHAR2(20),
       campus_address      VARCHAR2(50),
       campus_phone        VARCHAR2(12),
       campus_fax          VARCHAR2(12),
       campus_head         VARCHAR2(20))
   /
```

```
CREATE TABLE Campus OF Campus_T
   (campus_location NOT NULL,
    PRIMARY KEY (campus_location));
```

Faculty_T Class and Part Classes

The Faculty table contains three nested tables, and thus the classes for each of them have to be created first. The attributes school_prof, dept_prof, and rc_unit are collection types and will be implemented using varray. In addition, the first two are varrays of Professor_T. Therefore, we have to create this object first before creating the object of the collection types.

It is the same with the Campus_T class; we do not use generic methods frequently for these classes, so there will be no generic member methods implemented. However, as it is required, a user-defined method is implemented to show the names and the heads of the schools, departments, and research centres, given the faculty ID.

```
Relational Schemas
   Faculty (fac_ID, fac_name, fac_dean, school,
   department,
          research_centre)
   School (school_ID, school_name, school_head,
                school_prof)
   Dept (dept_ID, dept_name, dept_head, dept_prof)
   Research_Centre (rc_ID, rc_name, rc_head, rc_unit)

   Class, Table, and Method Declaration

   CREATE OR REPLACE TYPE Professor_T AS OBJECT
      (prof_id      VARCHAR2(10),
       prof_name    VARCHAR2(20),
       prof_contact VARCHAR2(12),
       prof_year    NUMBER)
   /

   CREATE OR REPLACE TYPE Professors AS VARRAY(5) OF
      Professor_T
   /

   CREATE OR REPLACE TYPE Units AS VARRAY(5) OF
   VARCHAR2(50)
   /
```

```
CREATE OR REPLACE TYPE School_T AS OBJECT
   (school_id     VARCHAR2(12),
    school_name   VARCHAR2(20),
    school_head   VARCHAR2(20),
    school_prof   Professors)
/

CREATE OR REPLACE TYPE School_Table_T AS TABLE OF
   School_T
/

CREATE OR REPLACE TYPE Department_T AS OBJECT
   (dept_id       VARCHAR2(12),
    dept_name     VARCHAR2(20),
    dept_head     VARCHAR2(20),
    dept_prof     Professors)
/

CREATE OR REPLACE TYPE Department_Table_T AS TABLE OF
   Department_T
/

CREATE OR REPLACE TYPE Research_Centre_T AS OBJECT
   (rc_id   VARCHAR2(12),
    rc_name VARCHAR2(20),
    rc_head VARCHAR2(20),
    rc_unit Units)
/

CREATE OR REPLACE TYPE Research_Centre_Table_T AS
   TABLE OF Research_Centre_T
/

CREATE OR REPLACE TYPE Faculty_T AS OBJECT
   (fac_id              VARCHAR2(10),
    fac_name            VARCHAR2(20),
    fac_dean            VARCHAR2(20),
    school              School_Table_T,
    department          Department_Table_T,
    research_centre     Research_Centre_Table_T,

    MEMBER PROCEDURE show_parts)
/

CREATE TABLE Faculty OF Faculty_T
   (fac_id NOT NULL,
```

```
  PRIMARY KEY (fac_id))
 NESTED TABLE school STORE AS school_tab
 NESTED TABLE department STORE AS dept_tab
 NESTED TABLE research_centre STORE AS rc_tab;
```

Methods Implementation

```
CREATE OR REPLACE TYPE BODY Faculty_T AS

   — We need three different cursors for the
   different nested tables.

   MEMBER PROCEDURE show_parts IS

   CURSOR c_school IS
      SELECT school_name, school_head
      FROM THE
         (SELECT school FROM Faculty
          WHERE fac_id = self.fac_id);

   CURSOR c_dept IS
      SELECT dept_name, dept_head
      FROM THE
         (SELECT department FROM Faculty
          WHERE fac_id = self.fac_id);

   CURSOR c_rc IS
      SELECT rc_name, rc_head
      FROM THE
         (SELECT research_centre FROM Faculty
          WHERE fac_id self.fac_id);

   BEGIN
      DBMS_OUTPUT.PUT_LINE
         ('Part Name'||'         '||'Head Name');
      DBMS_OUTPUT.PUT_LINE
         ('——————————————————');
      FOR v_school IN c_school LOOP
         DBMS_OUTPUT.PUT_LINE
            (v_school.school_name||'
            '||v_school.school_head);
      END LOOP;

      FOR v_dept IN c_dept LOOP
         DBMS_OUTPUT.PUT_LINE
```

```
            (v_dept.dept_name||'
            '||v_dept.dept_head);
        END LOOP;

        FOR v_rc IN c_rc LOOP
          DBMS_OUTPUT.PUT_LINE
            (v_rc.rc_name||'     '||v_rc.rc_head);
        END LOOP;
    END show_parts;

END;
/
```

Building_T Class and Part Classes

For the Building_T class and the part classes, we use the clustering technique, so in each part table, there is a whole-class primary key included. Again, there is no generic member method required in this class. Nevertheless, we still need a user-defined method to save into the new table, namely, Building_Details. We need to create this table first before being able to implement the member-method body.

```
Relational Schemas
    — Note that the first primary key in each part class
    is also a foreign
    — key to the whole class. The relationship from
    Building_T to other
    — classes is made using object references in in_campus
    and
    — for_faculty respectively for the Campus_T class and
    Faculty_T class.

    Buildings (bld_ID, bld_name, bld_location,
    bld_level,
              in_campus, for_faculty)
    Office (bld_ID, off_no, off_phone)
    Classroom (bld_ID, class_no, class_capacity)
    Lab (bld_ID, lab_no, lab_capacity, lab_equipment)

Class, Table, and Method Declaration
    — Equipments is a collection type of array to store
    — the attribute lab_equipment of Lab_T.
```

```
CREATE OR REPLACE TYPE Equipments AS VARRAY(3) OF
   VARCHAR2(20)
/

CREATE OR REPLACE TYPE Building_T AS OBJECT
   (bld_id         VARCHAR2(10),
    bld_name       VARCHAR2(20),
    bld_location   VARCHAR2(10),
    bld_level      NUMBER,
    in_campus REF Campus_T,
    for_faculty REF Faculty_T,

    MEMBER PROCEDURE show_bld_details)
/

CREATE CLUSTER Building_Cluster
   (bld_id         VARCHAR2(10));

CREATE TABLE Building OF Building_T
   (bld_id NOT NULL,
    PRIMARY KEY (bld_id))
   CLUSTER Building_Cluster(bld_id);

CREATE OR REPLACE TYPE Office_T AS OBJECT
   (bld_id         VARCHAR2(10),
    off_no         VARCHAR2(10),
    off_phone      VARCHAR2(12),

    MEMBER PROCEDURE show_office)
/

CREATE TABLE Office OF Office_T
   (bld_id NOT NULL,
    off_no NOT NULL,
    PRIMARY KEY (bld_id, off_no),
    FOREIGN KEY (bld_id) REFERENCES
   Building(bld_id))
   CLUSTER Building_Cluster(bld_id);

CREATE OR REPLACE TYPE Classroom_T AS OBJECT
   (bld_id              VARCHAR2(10),
    class_no            VARCHAR2(10),
    class_capacity      NUMBER)
/
```

```
CREATE TABLE Classroom OF Classroom_T
   (bld_id NOT NULL,
    class_no NOT NULL,
    PRIMARY KEY (bld_id, class_no),
    FOREIGN KEY (bld_id) REFERENCES
   Building(bld_id))
   CLUSTER Building_Cluster(bld_id);

CREATE OR REPLACE TYPE Lab_T AS OBJECT
   (bld_id        VARCHAR2(10),
    lab_no        VARCHAR2(10),
    lab_capacity  NUMBER,
    lab_equipment Equipments)
/

CREATE TABLE Lab OF Lab_T
   (bld_id NOT NULL,
    lab_no NOT NULL,
    PRIMARY KEY (bld_id, lab_no),
    FOREIGN KEY (bld_id) REFERENCES
   Building(bld_id))
   CLUSTER Building_Cluster(bld_id);

CREATE INDEX Building_Cluster_Index
   ON CLUSTER Building_Cluster;

— The Building_Details table has to be created before
we
— create the implementation of show_bld_details.

CREATE TABLE Building_Details
   (Building_Name      VARCHAR2(20),
    Building_Location  VARCHAR2(10));
```

Method Implementation

```
CREATE OR REPLACE TYPE BODY Building_T AS

   MEMBER PROCEDURE show_bld_details IS

   BEGIN
      INSERT INTO Building_Details
      VALUES (self.bld_name, self.bld_location);
   END show_bld_details;

END;
/
```

```
   — Before implementing this method, we need to create
the
   — tables for Person and Staff first. Otherwise, there
will
   — be a warning message during the procedure compilation.

   CREATE OR REPLACE TYPE BODY Office_T AS

      MEMBER PROCEDURE show_office IS

      CURSOR c_office IS
         SELECT c.pers_surname, b.off_no, b.off_phone
         FROM Building a, Office b, Person c, Staff d
         WHERE a.bld_id = self.bld_id AND a.bld_id =
      b.bld_id
         AND c.pers_id = d.pers_id AND d.in_office = REF
      (b);

      BEGIN
         DBMS_OUTPUT.PUT_LINE
            ('Surname'||'    '||'Office no'||'  '||'Office
         Phone');
         DBMS_OUTPUT.PUT_LINE
            ('————————————————————————');
         FOR v_office IN c_office LOOP
            DBMS_OUTPUT.PUT_LINE
               (v_office.pers_surname||' '||
                    v_office.off_no||'        '||
            v_office.off_phone);
         END LOOP;
      END show_office;

   END;
   /
```

Degree_T Class

For the Degree_T class, we will need the generic member method. In addition, there is also a user-defined method to store the data into the new table every time a new student has enrolled. For this purpose, we need to create a table named Degree_Records beforehand.

Relational Schemas

```
— The relationship from Degree_T to Faculty_T is
made
— using object references on attribute in_faculty.

Degree (deg_ID, deg_name, deg_length, deg_prereq,
in_faculty)
```

Class, Table, and Method Declaration

```
CREATE OR REPLACE TYPE Degree_T AS OBJECT
   (deg_id        VARCHAR2(10),
    deg_name      VARCHAR2(30),
    deg_length    VARCHAR2(10),
    deg_prereq    VARCHAR2(50),
    in_faculty REF Faculty_T,

    MEMBER PROCEDURE insert_degree(
      new_deg_id IN VARCHAR2,
      new_deg_name IN VARCHAR2,
      new_deg_length IN VARCHAR2,
      new_deg_prereq IN VARCHAR2,
      new_fac_id IN VARCHAR2),

    MEMBER PROCEDURE delete_degree,
    MEMBER PROCEDURE show_deg_record)
/

CREATE TABLE Degree OF Degree_T
   (deg_id NOT NULL,
    PRIMARY KEY (deg_id));

— The Degree_Records table has to be created before
we
— create the implementation of show_degree_records.

CREATE TABLE Degree_Records
   (deg_name      VARCHAR2(30),
    deg_length    VARCHAR2(10),
    deg_prereq    VARCHAR2(50),
    total_student NUMBER);
```

Method Implementation

```
— Before implementing this method, we need to create
the
— table for Person and Staff first.
```

```
CREATE OR REPLACE TYPE BODY Degree_T AS

   MEMBER PROCEDURE insert_degree(
      new_deg_id IN VARCHAR2,
      new_deg_name IN VARCHAR2,
      new_deg_length IN VARCHAR2,
      new_deg_prereq IN VARCHAR2,
      new_fac_id IN VARCHAR2) IS

      faculty_temp REF Faculty_T;

   BEGIN
      SELECT REF(a) INTO faculty_temp
      FROM Faculty a
      WHERE a.fac_id = new_fac_id;

      INSERT INTO Degree
      VALUES (new_deg_id, new_deg_name, new_deg_length,
              new_deg_prereq, faculty_temp);
   END insert_degree;

   MEMBER PROCEDURE delete_degree IS

   BEGIN
      DELETE FROM Degree
      WHERE deg_id = self.deg_id;
   END delete_degree;

   MEMBER PROCEDURE show_deg_record IS

   v_total INTEGER;

      SELECT COUNT (*) AS Total_Student
      INTO v_total
      FROM Degree a, Enrolls_In b
      WHERE b.degree = REF(a)
      GROUP BY a.deg_id;

   BEGIN
         INSERT INTO Degree_Records
         VALUES (self.deg_name, self.deg_length,
                 self.deg_prereq, v_total);
   END show_deg_record;

END;
/
```

Person_T Class, the Subclasses, and the Enrolls_In Table

The database for personal details is the biggest database needed for this case study. This is mainly because it involves multilevel inheritance. Person_T has union inheritance to its subclasses because a person can be a member of more than one subclass. Staff_T has partition inheritance to its subclasses because a staff can be the member of one, and only one, subclass. Finally, the inheritance type for Lecturer_T is also a partition type of inheritance.

Below is the implementation of these classes and their tables. Obviously, we will need the member method for insertion and deletion to most of these classes. In addition, according to the requirements, we need to add a user-defined method to display the details of the lecturers, their type, and their campus inside the Lecturer_T class.

```
Relational Schemas
   – Note that the association relationship between
   Person_T and Campus_T
   – and between Staff_T to Office_T is made using
   object references
   – respectively in attributes in_campus and
   in_office.

   Person (pers_ID, pers_surname, pers_fname, pers_title,
   pers_address, pers_phone, pers_postcode, in_campus)
   Staff (pers_ID, in_office, staff_type)
   Student (pers_ID, year)

Class, Table, and Method Declaration
   CREATE OR REPLACE TYPE Person_T AS OBJECT
      (pers_id       VARCHAR2(10),
       pers_surname  VARCHAR2(20),
       pers_fname    VARCHAR2(20),
       pers_title    VARCHAR2(10),
       pers_address  VARCHAR2(50),
       pers_phone    VARCHAR2(12),
       pers_postcode NUMBER,
       in_campus REF Campus_T,

       MEMBER PROCEDURE insert_person(
         new_pers_id IN VARCHAR2,
         new_pers_surname IN VARCHAR2,
```

```
        new_pers_fname IN VARCHAR2,
        new_pers_title IN VARCHAR2,
        new_pers_address IN VARCHAR2,
        new_pers_phone IN VARCHAR2,
        new_pers_postcode IN NUMBER,
        new_campus_location IN VARCHAR2),

      MEMBER PROCEDURE delete_person) NOT FINAL
/

CREATE TABLE Person OF Person_T
   (pers_id NOT NULL,
     PRIMARY KEY (pers_id));

— There is no generic method in Staff_T since it has
partition
— inheritance. Insertion and deletion have to be done
from the
— the subclasses.

CREATE OR REPLACE TYPE Staff_T UNDER Person_T
   (in_office REF Office_T,
     staff_type   VARCHAR2(20)) NOT FINAL
/

CREATE TABLE Staff OF Staff_T
   (pers_id NOT NULL,
     staff_type NOT NULL
   CHECK (staff_type IN 'Admin', 'Technician',
   'Senior_Lecturer', 'Associate_Lecturer', 'Tutor')),
     PRIMARY KEY (pers_id));

CREATE OR REPLACE TYPE Student_T UNDER Person_T
   (year    NUMBER,

     MEMBER PROCEDURE insert_student(
        new_pers_id IN VARCHAR2,
        new_pers_surname IN VARCHAR2,
        new_pers_fname IN VARCHAR2,
        new_pers_title IN VARCHAR2,
        new_pers_address IN VARCHAR2,
        new_pers_phone IN VARCHAR2,
        new_pers_postcode IN NUMBER,
        new_campus_location IN VARCHAR2,
        new_year IN NUMBER),
```

```
    MEMBER PROCEDURE delete_student)
/

CREATE TABLE Student OF Student_T
   (pers_id NOT NULL,
    PRIMARY KEY (pers_id));

CREATE OR REPLACE TYPE Admin_T UNDER Staff_T
   (admin_title   VARCHAR2(10),
    comp_skills   VARCHAR2(50),
    office_skills VARCHAR2(50),

    MEMBER PROCEDURE insert_admin(
     new_pers_id IN VARCHAR2,
     new_pers_surname IN VARCHAR2,
     new_pers_fname IN VARCHAR2,
     new_pers_title IN VARCHAR2,
     new_pers_address IN VARCHAR2,
     new_pers_phone IN VARCHAR2,
     new_pers_postcode IN NUMBER,
     new_campus_location IN VARCHAR2,
     new_bld_id IN VARCHAR2,
     new_off_no IN VARCHAR2,
     new_admin_title IN VARCHAR2,
     new_comp_skills IN VARCHAR2,
     new_office_skills IN VARCHAR2),

    MEMBER PROCEDURE delete_admin)
/

CREATE OR REPLACE TYPE Technician_T UNDER Staff_T
   (tech_title    VARCHAR2(10),
    tech_skills   VARCHAR2(50),

    MEMBER PROCEDURE insert_technician(
     new_pers_id IN VARCHAR2,
     new_pers_surname IN VARCHAR2,
     new_pers_fname IN VARCHAR2,
     new_pers_title IN VARCHAR2,
     new_pers_address IN VARCHAR2,
     new_pers_phone IN VARCHAR2,
     new_pers_postcode IN NUMBER,
     new_campus_location IN VARCHAR2,
     new_bld_id IN VARCHAR2,
     new_off_no IN VARCHAR2,
     new_tech_title IN VARCHAR2,
```

```
       new_tech_skills IN VARCHAR2),

     MEMBER PROCEDURE delete_technician)
/

— There is no generic method in Lecturer_T because
it has partition
— inheritance. Update operations are done through the
subclasses.

CREATE OR REPLACE TYPE Lecturer_T UNDER Staff_T
   (area          VARCHAR2(50),
     lect_type    VARCHAR2(20),

     MEMBER PROCEDURE show_lecturer) NOT FINAL
/

CREATE OR REPLACE TYPE Senior_Lecturer_T UNDER
Lecturer_T
   (no_phd        NUMBER,
     no_master    NUMBER,
     no_honours   NUMBER,

     MEMBER PROCEDURE insert_senior_lecturer(
       new_pers_id IN VARCHAR2,
       new_pers_surname IN VARCHAR2,
       new_pers_fname IN VARCHAR2,
       new_pers_title IN VARCHAR2,
       new_pers_address IN VARCHAR2,
       new_pers_phone IN VARCHAR2,
       new_pers_postcode IN NUMBER,
       new_campus_location IN VARCHAR2,
       new_bld_id IN VARCHAR2,
       new_off_no IN VARCHAR2,
       new_area IN VARCHAR2,
       new_no_phd IN NUMBER,
       new_no_master IN NUMBER,
       new_no_honours IN NUMBER),

     MEMBER PROCEDURE delete_senior_lecturer)
/

CREATE OR REPLACE TYPE Associate_Lecturer_T UNDER
Lecturer_T
   (no_honours    NUMBER,
     year_join    NUMBER,
```

```
    MEMBER PROCEDURE insert_associate_lecturer(
     new_pers_id IN VARCHAR2,
     new_pers_surname IN VARCHAR2,
     new_pers_fname IN VARCHAR2,
     new_pers_title IN VARCHAR2,
     new_pers_address IN VARCHAR2,
     new_pers_phone IN VARCHAR2,
     new_pers_postcode IN NUMBER,
     new_campus_location IN VARCHAR2,
     new_bld_id IN VARCHAR2,
     new_off_no IN VARCHAR2,
     new_area IN VARCHAR2,
     new_no_honours IN NUMBER,
     new_year_join IN NUMBER),

    MEMBER PROCEDURE delete_associate_lecturer)
/

CREATE OR REPLACE TYPE Tutor_T UNDER Staff_T
   (no_hours      NUMBER,
    rate          NUMBER,

    MEMBER PROCEDURE insert_tutor(
     new_pers_id IN VARCHAR2,
     new_pers_surname IN VARCHAR2,
     new_pers_fname IN VARCHAR2,
     new_pers_title IN VARCHAR2,
     new_pers_address IN VARCHAR2,
     new_pers_phone IN VARCHAR2,
     new_pers_postcode IN NUMBER,
     new_campus_location IN VARCHAR2,
     new_bld_id IN VARCHAR2,
     new_off_no IN VARCHAR2,
     new_year IN NUMBER, — from Student_T class
     new_no_hours IN NUMBER,
     new_rate IN NUMBER),

    MEMBER PROCEDURE delete_tutor)
/

— The Enrolls_In table is derived from the relationship
— between the Student_T and Degree_T classes.

CREATE TABLE Enrolls_In
   (student REF Student_T,
    degree REF Degree_T);
```

Methods Implementation

```
CREATE OR REPLACE TYPE BODY Person_T AS

   MEMBER PROCEDURE insert_person(
      new_pers_id IN VARCHAR2,
      new_pers_surname IN VARCHAR2,
      new_pers_fname IN VARCHAR2,
      new_pers_title IN VARCHAR2,
      new_pers_address IN VARCHAR2,
      new_pers_phone IN VARCHAR2,
      new_pers_postcode IN NUMBER,
      new_campus_location IN VARCHAR2) IS

      campus_temp REF Campus_T;

   BEGIN
      SELECT REF(a) INTO campus_temp
      FROM Campus a
      WHERE a.campus_location = new_campus_location;

      INSERT INTO Person
      VALUES (new_pers_id, new_pers_surname,
              new_pers_fname, new_pers_title,
              new_pers_address, new_pers_phone,
              new_pers_postcode, campus_temp);
   END insert_person;

   MEMBER PROCEDURE delete_person IS

   BEGIN
      DELETE FROM Person
      WHERE pers_id = self.pers_id;
   END delete_person;

END;
/

CREATE OR REPLACE TYPE BODY Student_T AS

   MEMBER PROCEDURE insert_student(
      new_pers_id IN VARCHAR2,
      new_pers_surname IN VARCHAR2,
      new_pers_fname IN VARCHAR2,
      new_pers_title IN VARCHAR2,
      new_pers_address IN VARCHAR2,
      new_pers_phone IN VARCHAR2,
      new_pers_postcode IN NUMBER,
```

```
        new_campus_location IN VARCHAR2,
        new_year IN NUMBER) IS

        campus_temp REF Campus_T;

    BEGIN
        SELECT REF(a) INTO campus_temp
        FROM Campus a
        WHERE a.campus_location = new_campus_location;

        INSERT INTO Student
        VALUES (new_pers_id, new_pers_surname,
                new_pers_fname, new_pers_title,
                new_pers_address, new_pers_phone,
               new_pers_postcode, campus_temp, new_year);
    END insert_student;

    MEMBER PROCEDURE delete_student IS

    BEGIN
            DELETE FROM Student
            WHERE pers_id = self.pers_id;
    END delete_student;

END;
/

CREATE OR REPLACE TYPE BODY Admin_T AS

    MEMBER PROCEDURE insert_admin(
        new_pers_id IN VARCHAR2,
        new_pers_surname IN VARCHAR2,
        new_pers_fname IN VARCHAR2,
        new_pers_title IN VARCHAR2,
        new_pers_address IN VARCHAR2,
        new_pers_phone IN VARCHAR2,
        new_pers_postcode IN NUMBER,
        new_campus_location IN VARCHAR2,
        new_bld_id IN VARCHAR2,
        new_off_no IN VARCHAR2,
        new_admin_title IN VARCHAR2,
        new_comp_skills IN VARCHAR2,
        new_office_skills IN VARCHAR2) IS

        campus_temp REF Campus_T;
        office_temp REF Office_T;
```

```
    BEGIN
       SELECT REF(a) INTO campus_temp
       FROM Campus a
       WHERE a.campus_location = new_campus_location;

       SELECT REF(b) INTO office_temp
       FROM Office b
       WHERE b.bld_id = new_bld_id
       AND b.off_no = new_off_no;

       INSERT INTO Staff
       VALUES (Admin_T(new_pers_id, new_pers_surname,
               new_pers_fname, new_pers_title,
               new_pers_address, new_pers_phone,
                new_pers_postcode, campus_temp,
             office_temp, 'Admin',
             new_admin_title, new_comp_skills,
             new_office_skills));
    END insert_admin;

    MEMBER PROCEDURE delete_admin IS

    BEGIN
       DELETE FROM Staff
       WHERE pers_id = self.pers_id;
    END delete_admin;

END;
/

CREATE OR REPLACE TYPE BODY Technician_T AS

    MEMBER PROCEDURE insert_technician(
       new_pers_id IN VARCHAR2,
       new_pers_surname IN VARCHAR2,
       new_pers_fname IN VARCHAR2,
       new_pers_title IN VARCHAR2,
       new_pers_address IN VARCHAR2,
       new_pers_phone IN VARCHAR2,
       new_pers_postcode IN NUMBER,
       new_campus_location IN VARCHAR2,
       new_bld_id IN VARCHAR2,
       new_off_no IN VARCHAR2,
       new_tech_title IN VARCHAR2,
       new_tech_skills IN VARCHAR2) IS
```

```
      campus_temp REF Campus_T;
      office_temp REF Office_T;

   BEGIN
      SELECT REF(a) INTO campus_temp
      FROM Campus a
      WHERE a.campus_location = new_campus_location;

      SELECT REF(b) INTO office_temp
      FROM Office b
      WHERE b.bld_id = new_bld_id
      AND b.off_no = new_off_no;

      INSERT INTO Staff
      VALUES (Technician_T(new_pers_id,
             new_pers_surname,
             new_pers_fname, new_pers_title,
             new_pers_address, new_pers_phone,
              new_pers_postcode, campus_temp,
            office_temp, 'Technician',
             new_tech_title, new_tech_skills));
   END insert_technician;

   MEMBER PROCEDURE delete_technician IS

   BEGIN
      DELETE FROM Staff
      WHERE pers_id = self.pers_id;
   END delete_technician;

END;
/

CREATE OR REPLACE TYPE BODY Lecturer_T AS

   MEMBER PROCEDURE show_lecturer IS

   BEGIN
         DBMS_OUTPUT.PUT_LINE
         (self.pers_surname||'  '||self.pers_fname||'
         '||
          self.pers_address||' '||self.lect_type||' '||
           self.area||'  '||self.lect_type);
   END show_lecturer;

END;
```

```
/

CREATE OR REPLACE TYPE BODY Senior_Lecturer_T AS

   MEMBER PROCEDURE insert_senior_lecturer(
      new_pers_id IN VARCHAR2,
      new_pers_surname IN VARCHAR2,
      new_pers_fname IN VARCHAR2,
      new_pers_title IN VARCHAR2,
      new_pers_address IN VARCHAR2,
      new_pers_phone IN VARCHAR2,
      new_pers_postcode IN NUMBER,
      new_campus_location IN VARCHAR2,
      new_bld_id IN VARCHAR2,
      new_off_no IN VARCHAR2,
      new_area IN VARCHAR2,
      new_no_phd IN NUMBER,
      new_no_master IN NUMBER,
      new_no_honours IN NUMBER) IS

      campus_temp REF Campus_T;
      office_temp REF Office_T;

   BEGIN
      SELECT REF(a) INTO campus_temp
      FROM Campus a
      WHERE a.campus_location = new_campus_location;

      SELECT REF(b) INTO office_temp
      FROM Office b
      WHERE b.bld_id = new_bld_id
      AND b.off_no = new_off_no;

      INSERT INTO Staff
      VALUES (Senior_Lecturer_T(new_pers_id,
      new_pers_surname,
            new_pers_fname, new_pers_title,
            new_pers_address,
            new_pers_phone, new_pers_postcode,
            campus_temp,
            office_temp, 'Lecturer', new_area, 'Senior
            Lecturer',
            new_no_phd, new_no_master,
            new_no_honours);
   END insert_senior_lecturer;
```

```
   MEMBER PROCEDURE delete_senior_lecturer IS

   BEGIN
      DELETE FROM Staff
      WHERE pers_id = self.pers_id;
   END delete_senior_lecturer;

END;
/

CREATE OR REPLACE TYPE BODY Associate_lecturer_T AS

   MEMBER PROCEDURE insert_associate_lecturer(
      new_pers_id IN VARCHAR2,
      new_pers_surname IN VARCHAR2,
      new_pers_fname IN VARCHAR2,
      new_pers_title IN VARCHAR2,
      new_pers_address IN VARCHAR2,
      new_pers_phone IN VARCHAR2,
      new_pers_postcode IN NUMBER,
      new_campus_location IN VARCHAR2,
      new_bld_id IN VARCHAR2,
      new_off_no IN VARCHAR2,
      new_area IN VARCHAR2,
      new_no_honours IN NUMBER,
      new_year_join IN NUMBER) IS

      campus_temp REF Campus_T;
      office_temp REF Office_T;

   BEGIN
      SELECT REF(a) INTO campus_temp
      FROM Campus a
      WHERE a.campus_location = new_campus_location;

      SELECT REF(b) INTO office_temp
      FROM Office b
      WHERE b.bld_id = new_bld_id
      AND b.off_no = new_off_no;

      INSERT INTO Staff
      VALUES (Associate_Lecturer_T(new_pers_id,
      new_pers_surname,
            new_pers_fname,   new_pers_title,
             new_pers_address,
```

```
            new_pers_phone, new_pers_postcode,
             campus_temp,
            office_temp, 'Lecturer', new_area,
             'Associate Lecturer'
             new_no_honours, new_year_join));
    END insert_associate_lecturer;

    MEMBER PROCEDURE delete_associate_lecturer IS

    BEGIN
       DELETE FROM Staff
       WHERE pers_id = self.pers_id;
    END delete_associate_lecturer;

END;
/

CREATE OR REPLACE TYPE BODY Tutor_T AS

    MEMBER PROCEDURE insert_tutor(
       new_pers_id IN VARCHAR2,
       new_pers_surname IN VARCHAR2,
       new_pers_fname IN VARCHAR2,
       new_pers_title IN VARCHAR2,
       new_pers_address IN VARCHAR2,
       new_pers_phone IN VARCHAR2,
       new_pers_postcode IN NUMBER,
       new_campus_location IN VARCHAR2,
       new_bld_id IN VARCHAR2,
       new_off_no IN VARCHAR2,
       new_year IN NUMBER,
       new_no_hours IN NUMBER,
       new_rate IN NUMBER) IS

       campus_temp REF Campus_T;
       office_temp REF Office_T;

    BEGIN
       SELECT REF(a) INTO campus_temp
       FROM Campus a
       WHERE a.campus_location = new_campus_location;

       SELECT REF(b) INTO office_temp
       FROM Office b
       WHERE b.bld_id = new_bld_id
       AND b.off_no = new_off_no;
```

```
      INSERT INTO Staff
      VALUES (Tutor_T(new_pers_id, new_pers_surname,
             new_pers_fname, new_pers_title,
             new_pers_address, new_pers_phone,
             new_pers_postcode, campus_temp,
            office_temp, 'Tutor',
             new_no_hours, new_rate));
   END insert_tutor;

   MEMBER PROCEDURE delete_tutor IS

   BEGIN
      DELETE FROM Staff
      WHERE pers_id = self.pers_id;
   END delete_tutor;

END;
/

— Beside member methods, we also need to provide the
— stored procedures for the Enrolls_In table.

CREATE OR REPLACE PROCEDURE Insert_Enrolls_In(
   new_pers_id IN Person.pers_id%TYPE,
   new_deg_id IN Degree.deg_id%TYPE) AS

   student_temp REF Student_T;
   degree_temp REF Degree_T;

BEGIN
   SELECT REF(a) INTO student_temp
   FROM Student a
   WHERE a.pers_id = new_pers_id;

   SELECT REF(b) INTO degree_temp
   FROM Degree b
   WHERE b.deg_id = new_deg_id;

   INSERT INTO Enrolls_In
   VALUES (student_temp, degree_temp);
END Insert_Enrolls_In;
/

CREATE OR REPLACE PROCEDURE Delete_Enrolls_In(
   deleted_pers_id IN Person.pers_id%TYPE,
   deleted_deg_id IN Degree.deg_id%TYPE) AS
```

```
BEGIN
   DELETE FROM Enrolls_In
   WHERE Enrolls_In.student IN
      (SELECT REF(a)
       FROM Student a
       WHERE a.pers_id = deleted_pers_id)
   AND Enrolls_In.degree IN
      (SELECT REF(b)
       FROM Degree b
       WHERE b.deg_id = deleted_deg_id);
END Delete_Enrolls_In;
/
```

Subject_T Class and Takes Table

The next class to be implemented is Subject_T, which has an association relationship with Student_T. However, as both are of equal importance, we cannot use a foreign key or object reference in either of them. Thus, another table Takes needs to be created that includes the object references to previous classes and an additional attribute, in this case, Marks.

```
Relational Schemas
   – The relationship between Subject_T and
   Lecturer_T is made using the object
   – reference Teach. The attributes inside the Takes
   table are also
   – implemented using object references Subject and
   Lecturer.

   Subject (subj_ID, subj_name, subj_credit,
   subj_prereq, teach)
   Takes (subject, lecturer, marks)

Class, Table, and Method Declaration
   CREATE OR REPLACE TYPE Subject_T AS OBJECT
      (subj_id       VARCHAR2(10),
       subj_name     VARCHAR2(30),
       subj_credit   VARCHAR2(10),
       subj_prereq   VARCHAR2(50),
       teach REF Lecturer_T,

       MEMBER PROCEDURE insert_subject(
        new_subj_id IN VARCHAR2,
        new_subj_name IN VARCHAR2,
```

```
        new_subj_credit IN VARCHAR2,
        new_subj_prereq IN VARCHAR2,
        new_pers_id IN VARCHAR2),

      MEMBER PROCEDURE delete_subject)
/

CREATE TABLE Subject OF Subject_T
   (subj_id NOT NULL,
    PRIMARY KEY (subj_id));

CREATE TABLE Takes
   (student REF Student_T,
    subject REF Subject_T,
    marks  NUMBER);
```

Methods Implementation

```
CREATE OR REPLACE TYPE BODY Subject_T AS

   MEMBER PROCEDURE insert_subject(
      new_subj_id IN VARCHAR2,
      new_subj_name IN VARCHAR2,
      new_subj_credit IN VARCHAR2,
      new_subj_prereq IN VARCHAR2,
      new_pers_id IN VARCHAR2) IS

      lecturer_temp REF Lecturer_T;

   BEGIN
      SELECT REF(a) INTO lecturer_temp
      FROM Lecturer a
      WHERE a.pers_id = new_pers_id;

      INSERT INTO Subject
      VALUES (new_subj_id, new_subj_name,
      new_subj_credit,
           new_subj_prereq, lecturer_temp);
   END insert_subject;

   MEMBER PROCEDURE delete_subject IS

   BEGIN
      DELETE FROM Subject
      WHERE subj_id = self.subj_id;
   END delete_subject;
```

```
END;
/

CREATE OR REPLACE PROCEDURE Insert_Takes(
   new_pers_id IN Person.pers_id%TYPE,
   new_subj_id IN Subject.subj_id%TYPE,
   new_marks IN NUMBER) AS

   student_temp REF Student_T;
   subject_temp REF Subject_T;

BEGIN
   SELECT REF(a) INTO student_temp
   FROM Student a
   WHERE a.pers_id = new_pers_id;

   SELECT REF(b) INTO subject_temp
   FROM Subject b
   WHERE b.subj_id = new_subj_id;

   INSERT INTO Takes
   VALUES (student_temp, subject_temp, new_marks);
END Insert_Takes;
/

CREATE OR REPLACE PROCEDURE Delete_Takes(
   deleted_pers_id IN Person.pers_id%TYPE,
   deleted_subj_id IN Subject.subj_id%TYPE) AS

BEGIN
   DELETE FROM Takes
   WHERE Takes.student IN
       (SELECT REF(a)
        FROM Student a
        WHERE a.pers_id = deleted_pers_id)
   AND Takes.subject IN
       (SELECT REF(b)
        FROM Subject b
        WHERE b.subj_id = deleted_subj_id);
END Delete_Takes;
/
```

Sample Database Execution

In this section, we will demonstrate the execution of the created database. There will be a simple example on how to use the generic and the user-defined methods for some classes. We will also try to display the results of some of the retrieval methods. Unlike the section of Problem Solutions, we will not divide this section based on the classes but rather on the type of member methods, that is, generic methods and user-defined methods.

Generic Methods Sample

Most classes in this case study have generic member methods attached to them. The methods are used for insertion into, and deletion from, the object tables. Besides the generic member methods, there are also generic methods that are implemented as stored procedures. The two tables with these stored procedures are Enrolls_In and Takes.

Notice that the order of action will be very important because a record in a table might refer to another record in another table or object. The wrong order of deletion, for example, might result in having dangling object references. It might happen because the ORDB has not preserved a complete integrity constraint checking.

The first class where data needs to be inserted is Campus_T. As there are neither generic member methods nor generic stored procedures implemented, we have to use an ad hoc query to insert data into the Campus table.

```
INSERT INTO Campus
VALUES ('Albury/Wodonga', 'Parkers Road Wodonga VIC 3690',
'61260583700', '620260583777', 'John Hill');

INSERT INTO Campus
VALUES ('City', '215 Franklin St. Melb VIC 3000',
'61392855100', '61392855111', 'Michael A. Leary');

INSERT INTO Campus
VALUES ('Mildura', 'Benetook Ave. Mildura VIC 3502',
'61350223757', '61350223646', 'Ron Broadhead');

INSERT INTO Campus
```

```
VALUES ('Bundoora', 'Kingsbury Dv Bundoora VIC 3083',
'61395485410', '61398520148', 'Michael Osborne');
```

We can check the records by retrieving the data. The retrieval query on table Campus will display the result shown below.

```
SELECT campus_location, campus_address
FROM Campus;

CAMPUS_LOCATION          CAMPUS_ADDRESS
___________  _________________

Albury/Wodonga           Parkers Road Wodonga VIC 3690
City                      215 Franklin St. Melb VIC 3000
Mildura                   Benetook Ave. Mildura VIC 3502
Bundoora                  Kingsbury Dv Bundoora VIC 3083
```

The next class to be implemented is Faculty_T and its nested tables. These classes also do not have generic member methods and thus, we need to use an ad hoc query like that for the Campus table. We will show the sample to demonstrate the application for nested tables.

```
INSERT INTO Faculty
VALUES ('1', 'Health Sciences', 'S.Duckett',
   School_Table_T(School_T(NULL,NULL,NULL,NULL)),
   Department_Table_T(Department_T(NULL,NULL,NULL,NULL)),
   Research_Centre_Table_T(Research_Centre_T(NULL,NULL,NULL,NULL)));

INSERT INTO Faculty
VALUES ('4', 'Science, Tech, Eng.', 'D.Finlay',
   School_Table_T(School_T(NULL,NULL,NULL,NULL)),
   Department_Table_T(Department_T(NULL,NULL,NULL,NULL)),
   Research_Centre_Table_T(Research_Centre_T(NULL,NULL,NULL,NULL)));
```

Note that we need a constructor for each nested table. It is a requirement before we are able to insert the values in these nested tables. The insertion example is shown below.

```
INSERT INTO THE
   (SELECT a.school
    FROM Faculty a
    WHERE a.fac_id = '1')
```

```
VALUES ('1-1', 'Human Biosciences', 'Chris Handley',
Professors(Professor_T('110', 'Chris Handley',
'0394584521', 1980)));

INSERT INTO THE
   (SELECT a.school
    FROM Faculty a
    WHERE a.fac_id = '1')
VALUES ('1-2', 'Human Comm. Sci.', 'Elizabeth
Lavender', Professors(Professor_T('120', 'Sheena
Reiley', '0395420001', 1991), Professor_T('130',
'Alison Perry', '0398219234', 1995),
Professor_T('140', 'Jan Branson', '0387210023',
2001)));

INSERT INTO THE
   (SELECT a.department
    FROM Faculty a
    WHERE a.fac_id = '4')
VALUES ('4-1', 'Agricultural Sci.', 'Mark Sandeman',
Professors(Professor_T(NULL,NULL,NULL,NULL)));
```

The deletion of a particular faculty from the Faculty table will delete all nested tables inside it. On the other side, we can delete a nested table without deleting the faculty. The disadvantage is that we have to delete the whole nested table record and we are not allowed to choose a particular record in the nested table. A simple SQL code below shows the deletion of a department record. The deletion of a faculty record is pretty straightforward and thus is not shown here.

```
DELETE FROM THE
   (SELECT a.department
    FROM Faculty a
    WHERE a.fac_id = '4');
```

The next class is Building_T and its subclasses. They will be implemented also using an ad hoc query. Although this is an aggregation using the clustering technique, the implementation of insertion and deletion will be very similar to that of the previous classes. Now we will show the example of generic member method usage in the Degree_T class.

```
DECLARE
   — Construct objects and initialise them to null.
   a_degree Degree_T := Degree_T
   (NULL,NULL,NULL,NULL,NULL);

BEGIN
   a_degree.insert_degree('D100', 'Bachelor of Comp.
   Sci', '3', 'Year 12 or Equivalent', '4');
   a_degree.insert_degree('D101', 'Master of Comp.
   Sci', '2', 'Bachelor of Comp. Sci', '4');
END;
/

SELECT deg_id, deg_name, deg_length
FROM Degree;

   DEG_ID      DEG_NAME                          DEG_LENGTH
   ------------------------------------------------------
   D100          Bachelor of Comp. Sci                    3
   D101          Master of Comp. Sci                      2
```

Deletion from this table is very simple and basically very similar to the implementation of insertion. The code below shows the implementation of the deletion member method. On completion of this method, the degree with a particular ID will be deleted.

```
BEGIN
   a_degree.delete_degree;
END;
/
```

We will not provide the examples of generic method implementation for the Person_T class and its subclasses because it is very similar to the implementation in the Degree_T class. However, there are a few things to remember. First, the insertion in a superclass might be done (and has to be done for partition inheritance) from the subclasses. Second, a deletion from the superclass will delete the data for the particular record in the subclasses as the consequence of the referential integrity constraint.

The implementation for the Subject_T class is very similar to the implementation in Degree_T. However, this is not the case for table Takes that is derived from the relationship between Subject_T and Student_T. The insertion has to be done using a stored procedure as is shown below. Note that in order to insert it, we have to make sure that the object references in the other classes have already been inserted.

```
EXECUTE Insert_Takes('01234234', 'CSE42ADB', 70);
EXECUTE Insert_Takes('10012568', 'CSE42ADB', 80);
```

We can check the records by retrieving the data. The code below shows the result of the query on table Takes. Note that the first two attributes show the address of the object it is referred to.

```
SELECT *
FROM Takes;

   STUDENT              SUBJECT              MARKS
   ------------------------------------------------
   0000220208A1…        0000220208D7…        70
   00002202084E…        000022020873…        80
```

Another piece of code below shows the implementation of deletion using the stored procedure Delete_Takes. On completion of this method, the student with a particular ID who takes a particular subject will be deleted.

```
EXECUTE Delete_Takes('01234234', 'CSE42ADB');
EXECUTE Delete_Takes('10012568'', 'CSE42ADB');
```

User-Defined Methods Sample

In this section, we give an example of a user-defined implementation. As in generic methods, we can also make an ad hoc user-defined query and user-defined stored procedure or function. However, we will only provide an example for the user-defined methods that have been created previously.

The first sample implementation is for Faculty_T. The method show_parts will display the names and the heads of the schools, departments, and research centres given the faculty ID as the parameter.

```
BEGIN
   — Assume the parameter is the faculty with ID 1;
   thus, the result
   — shown is the school, department, and research
   centre under faculty ID 1.

   a_faculty.show_parts;
END;
/

Part Name                    Head Name
______________________________________________
Human Biosciences            Chris Handley
Human Comm. Sci.             Elizabeth Lavender
Sex, Health, and Soc.        Marian Pitts
Inst. of Primary Care        Hal Swerissen

PL/SQL procedure successfully completed.
```

The piece of code below shows the record inside the Building_Details table before the method is executed. The next code shows the data that has been inserted into the table after we execute the method.

```
SELECT *
FROM Building_Details;

no rows selected

SELECT *
FROM Building_Details;

BUILDING_NAME              BUILDING_L
_____________________________________
Beth Gleeson                 D5
Martin Building              F3
Thomas Cherry                D4
Physical Science 1           D5
```

Another sample implementation is for Office_T to display the details of the office and the occupant given the building ID.

```
BEGIN
   — Assume that the parameter is building BB1; thus,
   the result
   — shown below is all offices in building BB1.

   an_office.show_office('BB1');
END;
/

Surname          Office No    Office Phone
__________________________________________

Jones                  BG210        94792001
Zulu                   BG325        94791251
Stojnovski       BG310        94791212
Langley          BG311        94791213
Ling                   BG200        94792350
Husein           BG215        94792341
Xin                    BG212        94792002
Kilby                  BG220        94792450

PL/SQL procedure successfully completed.
```

There are a few other methods that have been created in this case study. However, we will not show all of them as the previous examples have clearly shown how to execute the user-defined methods.

Building Case Application

In Section 7.2 we provided a problem solution that is constructed of several small types, tables, and procedures. Despite their ability to address the problem, they are not really simple to use. Users will easily forget the names of the tables and procedures, the number and the order of the parameters, and so forth. Therefore, we need to put them together into one container that can help users to choose the object that they want to use.

Oracle™ implements a PL/SQL container named Package that can group procedures and functions together. Unfortunately, Package in Oracle™ does not recognize object types. Thus, to access member methods, we have to apply helper stored procedures as an additional layer.

For some operations, there will be redundancy because users need to repeat the methods. On the other side, keeping them together makes the application more user friendly. In addition, we can also make the application more interactive by providing a menu to the users.

Like object type, in a package, the user divides the process into two parts: the declaration and the implementation or the header and the body. The code below shows the whole implementation of this case study inside an application name University.

```
General Syntax:

CREATE [OR REPLACE] PACKAGE <package schema>
   – public
   TYPE <type name> IS RECORD [(record attribute)];
   PROCEDURE <procedure name> [(procedure
   parameters)];
     ...
END <package name>;

CREATE [OR REPLACE] PACKAGE BODY <package schema>
   – private
   TYPE <type name> IS RECORD [(record attribute)];
   PROCEDURE <procedure name> [(procedure parameters)]
   IS
   BEGIN
      <procedure body>
   END <procedure name>;
     ...
END <package name>;
```

```
CREATE OR REPLACE PACKAGE University AS
   PROCEDURE Start_Program;
   PROCEDURE Table_Details;
   PROCEDURE Method_Details;
   PROCEDURE Insertion(options IN NUMBER);
   PROCEDURE Insert_Campus(new_campus_location IN
      VARCHAR2, new_campus_address IN VARCHAR2,
      new_campus_phone IN VARCHAR2, new_campus_fax IN
      VARCHAR2,
      new_campus_head IN VARCHAR2);
   PROCEDURE Insert_Faculty(new_fac_id IN VARCHAR2,
   new_fac_name IN VARCHAR2,
      new_fac_dean IN VARCHAR2);
```

```
PROCEDURE Insert_School(new_fac_id IN VARCHAR2,
new_school_id IN VARCHAR2,
   new_school_name IN VARCHAR2, new_school_head IN
   VARCHAR2, new_prof_id IN VARCHAR2, new_prof_name IN
   VARCHAR2, new_prof_contact IN VARCHAR2,
   new_prof_year IN NUMBER);
PROCEDURE Insert_Department(new_fac_id IN VARCHAR2,
new_dept_id IN VARCHAR2,
   new_dept_name IN VARCHAR2, new_dept_head IN
   VARCHAR2, new_prof_id IN VARCHAR2, new_prof_name IN
   VARCHAR2, new_prof_contact IN VARCHAR2,
   new_prof_year IN NUMBER);
PROCEDURE Insert_Research_Centre(new_fac_id IN
   VARCHAR2, new_rc_id IN VARCHAR2, new_rc_name IN
   VARCHAR2, new_rc_head IN VARCHAR2, new_unit1 IN
   VARCHAR2, new_unit2 IN VARCHAR2, new_unit3 IN
   VARCHAR2, new_unit4 IN VARCHAR2, new_unit5 IN
   VARCHAR2);
PROCEDURE Insert_Building(new_building_id IN VARCHAR2,
   new_building_name IN VARCHAR2, new_building_location
   IN VARCHAR2, new_building_level IN NUMBER,
   new_campus_location IN VARCHAR2, new_faculty_id IN
   VARCHAR2);
PROCEDURE Insert_Office(new_building_id IN VARCHAR2,
   new_office_no IN VARCHAR2, new_office_phone IN
   VARCHAR2);
PROCEDURE Insert_Classroom(new_building_id IN VARCHAR2,
   new_class_no IN VARCHAR2, new_class_capacity IN
   NUMBER);
PROCEDURE Insert_Lab(new_building_id IN VARCHAR2,
   new_lab_no IN VARCHAR2, new_lab_capacity IN NUMBER,
   new_lab_equipment_1 IN VARCHAR2, new_lab_equipment_2
   IN VARCHAR2, new_lab_equipment_3 IN VARCHAR2,
   new_lab_equipment_4 IN VARCHAR2, new_lab_equipment_5
   IN VARCHAR2);
PROCEDURE Insert_Degree(new_degree_id IN VARCHAR2,
   new_degree_name IN VARCHAR2, new_degree_length IN
   VARCHAR2, new_degree_prerequisite IN VARCHAR2,
   new_faculty_id IN VARCHAR2);
PROCEDURE Insert_Person(new_person_id IN VARCHAR2,
   new_person_surname     IN VARCHAR2,
   new_person_fname IN VARCHAR2, new_person_title IN
   VARCHAR2, new_person_address IN VARCHAR2,
   new_person_phone IN VARCHAR2, new_person_postcode
   IN NUMBER, new_campus_location IN VARCHAR2);
```

```
PROCEDURE Insert_Student(new_person_id IN VARCHAR2,
   new_person_surname IN VARCHAR2, new_person_fname IN
   VARCHAR2, new_person_title IN VARCHAR2,
   new_person_address IN VARCHAR2, new_person_phone IN
   VARCHAR2, new_person_postcode IN NUMBER,
   new_campus_location IN VARCHAR2,
   new_year IN NUMBER);
PROCEDURE Insert_Admin(new_person_id IN VARCHAR2,
   new_person_surname IN VARCHAR2, new_person_fname IN
   VARCHAR2, new_person_title IN VARCHAR2,
   new_person_address IN VARCHAR2, new_person_phone IN
   VARCHAR2, new_person_postcode IN NUMBER,
   new_campus_location IN VARCHAR2, new_building_id IN
   VARCHAR2, new_office_no IN VARCHAR2,
   new_admin_title IN VARCHAR2, new_comp_skills IN
   VARCHAR2, new_office_skills IN VARCHAR2);
PROCEDURE Insert_Technician(new_person_id IN VARCHAR2,
   new_person_surname IN VARCHAR2, new_person_fname IN
   VARCHAR2, new_person_title IN VARCHAR2,
   new_person_address IN VARCHAR2, new_person_phone IN
   VARCHAR2, new_person_postcode IN NUMBER,
   new_campus_location IN VARCHAR2, new_building_id IN
   VARCHAR2, new_office_no IN VARCHAR2, new_tech_title
   IN VARCHAR2, new_tech_skills IN VARCHAR2);
PROCEDURE Insert_Senior_Lecturer(new_person_id IN
   VARCHAR2, new_person_surname IN VARCHAR2,
   new_person_fname IN VARCHAR2, new_person_title IN
   VARCHAR2, new_person_address IN VARCHAR2,
   new_person_phone IN VARCHAR2, new_person_postcode
   IN NUMBER, new_campus_location IN VARCHAR2,
   new_building_id IN VARCHAR2, new_office_no IN
   VARCHAR2, new_area IN VARCHAR2, new_no_phd IN
   NUMBER, new_no_master IN NUMBER, new_no_honours IN
   NUMBER);
PROCEDURE Insert_Associate_Lecturer(new_person_id IN
   VARCHAR2, new_person_surname IN VARCHAR2,
   new_person_fname IN VARCHAR2, new_person_title IN
   VARCHAR2, new_person_address IN VARCHAR2,
   new_person_phone IN VARCHAR2, new_person_postcode
   IN NUMBER, new_campus_location IN VARCHAR2,
   new_building_id IN VARCHAR2, new_office_no IN
   VARCHAR2, new_area IN VARCHAR2, new_no_honours IN
   NUMBER, new_year_join IN NUMBER);
PROCEDURE Insert_Tutor(new_person_id IN VARCHAR2,
   new_person_surname IN VARCHAR2, new_person_fname IN
   VARCHAR2, new_person_title IN VARCHAR2,
```

```
   new_person_address IN VARCHAR2, new_person_phone IN
   VARCHAR2, new_person_postcode IN NUMBER,
   new_campus_location IN VARCHAR2, new_building_id IN
   VARCHAR2, new_office_no IN VARCHAR2, new_year IN
   NUMBER, new_no_hours IN NUMBER, new_rate IN
   NUMBER);
PROCEDURE Insert_Enrolls_In(new_pers_id IN VARCHAR2,
   new_deg_id IN VARCHAR2);
PROCEDURE Insert_Subject(new_subject_id IN VARCHAR2,
   new_subject_name IN VARCHAR2, new_subject_credit IN
   VARCHAR2, new_subject_prereq IN VARCHAR2,
   new_person_id IN VARCHAR2);
PROCEDURE Insert_Takes(new_pers_id IN VARCHAR2, new_subj_id
   IN VARCHAR2, new_marks IN NUMBER);

PROCEDURE Deletion(options IN NUMBER);
PROCEDURE Delete_Campus(deleted_campus_location IN
   VARCHAR2);
PROCEDURE Delete_Faculty(deleted_fac_id IN VARCHAR2);
PROCEDURE Delete_School(deleted_fac_id IN VARCHAR2);
PROCEDURE Delete_Department(deleted_fac_id IN
VARCHAR2);
PROCEDURE Delete_Research_centre(deleted_fac_id IN
VARCHAR2);
PROCEDURE Delete_Building(deleted_building_id IN
VARCHAR2);
PROCEDURE Delete_Office(deleted_building_id IN
   VARCHAR2, deleted_office_no IN VARCHAR2);
PROCEDURE Delete_Classroom(deleted_building_id IN
   VARCHAR2, deleted_class_no IN VARCHAR2);
PROCEDURE Delete_Lab(deleted_building_id IN VARCHAR2,
   deleted_lab_no IN VARCHAR2);
PROCEDURE Delete_Degree(deleted_degree_id IN VARCHAR2);
PROCEDURE Delete_Person(deleted_person_id IN VARCHAR2);
PROCEDURE Delete_Student(deleted_person_id IN
VARCHAR2);
PROCEDURE Delete_Admin(deleted_person_id IN VARCHAR2);
PROCEDURE Delete_Technician(deleted_person_id IN
VARCHAR2);
PROCEDURE Delete_Senior_Lecturer(deleted_person_id IN
VARCHAR2);
PROCEDURE Delete_Associate_Lecturer(deleted_person_id
IN VARCHAR2);
PROCEDURE Delete_Tutor(deleted_person_id IN VARCHAR2);
PROCEDURE Delete_Enrolls_In(deleted_pers_id IN VARCHAR2,
   deleted_deg_id IN VARCHAR2);
```

```
   PROCEDURE Delete_Subject(deleted_subject_id IN
   VARCHAR2);
   PROCEDURE Delete_Takes(deleted_pers_id IN VARCHAR2,
      deleted_subj_id IN VARCHAR2);

END University;
/

CREATE OR REPLACE PACKAGE BODY University AS

   PROCEDURE Start_Program AS

   BEGIN
      DBMS_OUTPUT.PUT_LINE('—————————————————————————————————
      ——————————');
      DBMS_OUTPUT.PUT_LINE('For insertion, type "EXECUTE
      University.Insertion("table_no");"');
      DBMS_OUTPUT.PUT_LINE('For deletion, type "EXECUTE
      University.Deletion   ("table_no");"');
      DBMS_OUTPUT.PUT_LINE('For retrieval, type "EXECUTE
      University.Retrieval ("procedure no");"');
      DBMS_OUTPUT.PUT_LINE('—————————————————————————————————
      ——————————');
      DBMS_OUTPUT.PUT_LINE('To check the table no, type
      "EXECUTE University.Table_Details;"');
      DBMS_OUTPUT.PUT_LINE('To check the procedure no,
      type "EXECUTE University.Procedure_Details;"');
   END Start_Program;

   PROCEDURE Table_Details AS

   BEGIN
      DBMS_OUTPUT.PUT_LINE('————————————————');
      DBMS_OUTPUT.PUT_LINE('—————Table Name—————');
      DBMS_OUTPUT.PUT_LINE('————————————————');
      DBMS_OUTPUT.PUT_LINE('( 1) Campus');
      DBMS_OUTPUT.PUT_LINE('( 2) Faculty');
      DBMS_OUTPUT.PUT_LINE('( 3) School (Nested Table)');
      DBMS_OUTPUT.PUT_LINE('( 4) Department (Nested
      Table)');
      DBMS_OUTPUT.PUT_LINE('( 5) Research Centre (Nested
      Table)');
      DBMS_OUTPUT.PUT_LINE('( 6) Building');
      DBMS_OUTPUT.PUT_LINE('( 7) Office');
      DBMS_OUTPUT.PUT_LINE('( 8) Classroom');
      DBMS_OUTPUT.PUT_LINE('( 9) Lab');
```

```
      DBMS_OUTPUT.PUT_LINE('(10) Degree');
      DBMS_OUTPUT.PUT_LINE('(11) Person');
      DBMS_OUTPUT.PUT_LINE('(12) Staff');
      DBMS_OUTPUT.PUT_LINE('(13) Student');
      DBMS_OUTPUT.PUT_LINE('(14) Admin');
      DBMS_OUTPUT.PUT_LINE('(15) Technician');
      DBMS_OUTPUT.PUT_LINE('(16) Lecturer');
      DBMS_OUTPUT.PUT_LINE('(17) Senior_Lecturer');
      DBMS_OUTPUT.PUT_LINE('(18) Associate_Lecturer');
      DBMS_OUTPUT.PUT_LINE('(19) Tutor');
      DBMS_OUTPUT.PUT_LINE('(20) Enrolls_In');
      DBMS_OUTPUT.PUT_LINE('(21) Subject');
      DBMS_OUTPUT.PUT_LINE('(22) Takes');
   END Table_Details;

   PROCEDURE Procedure_Details AS

   BEGIN
      DBMS_OUTPUT.PUT_LINE('————————————————————————————————
      ——————————');
      DBMS_OUTPUT.PUT_LINE('—Frequent Retrieval Procedure
      Name—');
      DBMS_OUTPUT.PUT_LINE('————————————————————————————————
      ——————————');
      DBMS_OUTPUT.PUT_LINE('( 1) Show Professor');
      DBMS_OUTPUT.PUT_LINE('( 2) Show Building Details');
      DBMS_OUTPUT.PUT_LINE('( 3) Show Office');
      DBMS_OUTPUT.PUT_LINE('( 4) Show Degree Record');
      DBMS_OUTPUT.PUT_LINE('( 5) Show Lecturer');
   END Procedure_Details;

   PROCEDURE Insertion(options IN NUMBER) AS

   BEGIN
      DBMS_OUTPUT.PUT_LINE('————————————————————————————————
      ——————————');
      IF options = 1 THEN
         DBMS_OUTPUT.PUT_LINE('Insert into Campus');
         DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
         University.Insert_Campus(new_campus_location,
         new_campus_address, new_campus_phone,
         new_campus_fax, new_campus_head);"');
      ELSIF options = 2 THEN
         DBMS_OUTPUT.PUT_LINE('Insert into Faculty');
```

```
    DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
    University.Insert_Faculty (new_fac_id,
    new_fac_name, new_fac_dean);"');
ELSIF options = 3 THEN
    DBMS_OUTPUT.PUT_LINE('Insert into School Nested
    Table');
    DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
    University.Insert_School(new_fac_id,
    new_school_id, new_school_name, new_school_head,
    new_prof_id, new_prof_name, new_prof_contact,
    new_prof_year);"');
ELSIF options = 4 THEN
    DBMS_OUTPUT.PUT_LINE('Insert into Department
    Nested Table');
    DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
    University.Insert_Department(new_fac_id,
    new_dept_id, new_dept_name, new_dept_head,
    new_prof_id, new_prof_name, new_prof_contact,
    new_prof_year);"');
ELSIF options = 5 THEN
    DBMS_OUTPUT.PUT_LINE('Insert into Research Centre
Nested Table');
    DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
    University.Insert_Research_Centre(new_fac_id,
    new_rc_id, new_rc_name, new_rc_head, new_unit1,
    new_unit2, new_unit3, new_unit4, new_unit5);"');
ELSIF options = 6 THEN
    DBMS_OUTPUT.PUT_LINE('Insert into Building');
    DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
    University.Insert_Building(new_building_id,
    new_building_name, new_building_location,
    new_building_level, new_campus_location,
    new_faculty_id);"');
ELSIF options = 7 THEN
    DBMS_OUTPUT.PUT_LINE('Insert into Office');
    DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
    University.Insert_Office(new_building_id,
    new_office_no, new_office_phone);"');
ELSIF options = 8 THEN
    DBMS_OUTPUT.PUT_LINE('Insert into Classroom');
    DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
    University.Insert_Classroom(new_building_id,
    new_class_no, new_class_capacity);"');
ELSIF options = 9 THEN
    DBMS_OUTPUT.PUT_LINE('Insert into Lab');
```

```
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Lab(new_building_id, new_lab_no,
      new_lab_capacity, new_lab_equipment_1,
      new_lab_equipment_2, new_lab_equipment_3,
      new_lab_equipment_4, new_lab_equipment_5);"');
   ELSIF options = 10 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Degree');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Degree(new_degree_id,
      new_degree_name, new_degree_length,
      new_degree_prerequisite, new_faculty_id);"');
   ELSIF options = 11 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Person');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Degree(new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location);"');
   ELSIF options = 12 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Staff');
      DBMS_OUTPUT.PUT_LINE('You have to insert from
      the child class');
   ELSIF options = 13 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Student');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Student(new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_year);"');
   ELSIF options = 14 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Admin');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Admin(new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_building_id,
      new_office_no, new_admin_title, new_comp_skills,
      new_office_skills);"');
   ELSIF options = 15 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Technician');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Technician(new_person_id,
      new_person_surname, new_person_fname,
```

```
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_building_id,
      new_office_no, new_tech_title,
      new_tech_skills);"');
   ELSIF options = 16 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Lecturer');
      DBMS_OUTPUT.PUT_LINE('You have to insert from
   the child class');
   ELSIF options = 17 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Senior
   Lecturer');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Senior_Lecturer(new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_building_id,
      new_office_no, new_area, new_no_phd,
      new_no_master, new_no_honours);"');
   ELSIF options = 18 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Associate
   Lecturer');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Associate_Lecturer(new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_building_id,
      new_office_no, new_area, new_no_honours,
      new_year_join);"');
   ELSIF options = 19 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Tutor');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Tutor(new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_building_id,
      new_office_no, new_year, new_no_hours,
      new_rate);"');
   ELSIF options = 20 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Enrolls_In');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Enrolls_In(new_pers_id,
      new_deg_id);"');
```

```
   ELSIF options = 21 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Subject');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Subject(new_subject_id,
      new_subject_name, new_subject_credit,
      new_subject_prereq, new_person_id);"');
   ELSIF options = 22 THEN
      DBMS_OUTPUT.PUT_LINE('Insert into Takes');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Insert_Takes(new_pers_id, new_subj_id,
      new_marks);"');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Wrong Option');
   END IF;
END Insertion;

— #1
PROCEDURE Insert_Campus(new_campus_location IN
   VARCHAR2, new_campus_address IN VARCHAR2,
   new_campus_phone IN VARCHAR2, new_campus_fax IN
   VARCHAR2,
   new_campus_head IN VARCHAR2) IS

BEGIN
   INSERT INTO Campus
   VALUES(new_campus_location, new_campus_address,
       new_campus_phone, new_campus_fax,
       new_campus_head);
END Insert_Campus;

— #2
PROCEDURE Insert_Faculty(new_fac_id IN VARCHAR2,
new_fac_name IN VARCHAR2,
   new_fac_dean IN VARCHAR2) IS

BEGIN
   INSERT INTO Faculty
   VALUES (new_fac_id, new_fac_name, new_fac_dean,
      School_Table_T(School_T(NULL,NULL,NULL,NULL)),
      Department_Table_T(Department_T(NULL,NULL,NULL,NULL)),
      Research_Centre_Table_T(Research_Centre_T(NULL,NULL,NULL,NULL)));
END Insert_Faculty;

— #3
PROCEDURE Insert_School(new_fac_id IN VARCHAR2,
new_school_id IN VARCHAR2,
```

```
   new_school_name IN VARCHAR2, new_school_head IN
   VARCHAR2, new_prof_id IN VARCHAR2, new_prof_name IN
   VARCHAR2, new_prof_contact IN VARCHAR2,
   new_prof_year IN NUMBER) IS

BEGIN
   INSERT INTO THE
      (SELECT a. school
       FROM Faculty a
       WHERE a.fac_id = new_fac_id)
   VALUES (new_school_id, new_school_name,
      new_school_head,
      Professors(Professor_T(new_prof_id, new_prof_name,
      new_prof_contact, new_prof_year)));
END Insert_School;

— #4
PROCEDURE Insert_Department(new_fac_id IN VARCHAR2,
new_dept_id IN VARCHAR2,
   new_dept_name IN VARCHAR2, new_dept_head IN
   VARCHAR2, new_prof_id IN VARCHAR2, new_prof_name IN
   VARCHAR2, new_prof_contact IN VARCHAR2,
   new_prof_year IN NUMBER) IS

BEGIN
   INSERT INTO THE
      (SELECT a.department
       FROM Faculty a
       WHERE a.fac_id = new_fac_id)
   VALUES (new_dept_id, new_dept_name, new_dept_head,
      Professors(Professor_T(new_prof_id, new_prof_name,
      new_prof_contact, new_prof_year)));
END Insert_Department;

— #5
PROCEDURE Insert_Research_Centre(new_fac_id IN
   VARCHAR2, new_rc_id IN VARCHAR2, new_rc_name IN
   VARCHAR2, new_rc_head IN VARCHAR2, new_unit1 IN
   VARCHAR2, new_unit2 IN VARCHAR2, new_unit3 IN
   VARCHAR2, new_unit4 IN VARCHAR2, new_unit5 IN
   VARCHAR2) IS

BEGIN
   INSERT INTO THE
      (SELECT a.research_centre
       FROM Faculty a
       WHERE a.fac_id = new_fac_id)
```

```
   VALUES (new_rc_id, new_rc_name, new_rc_head,
   Units(new_unit1, new_unit2, new_unit3, new_unit4,
   new_unit5));
END Insert_Research_Centre;

— #6
PROCEDURE Insert_Building(new_building_id IN VARCHAR2,
   new_building_name IN VARCHAR2, new_building_location
   IN VARCHAR2, new_building_level IN NUMBER,
   new_campus_location IN VARCHAR2, new_faculty_id IN
   VARCHAR2) IS

   campus_temp REF Campus_T;
   faculty_temp REF Faculty_T;

BEGIN
   SELECT REF(a) INTO campus_temp
   FROM Campus a
   WHERE a.campus_location = new_campus_location;

   SELECT REF(b) INTO faculty_temp
   FROM Faculty b
   WHERE b.fac_id = new_faculty_id;

   INSERT INTO Building
   VALUES(new_building_id, new_building_name,
      new_building_location, new_building_level,
      campus_temp, faculty_temp);
END Insert_Building;

— #7
PROCEDURE Insert_Office(new_building_id IN VARCHAR2,
   new_office_no IN VARCHAR2, new_office_phone IN
   VARCHAR2) IS

BEGIN
   INSERT INTO Office
   VALUES(new_building_id, new_office_no,
   new_office_phone);
END Insert_Office;

— #8
PROCEDURE Insert_Classroom(new_building_id IN VARCHAR2,
   new_class_no IN VARCHAR2, new_class_capacity IN
   NUMBER) IS
```

```
BEGIN
   INSERT INTO Classroom
   VALUES(new_building_id, new_class_no,
   new_class_capacity);
END Insert_Classroom;

— #9
PROCEDURE Insert_Lab(new_building_id IN VARCHAR2,
   new_lab_no IN VARCHAR2, new_lab_capacity IN NUMBER,
   new_lab_equipment_1 IN VARCHAR2, new_lab_equipment_2
   IN VARCHAR2, new_lab_equipment_3 IN VARCHAR2,
   new_lab_equipment_4 IN VARCHAR2, new_lab_equipment_5
   IN VARCHAR2) IS

BEGIN
   INSERT INTO Lab
   VALUES(new_building_id, new_lab_no,
      new_lab_capacity, Equipments(new_lab_equipment_1,
      new_lab_equipment_2, new_lab_equipment_3,
      new_lab_equipment_4, new_lab_equipment_5));
END Insert_Lab;

— #10
PROCEDURE Insert_Degree(new_degree_id IN VARCHAR2,
   new_degree_name IN VARCHAR2, new_degree_length IN
   VARCHAR2, new_degree_prerequisite IN VARCHAR2,
   new_faculty_id IN VARCHAR2) IS

   a_degree Degree_T :=
   Degree_T(NULL,NULL,NULL,NULL,NULL);

BEGIN
   a_degree.insert_degree (new_degree_id,
      new_degree_name, new_degree_length,
      new_degree_prerequisite, new_faculty_id);
END Insert_Degree;

— #11
PROCEDURE Insert_Person(new_person_id IN VARCHAR2,
   new_person_surname      IN VARCHAR2,
   new_person_fname IN VARCHAR2, new_person_title IN
   VARCHAR2, new_person_address IN VARCHAR2,
   new_person_phone IN VARCHAR2, new_person_postcode
   IN NUMBER, new_campus_location IN VARCHAR2) IS
```

```
   a_person Person_T :=
   Person_T(NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL);

BEGIN
   a_person.insert_person (new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location);
END Insert_Person;

— #13 (no procedure for Option 12)
PROCEDURE Insert_Student(new_person_id IN VARCHAR2,
   new_person_surname IN VARCHAR2, new_person_fname IN
   VARCHAR2, new_person_title IN VARCHAR2,
   new_person_address IN VARCHAR2, new_person_phone IN
   VARCHAR2, new_person_postcode IN NUMBER,
   new_campus_location IN VARCHAR2,
   new_year IN NUMBER) IS

   a_student Student_T := Student_T(NULL,NULL);

BEGIN
   a_student.insert_student (new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_year);
END Insert_Student;

— #14
PROCEDURE Insert_Admin(new_person_id IN VARCHAR2,
   new_person_surname IN VARCHAR2, new_person_fname IN
   VARCHAR2, new_person_title IN VARCHAR2,
   new_person_address IN VARCHAR2, new_person_phone IN
   VARCHAR2, new_person_postcode IN NUMBER,
   new_campus_location IN VARCHAR2, new_building_id IN
   VARCHAR2, new_office_no IN VARCHAR2,
   new_admin_title IN VARCHAR2, new_comp_skills IN
   VARCHAR2, new_office_skills IN VARCHAR2) IS

   an_admin Admin_T := Admin_T(NULL,NULL,NULL,NULL);

BEGIN
   an_admin.insert_admin (new_person_id,
      new_person_surname,  new_person_fname,
```

```
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_building_id,
      new_office_no, new_admin_title, new_comp_skills,
      new_office_skills);
END Insert_Admin;

— #15
PROCEDURE Insert_Technician(new_person_id IN VARCHAR2,
   new_person_surname IN VARCHAR2, new_person_fname IN
   VARCHAR2, new_person_title IN VARCHAR2,
   new_person_address IN VARCHAR2, new_person_phone IN
   VARCHAR2, new_person_postcode IN NUMBER,
   new_campus_location IN VARCHAR2, new_building_id IN
   VARCHAR2, new_office_no IN VARCHAR2, new_tech_title
   IN VARCHAR2, new_tech_skills IN VARCHAR2) IS

   a_technician Technician_T :=
      Technician_T(NULL,NULL,NULL);

BEGIN
   a_technician.insert_technician (new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_building_id,
      new_office_no, new_tech_title, new_tech_skills);
END Insert_Technician;

— #17 (no procedure for Option 16)
PROCEDURE Insert_Senior_Lecturer(new_person_id IN
   VARCHAR2, new_person_surname IN VARCHAR2,
   new_person_fname IN VARCHAR2, new_person_title IN
   VARCHAR2, new_person_address IN VARCHAR2,
   new_person_phone IN VARCHAR2, new_person_postcode
   IN NUMBER, new_campus_location IN VARCHAR2,
   new_building_id IN VARCHAR2, new_office_no IN
   VARCHAR2, new_area IN VARCHAR2, new_no_phd IN
   NUMBER, new_no_master IN NUMBER, new_no_honours IN
   NUMBER) IS

   a_senior_lect Senior_Lecturer_T :=
      Senior_Lecturer_T(NULL,NULL,NULL,NULL);

BEGIN
```

```
   a_senior_lect.insert_senior_lecturer (new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_building_id,
      new_office_no, new_area, new_no_phd,
      new_no_master, new_no_honours );
END Insert_Senior_Lecturer;

— #18
PROCEDURE Insert_Associate_Lecturer(new_person_id IN
   VARCHAR2, new_person_surname IN VARCHAR2,
   new_person_fname IN VARCHAR2, new_person_title IN
   VARCHAR2, new_person_address IN VARCHAR2,
   new_person_phone IN VARCHAR2, new_person_postcode
   IN NUMBER, new_campus_location IN VARCHAR2,
   new_building_id IN VARCHAR2, new_office_no IN
   VARCHAR2, new_area IN VARCHAR2, new_no_honours IN
   NUMBER, new_year_join IN NUMBER) IS

   a_associate_lect Associate_Lecturer_T :=
      Associate_Lecturer_T(NULL,NULL,NULL);

BEGIN
   a_associate_lect.insert_associate_lecturer
      (new_person_id, new_person_surname,
      new_person_fname, new_person_title,
      new_person_address, new_person_phone,
      new_person_postcode, new_campus_location,
      new_building_id, new_office_no, new_area,
      new_no_honours, new_year_join);
END Insert_Associate_Lecturer;

— #19
PROCEDURE Insert_Tutor(new_person_id IN VARCHAR2,
   new_person_surname IN VARCHAR2, new_person_fname IN
   VARCHAR2, new_person_title IN VARCHAR2,
   new_person_address IN VARCHAR2, new_person_phone IN
   VARCHAR2, new_person_postcode IN NUMBER,
   new_campus_location IN VARCHAR2, new_building_id IN
   VARCHAR2, new_office_no IN VARCHAR2, new_year IN
   NUMBER, new_no_hours IN NUMBER, new_rate IN
   NUMBER) IS

   a_tutor Tutor_T := Tutor_T(NULL,NULL,NULL);
```

```
BEGIN
   a_tutor.insert_tutor (new_person_id,
      new_person_surname, new_person_fname,
      new_person_title, new_person_address,
      new_person_phone, new_person_postcode,
      new_campus_location, new_building_id,
      new_office_no, new_year, new_no_hours, new_rate);
END Insert_Tutor;

— #20
PROCEDURE Insert_Enrolls_In(new_pers_id IN VARCHAR2,
   new_deg_id IN VARCHAR2) IS

   student_temp REF Student_T;
   degree_temp REF Degree_T;

BEGIN
    SELECT REF(a) INTO student_temp
    FROM Student a
    WHERE a.pers_id = new_pers_id;

    SELECT REF(b) INTO degree_temp
    FROM Degree b
    WHERE b.deg_id = new_deg_id;

    INSERT INTO Enrolls_In
    VALUES (student_temp, degree_temp);
END Insert_Enrolls_In;

— #21
PROCEDURE Insert_Subject(new_subject_id IN VARCHAR2,
   new_subject_name IN VARCHAR2, new_subject_credit IN
   VARCHAR2, new_subject_prereq IN VARCHAR2,
   new_person_id IN VARCHAR2) IS

   a_subject Subject_T :=
   Subject_T(NULL,NULL,NULL,NULL,NULL);

BEGIN
   a_subject.insert_subject (new_subject_id,
      new_subject_name, new_subject_credit,
      new_subject_prereq, new_person_id);
END Insert_Subject;

— #22
```

```
PROCEDURE Insert_Takes(new_pers_id IN VARCHAR2, new_subj_id
   IN VARCHAR2, new_marks IN NUMBER) IS

   student_temp REF Student_T;
   subject_temp REF Subject_T;

BEGIN
   SELECT REF(a) INTO student_temp
   FROM Student a
   WHERE a.pers_id = new_pers_id;

   SELECT REF(b) INTO subject_temp
   FROM Subject b
   WHERE b.subj_id = new_subj_id;

   INSERT INTO Takes
   VALUES (student_temp, subject_temp, new_marks);
END Insert_Takes;

PROCEDURE Deletion(options IN NUMBER) AS

BEGIN
   DBMS_OUTPUT.PUT_LINE('—————————————————————————————
   ————————');
   IF options = 1 THEN
      DBMS_OUTPUT.PUT_LINE('Delete from Campus');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Campus(deleted campus
      location);"');
   ELSIF options = 2 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Faculty');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Faculty(deleted fac id);"');
   ELSIF options = 3 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From School');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_School(deleted fac id);"');
   ELSIF options = 4 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Department');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Department(deleted fac id);"');
   ELSIF options = 5 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Research
   Centre');
```

```
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Reseach_Centre(deleted fac
      id);"');
   ELSIF options = 6 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Building');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Building(deleted building
      id);"');
   ELSIF options = 7 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Office');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Office(deleted building id,
      deleted office no);"');
   ELSIF options = 8 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Classroom');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Classroom(deleted building id,
      deleted class no);"');
   ELSIF options = 9 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Lab');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Lab(deleted building id,
      deleted lab no);"');
   ELSIF options = 10 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Degree');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Degree(deleted degree id);"');
   ELSIF options = 11 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Person');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Person(deleted person id);"');
   ELSIF options = 12 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Staff');
      DBMS_OUTPUT.PUT_LINE('You have to delete from
   the child classes');
   ELSIF options = 13 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Student');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Student(deleted person id);"');
   ELSIF options = 14 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Admin');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Admin(deleted person id);"');
   ELSIF options = 15 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Technician');
```

```
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Technician(deleted person
      id);"');
    ELSIF options = 16 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Lecturer');
      DBMS_OUTPUT.PUT_LINE('You have to delete from
    the child classes');
    ELSIF options = 17 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Senior
    Lecturer');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE University.
      Delete_Senior_Lecturer(deleted person id);"');
    ELSIF options = 18 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Associate
    Lecturer');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Associate_Lecturer(deleted
      person id);"');
    ELSIF options = 19 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Tutor');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Tutor(deleted person id);"');
    ELSIF options = 20 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Enrolls_In');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Enrolls_In(deleted person id,
      deleted degree id);"');
    ELSIF options = 21 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Subject');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Subject(deleted subject
      id);"');
    ELSIF options = 22 THEN
      DBMS_OUTPUT.PUT_LINE('Delete From Takes');
      DBMS_OUTPUT.PUT_LINE('Type "EXECUTE
      University.Delete_Enrolls_In(deleted person id,
      deleted subject id);"');
    ELSE
      DBMS_OUTPUT.PUT_LINE('Wrong Option');
    END IF;
  END Deletion;

  — #1
  PROCEDURE Delete_Campus(deleted_campus_location IN
    VARCHAR2) IS
```

```
BEGIN
   DELETE FROM Campus
   WHERE campus_location = deleted_campus_location;
END Delete_Campus;

— #2
PROCEDURE Delete_Faculty(deleted_fac_id IN VARCHAR2)
IS

BEGIN
   DELETE FROM Faculty
   WHERE fac_id = deleted_fac_id;
END Delete_Faculty;

— #3
PROCEDURE Delete_School(deleted_fac_id IN VARCHAR2) IS

BEGIN
   DELETE FROM THE
      (SELECT a.school
       FROM Faculty a
       WHERE a.fac_id = deleted_fac_id);
END Delete_School;

— #4
PROCEDURE Delete_Department(deleted_fac_id IN VARCHAR2)
IS

BEGIN
   DELETE FROM THE
      (SELECT a.department
       FROM Faculty a
       WHERE a.fac_id = deleted_fac_id);
END Delete_Department;

— #5
PROCEDURE Delete_Research_centre(deleted_fac_id IN
VARCHAR2) IS

BEGIN
   DELETE FROM THE
      (SELECT a.research_centre
       FROM Faculty a
       WHERE a.fac_id = deleted_fac_id);
END Delete_Research_centre;
```

```
— #6
PROCEDURE Delete_Building(deleted_building_id IN
VARCHAR2) IS

BEGIN
   DELETE FROM Building
   WHERE bld_id = deleted_building_id;
END Delete_Building;

— #7
PROCEDURE Delete_Office(deleted_building_id IN
   VARCHAR2, deleted_office_no IN VARCHAR2) IS

BEGIN
   DELETE FROM Office
   WHERE bld_id = deleted_building_id
   AND off_no = deleted_office_no;
END Delete_Office;

— #8
PROCEDURE Delete_Classroom(deleted_building_id IN
   VARCHAR2, deleted_class_no IN VARCHAR2) IS

BEGIN
   DELETE FROM Classroom
   WHERE bld_id = deleted_building_id
   AND class_no = deleted_class_no;
END Delete_Classroom;

— #9
PROCEDURE Delete_Lab(deleted_building_id IN VARCHAR2,
   deleted_lab_no IN VARCHAR2) IS

BEGIN
   DELETE FROM Lab
   WHERE bld_id = deleted_building_id
   AND lab_no = deleted_lab_no;
END Delete_Lab;

— #10
PROCEDURE Delete_Degree(deleted_degree_id IN VARCHAR2)
IS

a_degree Degree_T :=
Degree_T(NULL,NULL,NULL,NULL,NULL);
```

```
BEGIN
   a_degree.delete_degree(deleted_degree_id);
END Delete_Degree;

— #11
PROCEDURE Delete_Person(deleted_person_id IN VARCHAR2)
IS

a_person Person_T :=
   Person_T(NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL);

BEGIN
   a_person.delete_person(deleted_person_id);
END Delete_Person;

— #13 (no procedure #12)
PROCEDURE Delete_Student(deleted_person_id IN VARCHAR2)
IS

a_student Student_T := Student_T(NULL,NULL);

BEGIN
   a_student.delete_student(deleted_person_id);
END Delete_Student;

— #14
PROCEDURE Delete_Admin(deleted_person_id IN VARCHAR2)
IS

an_admin Admin_T := admin_T(NULL,NULL,NULL,NULL);

BEGIN
   an_admin.delete_admin(deleted_person_id);
END Delete_Admin;

— #15
PROCEDURE Delete_Technician(deleted_person_id IN
VARCHAR2) IS

a_technician Technician_T :=
technician_T(NULL,NULL,NULL);

BEGIN
   a_technician.delete_technician(deleted_person_id);
END Delete_Technician;
```

```
— #17 (no procedure #16)
PROCEDURE Delete_Senior_Lecturer(deleted_person_id IN
VARCHAR2) IS

a_senior_lecturer Senior_lecturer_T :=
   senior_lecturer_T(NULL,NULL,NULL,NULL);

BEGIN
   a_senior_lecturer.delete_senior_lecturer(deleted_person_id);
END Delete_Senior_Lecturer;

— #18
PROCEDURE Delete_Associate_Lecturer(deleted_person_id
IN VARCHAR2) IS

a_associate_lecturer Associate_lecturer_T :=
   associate_lecturer_T(NULL,NULL,NULL);

BEGIN
   a_associate_lecturer.delete_associate_lecturer(deleted_person_id);
END Delete_Associate_Lecturer;

— #19
PROCEDURE Delete_Tutor(deleted_person_id IN VARCHAR2)
AS

a_tutor Tutor_T := tutor_T(NULL,NULL,NULL);

BEGIN
   a_tutor.delete_tutor(deleted_person_id);
END Delete_Tutor;

— #20
PROCEDURE Delete_Enrolls_In(deleted_pers_id IN VARCHAR2,
   deleted_deg_id IN VARCHAR2) IS

BEGIN
   DELETE FROM Enrolls_In
   WHERE Enrolls_In.student IN
      (SELECT REF(a)
       FROM Student a
       WHERE a.pers_id = deleted_pers_id)
   AND Enrolls_In.degree IN
      (SELECT REF(b)
       FROM Degree b
       WHERE b.deg_id = deleted_deg_id);
```

```
   END Delete_Enrolls_In;

   — #21
   PROCEDURE Delete_Subject(deleted_subject_id IN
   VARCHAR2) IS

   a_subject Subject_T :=
   subject_T(NULL,NULL,NULL,NULL,NULL);

   BEGIN
      a_subject.delete_subject(deleted_subject_id);
   END Delete_Subject;

   — #22
   PROCEDURE Delete_Takes(deleted_pers_id IN VARCHAR2,
      deleted_subj_id IN VARCHAR2) IS

   BEGIN
      DELETE FROM Takes
      WHERE Takes.student IN
         (SELECT REF(a)
          FROM Student a
          WHERE a.pers_id = deleted_pers_id)
      AND Takes.subject IN
         (SELECT REF(b)
          FROM Subject b
          WHERE b.subj_id = deleted_subj_id);
   END Delete_Takes;

END University;
/
```

Running a package is very similar to running a stored procedure or function. We show an example of an execution and the results of the execution of this retailer application below. Notice that by using a package, we can add some lines to help users in using the application. The interaction is not as straightforward as in a programming language because the package and the procedure in SQL do not allow user input during their executions. Nevertheless, the line provided inside the procedures gives guidance to users on what procedures to use.

```
General Syntax:

EXECUTE <package name>.<object name>;
— The object is the stored procedures inside the
package.

EXECUTE University.Start_Program;
---------------------------------------------
For insertion, type "EXECUTE
University.Insertion("table_no");"
For deletion, type "EXECUTE University.Deletion
("table_no");
For retrieval, type EXECUTE University.Retrieval
("procedure no");
---------------------------------------------
To check the table no, type "EXECUTE
University.Table_Details;"
To check the procedure no, type "EXECUTE
University.Procedure_Details;"

PL/SQL procedure successfully completed.
```

Summary

In this chapter we have demonstrated a complete walk-through of a university case study. We have shown how we build each object type, table, and generic and user-defined member method. We then created the links between those types and tables, and instantiated the tables with some data. We have also shown how we can run user-defined queries to those created tables.

Chapter VIII

Retailer Case Study

In this chapter we will demonstrate the usage of development tools provided by Oracle™ Developer. The tools help users create forms, queries, projects, and other applications needed for practical purposes. Notice that we use Oracle™ Developer 6.0 for this chapter. Newer versions will have more features. Before demonstrating the usage of Oracle™ Developer, we will present another case study whose database has to be developed first.

Problem Description

National Ltd. is a major retail-chain company. Being the market leader in the retail industry, National has been urged to give extra attention to its database system. The excellence of the database system helps National in controlling its inventory better, in providing better service to the customer before and after transactions, and in maintaining its huge collection of internal organizational data.

Currently, National has six major retail companies under it. Three of them concentrate on food and daily goods, which are called Company Type 1, and the other three focus their business on clothing, housing furniture, and appliances, which are called Company Type 2. Figure 8.1 shows the details for each company.

Figure 8.1. Company table

Company					
Comp_ID	**Comp_Name**	**Comp_Address**	**Comp_Phone**	**Comp_Fax**	**Comp_Type**
1	OZ Buyer	20 Russel St. Sydney 2000	0298394000 0298394005 1800489000	0298398371	1
2	Goodies	50 Collins St. Melbourne 3000	0394255000 0394255005 1800900000	0394250005	1
3	Super Mart	6/1 George St. Brisbane 4000	0782349000 0782349005 1800521325	0782340005	1
4	Housemate	17/2 Vince St. Sydney 2000	0292000001 0292000002 1800023001	0292000000	2
5	Piglet	10 Bourke St Melbourne 3000	0398300000 0398300001 1800876001	0398300005	2
6	Liz and Neil	5 Lonsdale St Melbourne 3000	0398301000 0398301001 1800876005	0398601005	2

While the first three are Type 1 companies that are segmented based on the operational state, the last three are Type 2 companies that are segmented based on the income of the market. Among Company Type 1, OZ Buyer operates in NSW and ACT, Goodies covers VIC, SA, and TAS, while Super Mart has a very wide operation area from QLD, NT, and WA. Among the other three companies, Housemate is in the lower market, Piglet is in the middle market, and Liz and Neil is in the upper market. The data stored in this database is shown in Figure 8.2.

As the size of each company has expanded tremendously in the last 5 years, National has decided to have different shares listed for each company. The information about the shareholders is kept in the database system, which

Figure 8.2. Company_Type tables

Company_Type_1		
Comp_ID	**Type_Desc**	**Area**
1	Food and Daily Goods	NSW ACT
2	Food and Daily Goods	VIC SA TAS
3	Food and Daily Goods	QLD NT WA

Company_Type_2		
Comp_ID	**Type_Desc**	**Market**
4	Clothing, Furniture, and Appliances	Lower
5	Clothing, Furniture, and Appliances	Middle
6	Clothing, Furniture, and Appliances	Upper

includes each shareholder's ID, name, address, and telephone number. As each company has been listed separately, a shareholder can have shares in more than one company. Therefore, the database also keeps the record of the share amount that each shareholder has in each company. Examples of the data relating to shareholders are shown in Figure 8.3.

Each company has also stored information about its management personnel. This includes the management employee's ID, name, address, telephone number, and management type (whether he or she is a director or a manager). For each director, there is information about bonuses, while for each manager, there is information on the managerial type and yearly salary. A person can be a member of management for one, and only one, company. A person can be a director and a manager at the same time.

Each company has a large number of stores nationwide. Some of the basic data regarding the stores are shown in Figure 8.5.

Each store is divided into several departments. For example, in all OZ Buyer stores, there are delis, bakeries, drink sections, and so forth. For Housemate

Figure 8.3. Shareholders and Own_Shares tables

Shareholders			
Sholders_ID	**Sholders_Name**	**Sholders_Address**	**Sholders_Phone**
100	Judith Maxwell	40 Pinnacles Rd Melbourne 3000	0393450293
200	Ian Hobbes	2 Red Oak Ave Hobart 7000	0362231658

Own_Shares		
Sholders_ID	**Comp_ID**	**Share_Amount**
100	1	1000
100	2	250
200	1	2500

Figure 8.4. Management, Director, and Manager tables

Management				
Manag_ID	**Manag_Name**	**Manag_Address**	**Manag_Phone**	**Comp_ID**
1001	Kunio Takahashi	20 Avondale Cr. Darlinghurst 2010	0296101024	1
1002	Lucia Zanetti	5 Noel St Double Bay 2028	0290125846	1
1003	Stanley Mann	2/2 Ross St Mascot 2020	0295211110	1

Director	
Manag_ID	**Bonus**
1001	5%
1002	10%

Manager		
Manag_ID	**Manag_Type**	**Yearly_Salary**
1001	Information System	100,000
1003	Operational	85,000

Figure 8.5. Store tables

Store					
Store_ID	**Store_Location**	**Store_Address**	**Store_Phone**	**Store_Manage**	**Is_In**
OB1	Paramatta	4 Victoria Rd Paramatta 2797	02 9854 5876	Alice Green	1
OB2	Newcastle	15 University Dv Callaghan 2308	02 4589 5444	Rob Hayes	1
H1	Wollongong	5 Princess Hwy Woll. 2500	02 4256 8751	Elda Stiebel	4
P1	Crawley	110 Gordon St. Crawley 6009	08 9368 5123	Beth Jackson	5
P2	Melbourne	12 Bourke St Melbourne 3000	09 9458 5482	Yusuf Kamal	5

Figure 8.6. Department table

Department			
Store_ID	**Dept_ID**	**Dept_Name**	**Dept_Head**
OB1	1	Deli	Jared Dench
OB1	2	Bakery	Charlie Williams
OB1	3	Drinks	Ameer Singh
H1	1	Clothing	Lola Bing
H1	2	Furniture	Victor Mathewson
H1	3	Electrical	Raymond Chua

stores, there are departments of clothing, furniture, electrical appliances, and so forth. In general, department information consists of the department ID, name, and head (see Figure 8.6).

Considering the number of people who work in this retail company, the database for each employee is kept and linked to each store instead of each company. The Employee database includes the information about the employees' IDs, names, addresses, telephone numbers, the stores and departments they are working in, their account numbers, tax file numbers, and their types of employment.

There are three types of employment at National Ltd.: full time, part time, and casual. For full-time employees, the data about annual salaries and bonuses have to be recorded. For part-time employees, the data about weekly wages have to be recorded. Finally, for casual employees, the additional information is their hourly wages. Figure 8.7 shows the table sample.

For inventory control, the database system covers the items database and includes the information of the item ID, name, description, cost, selling price, stock amount, and finally the information about the item distributor. According to the policy of the company, a specific item can be bought only from one

Figure 8.7. Employee tables

Employee				
Emp_ID	**Emp_Name**	**Emp_Address**	**Emp_Phone**	**Emp_Type**
OB1-01	Glenda Row	10/1 Harold St. Thornbury 3125	0395854523	Full Time
OB1-02	Ruben Boestch	5 Greenwood Dv Bundoora 3083	0402251657	Full Time
OB1-03	Lily Hui	6 Mornane St Preston 3203	0411528876	Part Time
OB1-04	David Tran	12 Gillies St Fairfield 3175	0398575854	Part Time
OB1-05	Debbie Bradsord	740 High St Northcote 3185	0399587410	Casual
OB1-06	Turi Riswanant	3/2 George St Reservoir 3158	0403528587	Null

Employee			
Emp_Account_No	**Emp_TFN**	**Work_In**	**Dept_ID**
2568-548-586	081253654	OB1	1
2568-587-875	084568789	OB1	1
1525-288-888	084565896	OB1	1
1259-986-458	089658754	OB1	2
3366-000-120	098658423	OB1	3
3366-895-452	098547785	OB1	3

Full_Time		
Emp_ID	**Annual_Wage**	**Emp_Bonus**
OB1-01	30,000	2,000
OB1-02	28,000	1,750

Part_Time	
Emp_ID	**Weekly_Wage**
OB1-03	400
OB1-04	500

Casual	
Emp_ID	**Hourly_Wage**
OB1-05	13

Figure 8.8. Maker and Item tables

Maker			
Maker_ID	**Maker_Name**	**Maker_Address**	**Maker_Phone**
M1	Smiths	15 Princess Hwy Sydney 2000	1800157856
M2	Homemade	450 Light Ave Albury 2780	0245245263

Item					
Item_ID	**Item_Name**	**Item_Desc**	**Item_Cost**	**Item_Price**	**Made_By**
I-1001	Crisp Original	Potato Chips 250 gr	2.00	3.10	M1
I-1002	Crisp Cheese	Potato Chips 250 gr	2.00	3.10	M1
I-1051	Cheese Bun	Homemade 500 gr	3.00	3.25	M2

maker, but one maker can sell many items to National. Figure 8.8 shows the Maker and Item tables.

As one item can be sold in many stores and one store can sell many items, we need another table to store the relationship called the Available_In table (see Figure 8.9). In this table we also store the information on the stock available at any given time.

For better service to the customer, National keeps information about the main customers who are registered and have membership cards. By using the card

Figure 8.9. Available_In table

Available_In		
Item_ID	**Store_ID**	**Item_Stock**
I-1001	OB1	5,000
I-1002	OB1	5,000
I-1051	OB1	200

Figure 8.10. Customer table

Customer						
Cust_ID	**Cust_Name**	**Cust_Address**	**Cust_Phone**	**Cust_ Gender**	**Cust_DOB**	**Bonus _Point**
C1001	Sally Lange	14 Milky Way St Melbourne 3000	0395486542	F	01-Mar-1970	100
C1002	Raylene Roberts	1/1 Howard St Box Hill 3128	0398306360	F	23-Feb-1950	125

Figure 8.11. Transaction table

Transaction				
Trans_ID	**Trans_Date**	**Cust_ID**	**Item_ID**	**Quantity**
1602027891	16-Feb-2002	C1002	I-1001	5
1802021009	18-Feb-2002	C1001	I-1002	4
1802021010	18-Feb-2002	C1002	I-1001	5
1802021049	18-Feb-2002	C1002	I-1003	10

while shopping, the customers will amass points that can be redeemed for an annual prize. Figure 8.10 shows the example of the Customer table.

These Customer and Item tables are linked to another table named Transaction. We need this table to analyze the items that a particular customer always buys. It can be very useful for marketing strategy. Note that the transaction is differentiated based on the date, customer, and item.

The information-system department uses an ORDB for its database system and stores the data in classes. Some methods, mainly generic methods, are implemented as member procedures. They are usually methods for insertion and deletion. Not every class needs member procedures. Only those classes that frequently undergo insertion and deletion will need these generic member procedures.

Classes that need generic member methods are Store_T and its part class, Department_T. The Employee_T class and all its children also need these methods because in this business, there are frequent ups and downs that urge the company to have a flexible number of employees, therefore insertion and deletion will be frequent. The next class that needs member methods is Shareholders_T. Notice that the relation between the shareholders and the companies they have invested in is equally important. For this purpose, we might need a regular stored procedure. We also need member procedures for the Item_T and Customer_T classes as they are the two most frequently accessed databases in the retail industry. Finally, regular stored procedures will be needed for accessing the tables that emerge from the relationship between Item_T and Store_T.

Besides generic methods, there are some user-defined queries that are frequently made for this database. These queries can also be implemented as member methods. The list of these methods is shown below.

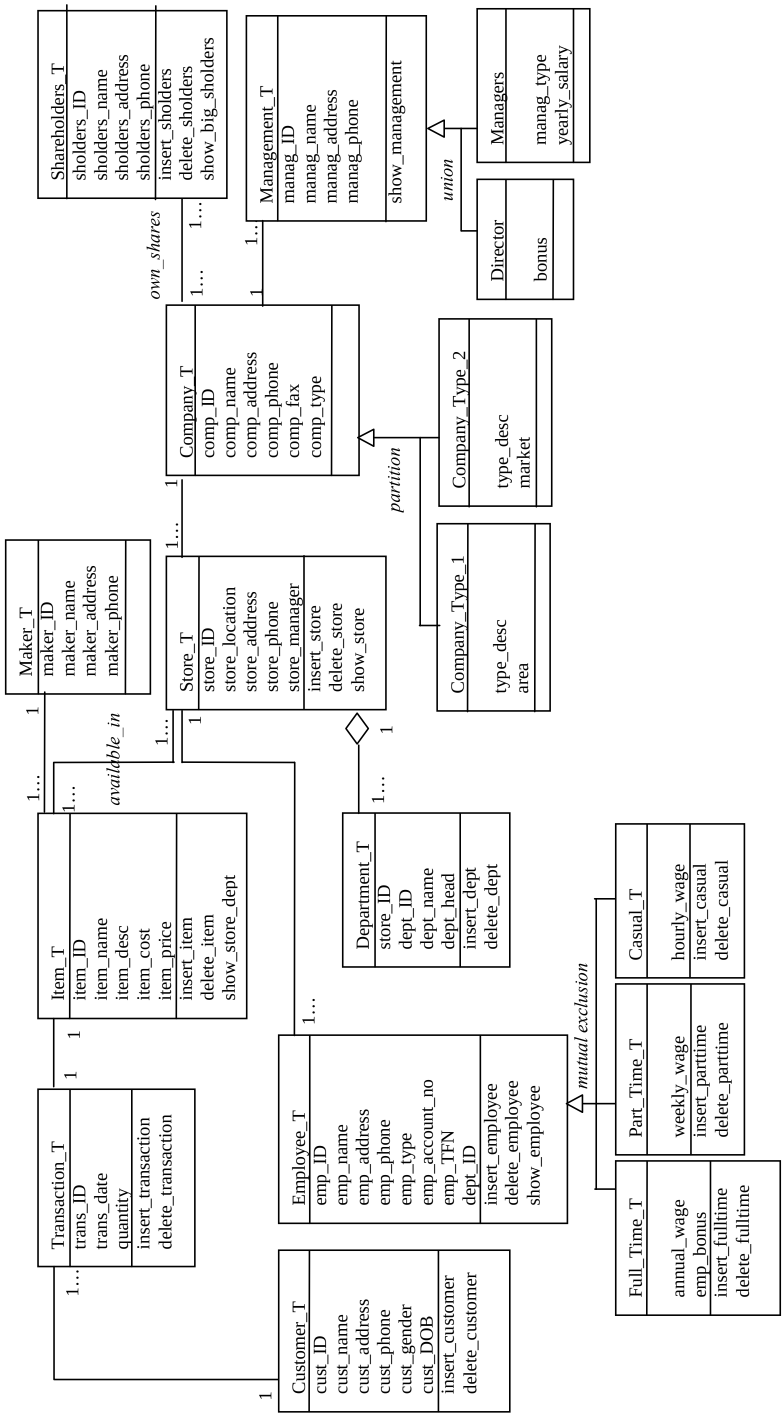
- Method to show the details of an certain store, which will be implemented as a member procedure of the Store_T class
- Method to show the details of shareholders if they have more than 1,000 shares in a given company type. This procedure will be implemented in the Shareholders_T class.
- Method to show the names of management employees and details including the companies they are in, ordered by the name of the company. This member procedure will be implemented in the Management_T class.
- Method to show the details of an employee, which will be implemented in Employee_T class

Finally, we will require a stored procedure to show the item details and the sum of each item that is bought by a specific customer gender in a residential suburb. It will also need to determine the maximum age of the customers that buy those items.

Problem Solution

The solution to the problem described in the previous section will be provided in this section. The first thing to be done in solving this problem is to design the database. We provide the design in an object-oriented diagram (see Figure 8.12). Note that the diagram does not indicate the number of tables we will need

Figure 8.12. Object-oriented diagram of National Ltd.



to create. We also have to consider the cardinality of the relationships before determining the number of tables needed.

To illustrate a clearer step-by-step development, the solution will be implemented for one class at a time. It starts with the object creation, then moves to table creation, and then, where applicable, to method creation. Note that the table that results from the many-to-many relationship will be implemented along with implementation of the second class.

Company_T Class and the Subclasses

Below we show the implementation of the Company_T class and the table derived from the class, along with its subclasses. There is no member method needed in this class, therefore, the data will be inserted using the regular insert statement.

Relational Schemas

```
Company (comp_ID, comp_name, comp_address,
comp_phone,
        comp_fax, comp_type)
```

Class and Table Declaration

```
CREATE OR REPLACE TYPE Contacts AS VARRAY(3) OF
VARCHAR2(12)
/

CREATE OR REPLACE TYPE Company_T AS OBJECT
   (comp_id      VARCHAR2(10),
    comp_name    VARCHAR2(20),
    comp_address VARCHAR2(50),
    comp_phone   Contacts,
    comp_fax     VARCHAR2(10),
    comp_type    NUMBER) NOT FINAL
/

CREATE TABLE Company OF Company_T
   (comp_id NOT NULL,
    comp_type CHECK (comp_type IN (1, 2)),
    PRIMARY KEY (comp_id));
```

```
CREATE OR REPLACE TYPE Company_Type_1_T UNDER
Company_T
   (type_desc     VARCHAR2(40),
    area          VARCHAR2(20))
/

CREATE OR REPLACE TYPE Company_Type_2_T UNDER
   Company_T
   (type_desc     VARCHAR2(40),
    market        VARCHAR2(20))
/
```

Shareholders_T Class and Own_Shares Table

This section shows the implementation of Shareholders_T, the Shareholders table, and the Own_Shares table, which is derived from the many-to-many relationship between the tables Company and Shareholders. As it is a regular table, we cannot have a member method for the Own_Shares table. Instead, we need regular stored procedures for insertion and deletion.

Besides generic methods, there is also a user-defined method in the Shareholders_T class to show the details of the shareholders who have more than 1,000 shares, including the name of the company they have invested in given the type of the company as an input parameter.

```
Relational Schemas
   — We do not use the primary-key and foreign-key
   concept in
   — the Own_Shares table. Instead, we are using ref as
   object references.

   Shareholders (sholders_ID, sholders_name,
                 sholders_address, sholders_phone)
   Own_Shares (shareholders, company, share_amount)

Class, Table, and Method Declaration

   CREATE OR REPLACE TYPE Shareholders_T AS OBJECT
      (sholders_id        VARCHAR2(10),
       sholders_name      VARCHAR2(20),
       sholders_address   VARCHAR2(50),
       sholders_phone     VARCHAR2(10),
```

```
      MEMBER PROCEDURE insert_sholders(
       new_sholders_id IN VARCHAR2,
       new_sholders_name IN VARCHAR2,
       new_sholders_address IN VARCHAR2,
       new_sholders_phone IN VARCHAR2),

      MEMBER PROCEDURE delete_sholders,

      MEMBER PROCEDURE show_big_sholders(
       new_comp_type IN NUMBER))
/

CREATE TABLE Shareholders OF Shareholders_T
   (sholders_id NOT NULL,
    PRIMARY KEY (sholders_id));

CREATE TABLE Own_Shares
   (shareholders REF Shareholders_T,
    company REF Company_T,
    share_amount NUMBER);
```

Method Implementation

```
CREATE OR REPLACE TYPE BODY Shareholders_T AS

   MEMBER PROCEDURE insert_sholders(
      new_sholders_id IN VARCHAR2,
      new_sholders_name IN VARCHAR2,
      new_sholders_address IN VARCHAR2,
      new_sholders_phone IN VARCHAR2) IS

   BEGIN
      INSERT INTO Shareholders
      VALUES (new_sholders_id, new_sholders_name,
              new_sholders_address, new_sholders_phone);
   END insert_sholders;

   MEMBER PROCEDURE delete_sholders IS

   BEGIN
      DELETE FROM Own_Shares
      WHERE Own_Shares.shareholders IN
         (SELECT REF(a)
          FROM Shareholders a
          WHERE a.sholders_id = self.sholders_id);
      DELETE FROM Shareholders
```

```
      WHERE sholders_id = self.sholders_id;
   END delete_sholders;

   MEMBER PROCEDURE show_big_sholders(
      new_comp_type IN NUMBER) IS

      v_sholders_name
      Shareholders.sholders_name%TYPE;
      v_sholders_address
      Shareholders.sholders_address%TYPE;
      v_sholders_phone
      Shareholders.sholders_phone%TYPE;

   BEGIN

      SELECT a.sholders_name, a.sholders_address,
      a.sholders_phone
      INTO v_sholders_name, v_sholders_address,
      v_sholders_phone
      FROM Shareholders a, Company b, Own_Shares c
      WHERE c.shareholders = REF(a)
      AND c.company = REF(b)
      AND b.comp_type = new_comp_type
      AND c.share_amount > 1000
      AND a.sholders_id = self.sholders_id;

      DBMS_OUTPUT.PUT_LINE
      ('Name'||'  '||'Address'||'  '||'Phone');
      DBMS_OUTPUT.PUT_LINE
      ('———————————————————');
      DBMS_OUTPUT.PUT_LINE
      (v_ sholders_name||'  '|| v_
      sholders_address|| '  '||
          v_ sholders_phone);
   END show_big_sholders;
END;
/

— The following are stored procedures for the Own_Shares
table
— for insertion and deletion.

CREATE OR REPLACE PROCEDURE Insert_Own_Shares(
   new_sholders_id IN VARCHAR2,
   new_comp_id IN VARCHAR2,
   new_share_amount IN NUMBER) AS
```

```
    sholders_temp REF Shareholders_T;
    comp_temp REF Company_T;

    BEGIN
       SELECT REF(a) INTO sholders_temp
       FROM Shareholders a
       WHERE a.sholders_id = new_sholders_id;

       SELECT REF(b) INTO comp_temp
       FROM Company b
       WHERE b.comp_id = new_comp_id;

       INSERT INTO Own_Shares
       VALUES (sholders_temp, comp_temp,
               new_share_amount);
    END Insert_Own_Shares;
/

CREATE OR REPLACE PROCEDURE Delete_Own_Shares(
    deleted_sholders_id IN VARCHAR2,
    deleted_comp_id IN VARCHAR2) AS

    BEGIN
       DELETE FROM Own_Shares
       WHERE Own_Shares.shareholders IN
              (SELECT REF(a)
               FROM Shareholders a
               WHERE a.sholders_id = deleted_sholders_id)
       AND Own_Shares.company IN
              (SELECT REF(b)
               FROM Company b
               WHERE b.comp_id = deleted_comp_id);
    END Delete_Own_Shares;
/
```

Management_T Class and the Subclasses

Next, we show the implementation of the Management_T class and the table derived from the class, along with its subclasses. As the frequency of insertion and deletion transactions for this class is considerably low, we do not need to implement the generic methods inside the class. However, we still need to implement a user-defined method show_management to display the management staff who has two roles: both as director and as manager.

Relational Schemas

```
— Note that manag_id in the subclasses is both a
primary key and
— foreign key at the same time. To implement the
relationship of
— Management with Company, we will use the object
reference of work_in.

Management (manag_ID, manag_name, manag_address,
manag_phone, work_in)
Directors (manag_ID, bonus)
Managers (manag_ID, manag_type, yearly_salary)
```

Class, Table, and Method Declaration

```
CREATE OR REPLACE TYPE Management_T AS OBJECT
   (manag_id      VARCHAR2(10),
    manag_name    VARCHAR2(20),
    manag_address VARCHAR2(50),
    manag_phone   VARCHAR2(10),
    work_in REF Company_T,

    MEMBER PROCEDURE show_management) NOT FINAL
/

CREATE TABLE Management OF Management_T
   (manag_id NOT NULL,
    PRIMARY KEY (manag_id));

CREATE OR REPLACE TYPE Directors_T UNDER Management_T
   (bonus         NUMBER)
/

CREATE TABLE Directors OF Directors_T
   (manag_id NOT NULL,
    PRIMARY KEY (manag_id));

CREATE OR REPLACE TYPE Managers_T UNDER Management_T
   (manag_type    VARCHAR2(20),
    yearly_salary NUMBER)
/

CREATE TABLE Managers OF Managers_T
   (manag_id NOT NULL,
    PRIMARY KEY (manag_id));
```

```
Method Implementation

   CREATE OR REPLACE TYPE BODY Management_T AS

      MEMBER PROCEDURE show_management IS

      CURSOR c_management IS
         SELECT a.manag_name, b.comp_name
         FROM Management a, Company b
         WHERE a.work_in = REF(b)
         AND a.manag_id = self.manag_id
         AND a.manag_id IN
            (SELECT manag_id FROM Directors)
         AND a.manag_id IN
            (SELECT manag_id FROM Managers)
         ORDER BY b.comp_name;

      BEGIN
         DBMS_OUTPUT.PUT_LINE
         ('Company Name'||' '||'Management Name');
         DBMS_OUTPUT.PUT_LINE
         ('——————————————————————');
         FOR v_management IN c_management LOOP
            DBMS_OUTPUT.PUT_LINE
            (v_management.comp_name||' '||
             v_management.manag_name);
         END LOOP;
      END show_management;
   END;
   /
```

Store_T Class and the Department_T Part Class

To store the aggregation relationship between Store_T and Department_T, we use the clustering technique. We need generic methods for both whole and part classes. In addition, we will need a user-defined method, show_stores in the Store_T class, to display the store details of a particular company in a particular location.

```
Relational Schemas
   — Note that there are two primary keys in the
   part class Department_T,
```

```
— and one of them (store_ID) is also a foreign
key to the whole class
— Store_T. The relationship between Stores and
Company is made using
— the object reference is_in.

Stores (store_ID, store_location, store_address,
        store_phone, store_manager, is_in)
Department (store_ID, dept_ID, dept_name,
dept_head)
```

Class, Table, and Method Declaration

```
CREATE OR REPLACE TYPE Store_T AS OBJECT
   (store_id           VARCHAR2(10),
    store_location     VARCHAR2(20),
    store_address      VARCHAR2(50),
    store_phone        VARCHAR2(10),
    store_manager      VARCHAR2(20),
    is_in REF Company_T,

    MEMBER PROCEDURE insert_store(
     new_store_id IN VARCHAR2,
     new_store_location IN VARCHAR2,
     new_store_address IN VARCHAR2,
     new_store_phone IN VARCHAR2,
     new_store_manager IN VARCHAR2,
     new_comp_id IN VARCHAR2),

    MEMBER PROCEDURE delete_store,
    MEMBER PROCEDURE show_store)
/

CREATE CLUSTER Store_Cluster
   (store_id      VARCHAR2(10));

CREATE TABLE Store OF Store_T
   (store_id NOT NULL,
    PRIMARY KEY (store_id))
   CLUSTER Store_Cluster(store_id);

CREATE OR REPLACE TYPE Department_T AS OBJECT
   (store_id      VARCHAR2(10),
    dept_id       VARCHAR2(10),
    dept_name     VARCHAR2(20),
    dept_head     VARCHAR2(20),
```

```
      MEMBER PROCEDURE insert_dept(
        new_store_id IN VARCHAR2,
        new_dept_id IN VARCHAR2,
        new_dept_name IN VARCHAR2,
        new_dept_head IN VARCHAR2),

      MEMBER PROCEDURE delete_dept)
/

CREATE TABLE Department OF Department_T
   (store_id NOT NULL,
    dept_id NOT NULL,
    PRIMARY KEY (store_id, dept_id),
    FOREIGN KEY (store_id)
      REFERENCES Store(store_id))
   CLUSTER Store_Cluster(store_id);

CREATE INDEX Store_Cluster_Index
   ON CLUSTER Store_Cluster;
```

Method Implementation

```
CREATE OR REPLACE TYPE BODY Store_T AS

   MEMBER PROCEDURE insert_store(
      new_store_id IN VARCHAR2,
      new_store_location IN VARCHAR2,
      new_store_address IN VARCHAR2,
      new_store_phone IN VARCHAR2,
      new_store_manager IN VARCHAR2,
      new_comp_id IN VARCHAR2) IS

      comp_temp REF Company_T;

   BEGIN
      SELECT REF(a) INTO comp_temp
      FROM Company a
      WHERE a.comp_id = new_comp_id;

      INSERT INTO Store
      VALUES (new_store_id, new_store_location,
              new_store_address, new_store_phone,
              new_store_manager, comp_temp);
   END insert_store;

   MEMBER PROCEDURE delete_store IS
```

```
   BEGIN
      — If a store is deleted, the data from table
      Available_In
      — regarding that particular store has to be
      removed as well.
      — Also note the table Available_In has to exist
      first.

      DELETE FROM Available_In
      WHERE Available_In.store IN
         (SELECT REF(a) FROM Store a
          WHERE a.store_id = self.store_id);

      DELETE FROM Employee
      WHERE store_id = self.store_id;

      DELETE FROM Store
      WHERE store_id = self.store_id;
   END delete_store;

   MEMBER PROCEDURE show_store IS

   BEGIN
      DBMS_OUTPUT.PUT_LINE
      ('Store Address'||' '||'Store Phone'||'
      '||'Store Manager');
      DBMS_OUTPUT.PUT_LINE
      ('——————————————————————————');
      DBMS_OUTPUT.PUT_LINE
      (self.store_address||' '||self.store_phone|| '
      '|| self.store_manager);
   END show_store;

END;
/

CREATE OR REPLACE TYPE BODY Department_T AS

   MEMBER PROCEDURE insert_dept(
      new_store_id IN VARCHAR2,
      new_dept_id IN VARCHAR2,
      new_dept_name IN VARCHAR2,
      new_dept_head IN VARCHAR2) IS
```

```
      BEGIN
         INSERT INTO Department
         VALUES (new_store_id, new_dept_id, new_dept_name,
         new_dept_head);
      END insert_dept;

      MEMBER PROCEDURE delete_dept IS

      BEGIN
         DELETE FROM Department
         WHERE store_id = self.store_id
         AND dept_id = self.dept_id;
      END delete_dept;

   END;
   /
```

Employee_T Class and the Subclasses

Next, we show the implementation of the Employee_T class and its table, along with the subclasses. As the type of inheritance is not mentioned, we will assume that the type is a mutual-exclusion inheritance. In other words, an employee has to be a member of only one subclass or none.

We need generic methods for the superclass and the subclasses. For insertion to subclasses, we will need to insert to the superclass first. The same applies for deletion. The data in the subclasses will be deleted automatically due to the referential integrity constraint. In addition, there is a user-defined method show_employee to display the details of a particular employee type that works in a particular store.

```
Relational Schemas
   — Note there is a reference work_in to the Store
   table.

   Employee (emp_ID, emp_name, emp_address, emp_phone,
            emp_type, emp_account_no, emp_TFN,
   work_in, dept_ID)

   Class, Table, and Method Declaration

   CREATE OR REPLACE TYPE Employee_T AS OBJECT
      (emp_id              VARCHAR2(10),
```

```
    emp_name          VARCHAR2(20),
    emp_address       VARCHAR2(50),
    emp_phone         VARCHAR2(10),
    emp_type          VARCHAR2(20),
    emp_account_no    VARCHAR2(15),
    emp_tfn           VARCHAR2(15),
    work_in REF Store_T,
    dept_id           VARCHAR2(10),

    MEMBER PROCEDURE insert_employee(
     new_emp_id IN VARCHAR2,
     new_emp_name IN VARCHAR2,
     new_emp_address IN VARCHAR2,
     new_emp_phone IN VARCHAR2,
     new_emp_account_no IN VARCHAR2,
     new_emp_tfn IN VARCHAR2,
     new_store_id IN VARCHAR2,
     new_dept_id IN VARCHAR2),

    MEMBER PROCEDURE delete_employee,
    MEMBER PROCEDURE show_employee) NOT FINAL
/

CREATE TABLE Employee OF Employee_T
   (emp_id NOT NULL,
    emp_type CHECK (emp_type IN
      ('Full Time', 'Part Time', 'Casual', NULL)),
    PRIMARY KEY (emp_id));

CREATE OR REPLACE TYPE Full_Time_T UNDER
Employee_T
   (annual_wage   NUMBER,
    emp_bonus     NUMBER,

    MEMBER PROCEDURE insert_fulltime(
     new_emp_id IN VARCHAR2,
     new_emp_name IN VARCHAR2,
     new_emp_address IN VARCHAR2,
     new_emp_phone IN VARCHAR2,
     new_emp_account_no IN VARCHAR2,
     new_emp_tfn IN VARCHAR2,
     new_store_id IN VARCHAR2,
     new_dept_id IN VARCHAR2,
     new_annual_wage IN NUMBER,
     new_emp_bonus IN NUMBER),
```

```
      MEMBER PROCEDURE delete_fulltime)
/

CREATE OR REPLACE TYPE Part_Time_T UNDER
Employee_T
   (weekly_wage   NUMBER,

      MEMBER PROCEDURE insert_parttime(
        new_emp_id IN VARCHAR2,
        new_emp_name IN VARCHAR2,
        new_emp_address IN VARCHAR2,
        new_emp_phone IN VARCHAR2,
        new_emp_account_no IN VARCHAR2,
        new_emp_tfn IN VARCHAR2,
        new_store_id IN VARCHAR2,
        new_dept_id IN VARCHAR2,
        new_weekly_wage IN NUMBER),

      MEMBER PROCEDURE delete_parttime)
/

CREATE OR REPLACE TYPE Casual_T UNDER Employee_T
   (hourly_wage   NUMBER,

      MEMBER PROCEDURE insert_casual(
        new_emp_id IN VARCHAR2,
        new_emp_name IN VARCHAR2,
        new_emp_address IN VARCHAR2,
        new_emp_phone IN VARCHAR2,
        new_emp_account_no IN VARCHAR2,
        new_emp_tfn IN VARCHAR2,
        new_store_id IN VARCHAR2,
        new_dept_id IN VARCHAR2,
        new_hourly_wage IN NUMBER),

      MEMBER PROCEDURE delete_casual)
/
```

Method Implementation

```
CREATE OR REPLACE TYPE BODY Employee_T AS

   MEMBER PROCEDURE insert_employee(
      new_emp_id IN VARCHAR2,
      new_emp_name IN VARCHAR2,
      new_emp_address IN VARCHAR2,
      new_emp_phone IN VARCHAR2,
```

```
      new_emp_account_no IN VARCHAR2,
      new_emp_tfn IN VARCHAR2,
      new_store_id IN VARCHAR2,
      new_dept_id IN VARCHAR2) IS

      store_temp REF Store_T;

   BEGIN
      SELECT REF(a) INTO store_temp
      FROM Store a
      WHERE a.store_id = new_store_id;

      INSERT INTO Employee
      VALUES (new_emp_id, new_emp_name, new_emp_address,
              new_emp_phone, NULL, new_emp_account_no,
              new_emp_tfn, store_temp, new_dept_id);
   END insert_employee;

   MEMBER PROCEDURE delete_employee IS

   BEGIN
      DELETE FROM Employee
      WHERE Employee.emp_id = self.emp_id;
   END delete_employee;

   MEMBER PROCEDURE show_employee IS

   BEGIN
      DBMS_OUTPUT.PUT_LINE
      ('Name'||' '||'Address'||' '||'Phone'||'
      '||'Emp Type'||' '||'Account No'||'
      '||'TFN'||' '|| 'Department');
      DBMS_OUTPUT.PUT_LINE
      ('—————————————————————————————————————————————
      ———————————');
      DBMS_OUTPUT.PUT_LINE
      (self.emp_name ||' '||self. emp_address||'
      '||self.emp_phone|| ' '||self.emp_type|| '
      '||self.emp_account_no|| ' '||self.emp_tfn|| '
      '||self.dept_id);
      END LOOP;
   END show_employee;

END;
/

CREATE OR REPLACE TYPE BODY Full_Time_T AS
```

```
    MEMBER PROCEDURE insert_fulltime(
       new_emp_id IN VARCHAR2,
       new_emp_name IN VARCHAR2,
       new_emp_address IN VARCHAR2,
       new_emp_phone IN VARCHAR2,
       new_emp_account_no IN VARCHAR2,
       new_emp_tfn IN VARCHAR2,
       new_store_id IN VARCHAR2,
       new_dept_id IN VARCHAR2,
       new_annual_wage IN NUMBER,
       new_emp_bonus IN NUMBER) IS

       store_temp REF Store_T;

    BEGIN
       SELECT REF(a) INTO store_temp
       FROM Store a
       WHERE a.store_id = new_store_id;

       INSERT INTO Employee
       VALUES (Full_Time_T(new_emp_id, new_emp_name,
       new_emp_address,
                new_emp_phone, 'Full Time',
             new_emp_account_no,
               new_emp_tfn, store_temp, new_dept_id,
             new_annual_wage,
               new_emp_bonus);
    END insert_fulltime;

    MEMBER PROCEDURE delete_fulltime IS

    BEGIN
       DELETE FROM Employee
       WHERE Employee.emp_id = self.emp_id;
    END delete_fulltime;

 END;
 /

 CREATE OR REPLACE TYPE BODY Part_Time_T AS

    MEMBER PROCEDURE insert_parttime(
       new_emp_id IN VARCHAR2,
       new_emp_name IN VARCHAR2,
       new_emp_address IN VARCHAR2,
       new_emp_phone IN VARCHAR2,
```

```
      new_emp_account_no IN VARCHAR2,
      new_emp_tfn IN VARCHAR2,
      new_store_id IN VARCHAR2,
      new_dept_id IN VARCHAR2,
      new_weekly_wage IN NUMBER) IS

      store_temp REF Store_T;

   BEGIN
      SELECT REF(a) INTO store_temp
      FROM Store a
      WHERE a.store_id = new_store_id;

      INSERT INTO Employee
      VALUES (Part_Time_T(new_emp_id, new_emp_name,
      new_emp_address,
            new_emp_phone, 'Part Time',
         new_emp_account_no,
           new_emp_tfn, store_temp, new_dept_id,
         new_weekly_wage));
   END insert_parttime;

   MEMBER PROCEDURE delete_parttime IS

   BEGIN
      DELETE FROM Employee
      WHERE Employee.emp_id = self.emp_id;
   END delete_parttime;
END;
/

CREATE OR REPLACE TYPE BODY Casual_T AS

   MEMBER PROCEDURE insert_casual(
      new_emp_id IN VARCHAR2,
      new_emp_name IN VARCHAR2,
      new_emp_address IN VARCHAR2,
      new_emp_phone IN VARCHAR2,
      new_emp_account_no IN VARCHAR2,
      new_emp_tfn IN VARCHAR2,
      new_store_id IN VARCHAR2,
      new_dept_id IN VARCHAR2,
      new_hourly_wage IN NUMBER) IS

      store_temp REF Store_T;

   BEGIN
```

```
        SELECT REF(a) INTO store_temp
        FROM Store a
        WHERE a.store_id = new_store_id;

        INSERT INTO Employee
        VALUES (Casual_T(new_emp_id, new_emp_name,
        new_emp_address,
                    new_emp_phone,      'Casual',
              new_emp_account_no, new_emp_tfn,
               store_temp, new_dept_id, new_hourly_wage);
    END insert_Casual;

    MEMBER PROCEDURE delete_casual IS

    BEGIN
        DELETE FROM Employee
        WHERE Employee.emp_id = self.emp_id;
    END delete_casual;

END;
/
```

Maker_T Class

The Maker_T class and its table have to be created first before the Item table because the Item_T class will have an object reference to Maker_T. We also need a user-defined method show_maker to display the maker details given an item ID.

```
Relational Schemas
   Maker (maker_ID, maker_name, maker_address,
   maker_phone)

   Class, Table, and Method Declaration

   CREATE OR REPLACE TYPE Maker_T AS OBJECT
      (maker_id      VARCHAR2(10),
       maker_name    VARCHAR2(20),
       maker_address VARCHAR2(50),
       maker_phone   VARCHAR2(10))
   /
```

```
CREATE TABLE Maker OF Maker_T
   (maker_id NOT NULL,
    PRIMARY KEY (maker_id));
```

Item_T Class and Available_In Table

Now we can implement Item_T, its table, and the Available_In table, which is derived from the many-to-many relationship between tables Store and Item. The Item_T class needs an object reference to the Maker_T class on the attribute made_by. As Available_In is a regular table, we cannot have member methods. Instead, we use regular stored procedures for insertion and deletion.

Apart from generic methods, there is also a user-defined method in the Item_T class to show the store address, phone number, and the store's stock available for a given item name.

```
Relational Schemas
   — Note that the Item and Store attributes in the
   Available_In
   — table are implemented using object references
   Item
   — for the Item_T class and Store for the Store_T
   class.

   Item (item_ID, item_name, item_desc, item_cost,
   item_price, made_by)
   Available_In (item, store, item_stock)

   Class, Table, and Method Declaration
   CREATE OR REPLACE TYPE Item_T AS OBJECT
      (item_id       VARCHAR2(10),
       item_name     VARCHAR2(20),
       item_desc     VARCHAR2(50),
       item_cost     NUMBER,
       item_price    NUMBER,
       made_by REF Maker_T,

       MEMBER PROCEDURE insert_item(
         new_item_id IN VARCHAR2,
         new_item_name IN VARCHAR2,
         new_item_desc IN VARCHAR2,
         new_item_cost IN NUMBER,
         new_item_price  IN NUMBER,
```

```
        new_maker_id IN VARCHAR2),

      MEMBER PROCEDURE delete_item)
/

CREATE TABLE Item OF Item_T
   (item_id NOT NULL,
    PRIMARY KEY (item_id));

CREATE TABLE Available_In
   (item REF Item_T,
    store REF Store_T,
    item_stock   NUMBER);
```

Method Implementation

```
CREATE OR REPLACE TYPE BODY Item_T AS

   MEMBER PROCEDURE insert_item(
      new_item_id IN VARCHAR2,
      new_item_name IN VARCHAR2,
      new_item_desc IN VARCHAR2,
      new_item_cost IN NUMBER,
      new_item_price IN NUMBER,
      new_maker_id IN VARCHAR2) IS

      maker_temp REF Maker_T;

   BEGIN
      SELECT REF(a) INTO maker_temp
      FROM Maker a
      WHERE a.maker_id = new_maker_id;

      INSERT INTO Item
      VALUES (new_item_id, new_item_name, new_item_desc,
              new_item_cost,  new_item_price,
            maker_temp);
   END insert_item;

   MEMBER PROCEDURE delete_item IS

   BEGIN
      DELETE FROM Available_In
      WHERE Available_In.item IN
         (SELECT REF(a) FROM Item a
          WHERE a.item_id = self.item_id);
```

```
      DELETE FROM Item
      WHERE item_id = self.item_id;
   END delete_item;

CREATE OR REPLACE PROCEDURE Insert_Available_In(
      new_item_id IN VARCHAR2,
      new_store_id IN VARCHAR2,
      new_item_stock IN NUMBER) AS

   item_temp REF Item_T;
   store_temp REF Store_T;

   BEGIN
      SELECT REF(a) INTO item_temp
      FROM Item a
      WHERE a.item_id = new_item_id;

      SELECT REF(b) INTO store_temp
      FROM Store b
      WHERE b.store_id = new_store_id;

      INSERT INTO Available_In
      VALUES (item_temp, store_temp, new_item_stock);
   END Insert_Available_In;
/

CREATE OR REPLACE PROCEDURE Delete_Available_In(
   deleted_item_id IN VARCHAR2,
   deleted_store_id IN VARCHAR2) AS

   BEGIN
      DELETE FROM Available_In
      WHERE Available_In.item IN
         (SELECT REF(a) FROM Item a
          WHERE a.item_id = deleted_item_id)
      AND Available_In.store IN
         (SELECT REF(b) FROM Store b
          WHERE b.store_id = deleted_store_id);
   END Delete_Available_In;
/
```

Customer_T Class

Next is the implementation of the Customer_T class and its table. It has generic methods for insertion and deletion, and a user-defined method to show the item name and total given a particular customer's gender and age in a given store location.

```
Relational Schemas
   Customer (cust_ID, cust_name, cust_address,
   cust_phone,
            cust_gender, cust_DOB, bonus_point)

   Class, Table, and Method Declaration

   CREATE OR REPLACE TYPE Customer_T AS OBJECT
      (cust_id        VARCHAR2(10),
       cust_name      VARCHAR2(20),
       cust_address   VARCHAR2(50),
       cust_phone     VARCHAR2(10),
       cust_gender    VARCHAR2(3),
       cust_dob       DATE,
       bonus_point    NUMBER,

       MEMBER PROCEDURE insert_customer(
        new_cust_id IN VARCHAR2,
        new_cust_name IN VARCHAR2,
        new_cust_address IN VARCHAR2,
        new_cust_phone IN VARCHAR2,
        new_cust_gender IN VARCHAR2,
        new_cust_dob IN DATE),

          MEMBER PROCEDURE delete_customer)
   /

   CREATE TABLE Customer OF Customer_T
      (cust_id NOT NULL,
       PRIMARY KEY (cust_id));

   Method Implementation

   — The implementation can only be done if the
   table
   — Transaction has been created beforehand.

   CREATE OR REPLACE TYPE BODY Customer_T AS
      — The number of bonus points inserted for a new
      customer is 0.

      MEMBER PROCEDURE insert_customer(
         new_cust_id IN VARCHAR2,
```

```
      new_cust_name IN VARCHAR2,
      new_cust_address IN VARCHAR2,
      new_cust_phone IN VARCHAR2,
      new_cust_gender IN VARCHAR2,
      new_cust_dob IN DATE) IS

   BEGIN
      INSERT INTO Customer
      VALUES (new_cust_id, new_cust_name,
      new_cust_address,
              new_cust_phone, new_cust_gender,
            new_cust_dob, 0);
   END insert_customer;

   MEMBER PROCEDURE delete_customer IS

   BEGIN
      DELETE FROM Customer
      WHERE Customer.cust_id = self.cust_id;
   END delete_customer;

CREATE OR REPLACE PROCEDURE Show_Cust_Item(
      new_cust_gender IN VARCHAR2,
      new_cust_age IN NUMBER,
      new_store_location IN VARCHAR2) AS

   BEGIN
   CURSOR c_show_cust_item IS
      SELECT b.item_name, SUM (c.quantity) AS
      Total_Item
      FROM Customer a, Item b, Transaction c,
          Available_In d, Store e
      WHERE c.customer = REF(a) AND c.item =
      REF(b)
      AND a.cust_gender = new_cust_gender
      AND (TO_NUMBER(SYSDATE,'YYYY')-
            TO_NUMBER(a.cust_dob, 'YYYY') ) <
          new_cust_age
      AND d.item = REF(b) AND d.store = REF(e)
      AND e.store_location = new_store_location
      GROUP BY b.item_name;

   BEGIN
      DBMS_OUTPUT.PUT_LINE
      ('Item Name'||' '||'Total Item');
      DBMS_OUTPUT.PUT_LINE
```

```
        ('———————————————');
        FOR v_show_cust_item IN c_show_cust_item LOOP
           DBMS_OUTPUT.PUT_LINE
              (v_show_cust_item.item_name||' '||
               v_show_cust_item.total_item);
        END LOOP;
     END show_cust_item;
  /
```

Transaction_T Class

Next we show the implementation of the Transaction_T class and its table, with its generic member methods. Note that we have object references to the Item_T and Customer_T classes on attribute Item because the participation in this side is total.

```
Relational Schemas
  — The Customer and Item attributes are implemented
  using ref.

  Transaction (trans_ID, trans_date, customer, item,
  quantity)

  Class, Table, and Method Declaration
  CREATE OR REPLACE TYPE Transaction_T AS OBJECT
     (trans_id      VARCHAR2(10),
      trans_date    DATE,
      customer REF Customer_T,
      item REF Item_T,
      quantity      NUMBER,

      MEMBER PROCEDURE insert_transaction(
       new_trans_id IN VARCHAR2,
       new_trans_date IN DATE,
       new_cust_id IN VARCHAR2,
       new_item_id IN VARCHAR2,
       new_quantity IN NUMBER),

      MEMBER PROCEDURE delete_transaction)
  /

  CREATE TABLE Transaction OF Transaction_T
     (trans_id NOT NULL,
      PRIMARY KEY (trans_id));
```

```
Method Implementation

CREATE OR REPLACE TYPE BODY Transaction_T AS

   MEMBER PROCEDURE insert_transaction(
      new_trans_id IN VARCHAR2,
      new_trans_date IN DATE,
      new_cust_id IN VARCHAR2,
      new_item_id IN VARCHAR2,
      new_quantity IN NUMBER) IS

      cust_temp REF Customer_T;
      item_temp REF Item_T;

   BEGIN
      SELECT REF(a) INTO cust_temp
      FROM Customer a
      WHERE a.cust_id = new_cust_id;

      SELECT REF(b) INTO item_temp
      FROM Item b
      WHERE b.item_id = new_item_id;

      INSERT INTO Transaction
      VALUES (new_trans_id, new_trans_date,
              cust_temp, item_temp, new_quantity);
   END insert_transaction;

   MEMBER PROCEDURE delete_transaction IS

   BEGIN
      DELETE FROM Transaction
      WHERE Transaction.trans_id = self.trans_id;
   END delete_transaction;

END;
/
```

Building Tools Using Oracle™ Developer

In this section, we will demonstrate the usage of one of the Oracle™ development tools provided to develop a client-server application. Using

Figure 8.13. Form Builder welcome screen

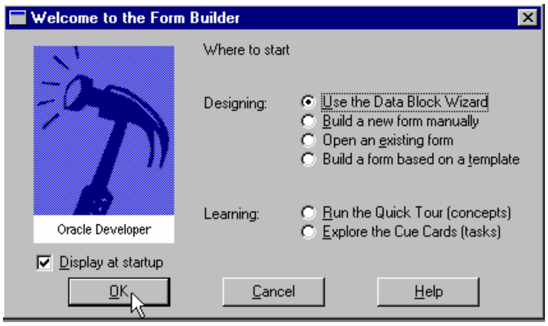
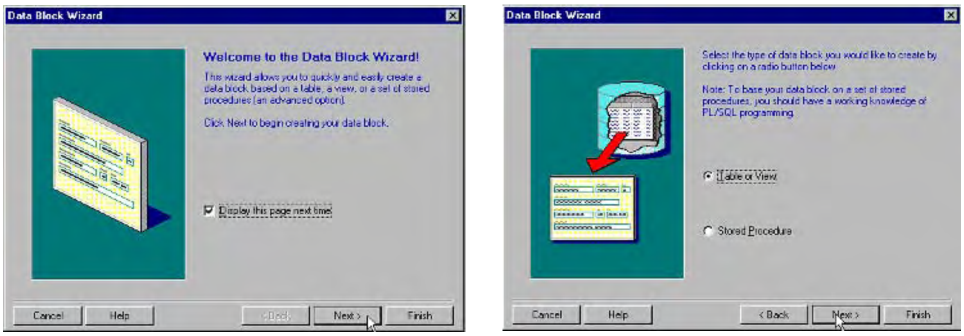


Figure 8.14. Data Block Wizard welcome screen



a. **b.**

Oracle™ forms, the users can control the layout of the screen, and these forms allow users to control the program flow in detail.

There will be two types of forms shown. The first one is the form built using the data-block form, and the second one is built using the custom form. In ORDBs, we will mainly need the second approach because there are user-defined methods to be shown. In addition, some built-in insertion and deletion methods implemented by Oracle™ Developer might not meet the ORDB requirements. Nevertheless, for demonstration purposes, we will provide both approaches in the following sections.

Creating a Form Using the Data-Block Form

In this section, we will demonstrate how to create a form using the data-block form in Form Builder. We choose the object Maker_T of the case study to be implemented using the data-block form. We notice that this object is transformed into an object table with simple attributes and without generic and user-defined methods embedded in it.

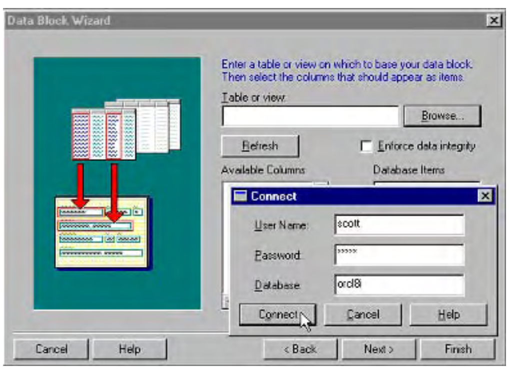
Once both sides, client and server, are ready to run the Oracle™ development tool, this step-by-step action will be very simple to follow. By choosing Form Builder from the program menu, the welcome screen should come up (see Figure 8.13). Choose the design using Data Block Wizard and click the OK button to go to the next step.

The next window shows another welcome screen, but this is the welcome screen to the Data Block Wizard (See Figure 8.14a). Click the Next button to go to the next process. In the next window (see Figure 8.14b) we have to select the type of data block, and for this example, we select the Table or View radio button and click the Next button.

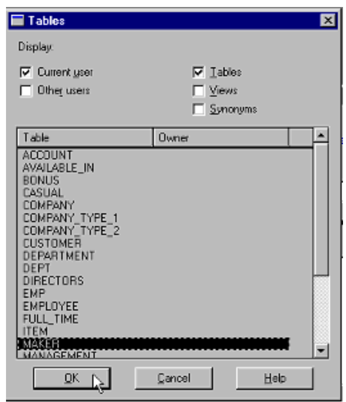
In the next window, we need to choose the table for the form. By clicking the Browse… button (see Figure 8.15a), users will be shown the list of tables and views available. However, if the user has not connected yet to the database, another window will come up to connect to the database. Once we connect to the database, the connection will remain until we log off of Form Builder. Figure 8.15b shows the window where we can select the table or view; in this case, we select table Maker.

After we select the table, the next page (see Figure 8.16a) displays the whole attributes in that particular table on the Available Column box. To select an individual attribute, we can use the single-arrow sign, but to select whole

Figure 8.15. Connecting to the database in Data Block Wizard

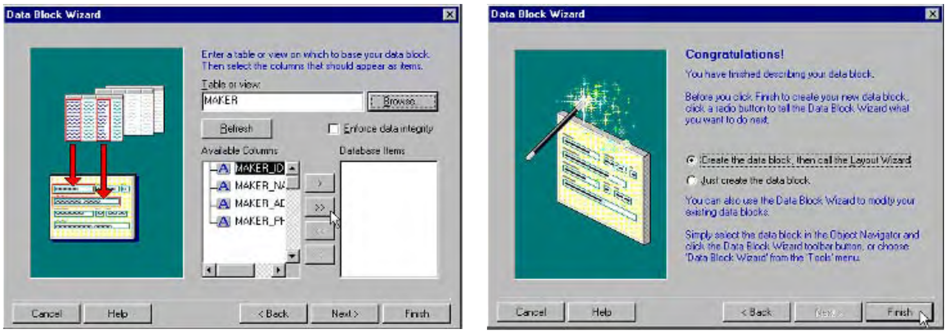


a.



b.

Figure 8.16. Selecting an attribute in Data Block Wizard and the end screen



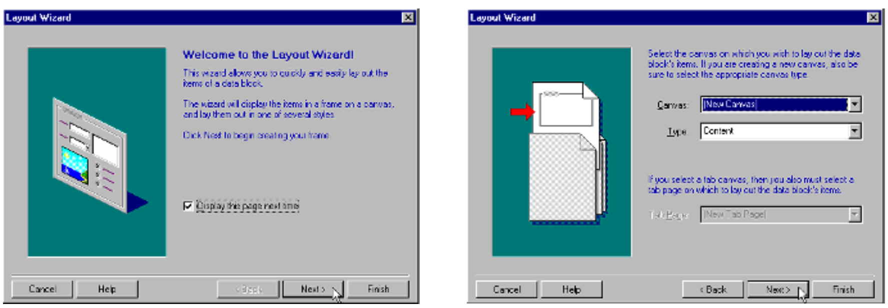
a. b.

attributes, we use the double arrow. Each attribute chosen will appear in the Database Items box.

This is the last step of Data Block Wizard; the end screen (see Figure 8.16b) should appear. We can keep the option of using the Layout Wizard to set the layout of the form by clicking the Finish button.

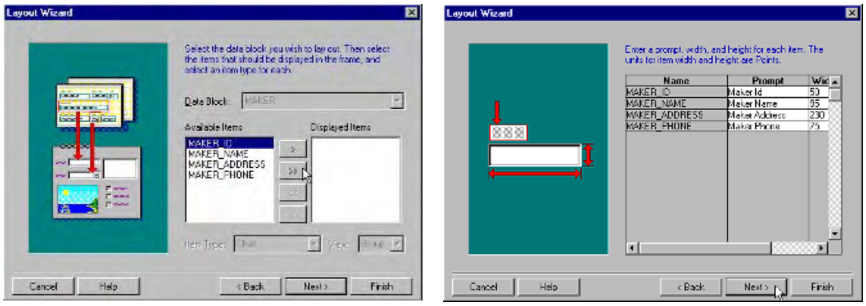
At the end of Data Block Wizard, the Layout Wizard will be displayed. The first window shown is the welcome screen (see Figure 8.17a). By clicking the Next button, we can start to build the layout of the form. Figure 8.17b shows the next window where the users can create a canvas on which the form will be displayed. We can also select the type of the canvas by choosing from the pull-down menu. For this example, we select the content type where the canvas can fill the entire window.

Figure 8.17. Layout Wizard welcome screen and creating a canvas



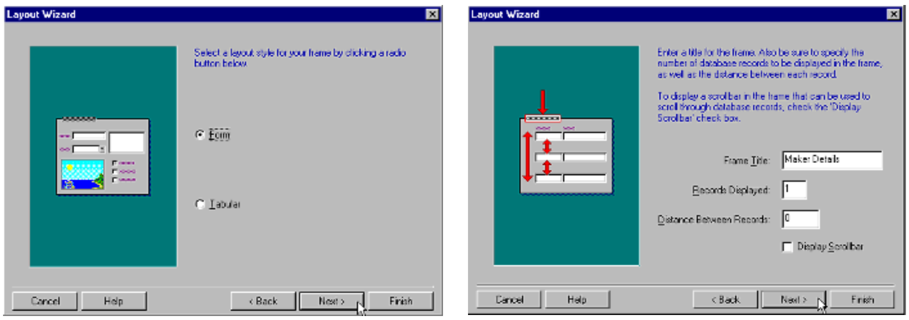
a. b.

Figure 8.18. Select columns and modify items in Layout Wizard



a. b.

Figure 8.19. Selecting-style page and setting-row page in Layout Wizard



a. b.

The next window (see Figure 8.18a) enables the user to choose the columns and attributes from the data block that will be displayed in the form. For this example, we select all columns, and by using the double arrow, we transpose them from the left box to the right one.

In the window after that (see Figure 8.18b), all of the columns created will be displayed. In this window, the user can modify the display of the prompt, and the width and the height of each item.

The next window (see Figure 8.19a) allows the users to select the layout style of the page. We select the form style where only one record can be displayed at a time. On the other side, if we select Tabular, the result will be displayed in a table format. By clicking the Next button, the next window (see Figure 8.19b) will appear where we can determine the title for the frame and the layout of the record. In our example, the frame title is "Maker Details."

Figure 8.20. Layout Wizard end screen

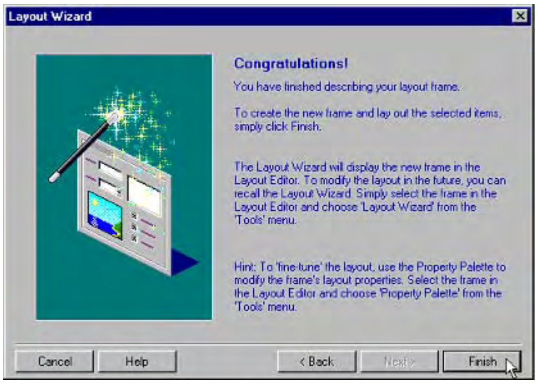
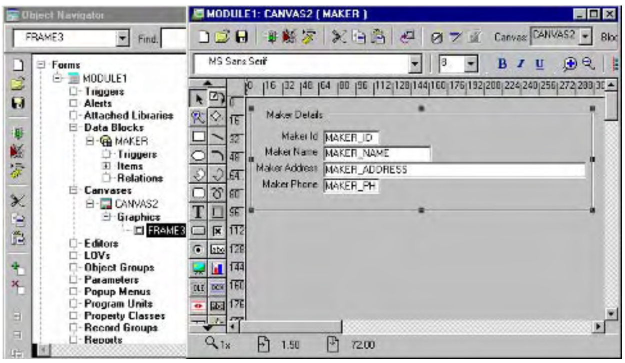


Figure 8.21. Object Navigator and Layout Editor

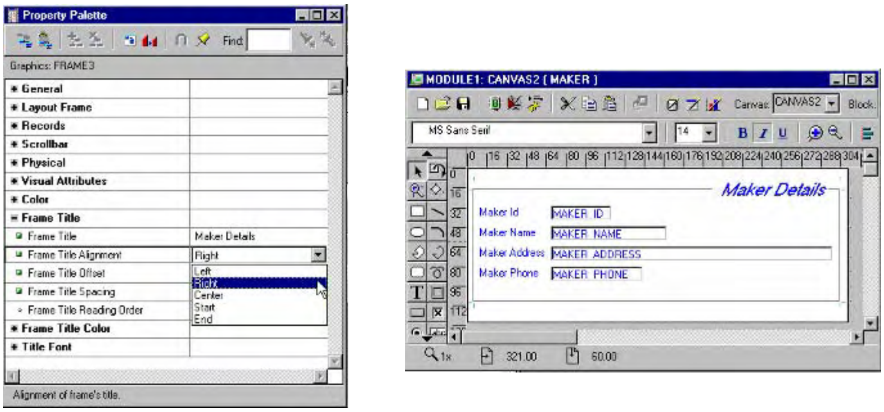


This is the last step of Layout Wizard, and the end screen (see Figure 8.20) should appear. We can keep the option of using Layout Wizard to set the layout of the form by clicking the Finish button.

Being finished with the Layout Wizard, we will automatically go to Layout Editor with the default Object Navigator (see Figure 8.21). This editor provides a graphical display of the canvas that is used to draw and to position form items.

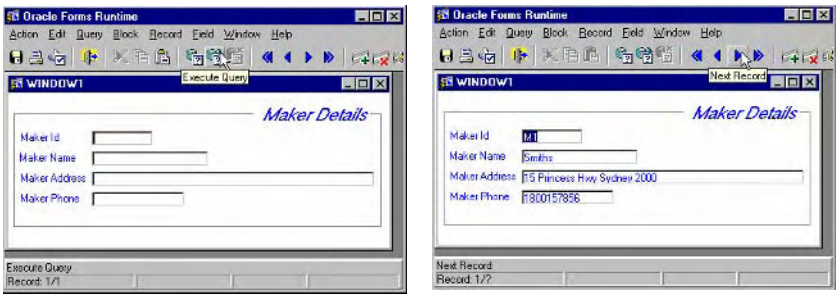
In the Layout Editor, we can edit the layout of the items by using the Property Palette (see Figure 8.22a). A simple, more appealing layout of the Maker Details form is shown in Figure 8.22b.

Figure 8.22. Property Palette and updated Layout Editor



a. **b.**

Figure 8.23. Form run time through client-server mode



a. **b.**

Once we have modified the form layout, we can run the form using different methods. One way is through the traditional client-server mode by clicking the client/server button in the Layout Editor or by selecting Client/Server from the menu Program under Run From. Figure 8.23a shows the form run-time window in client-server mode. By clicking the Execute Query button, the form will display records in the Maker table (see Figure 8.23b). We can navigate the record by clicking the arrow buttons on the toolbar.

Oracle™ forms provide insertion and deletion capabilities. Therefore, although there is no generic method embedded in an object, the form has provided its own generic methods. However, extra care has to be taken to manage the integrity constraint among objects, especially when using object references.

Figure 8.24. Inserting a record in a form

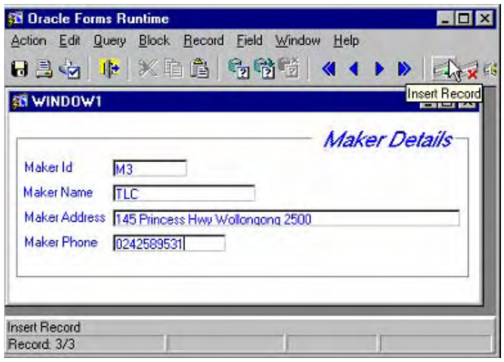


Figure 8.25. Form run time through the Web

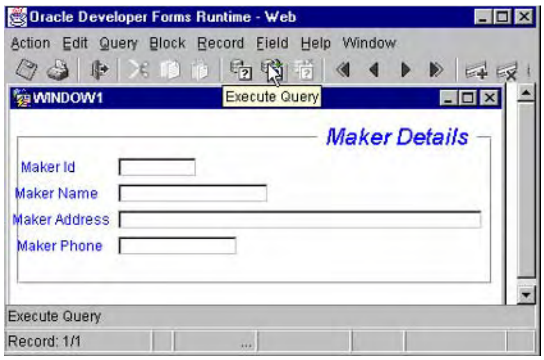
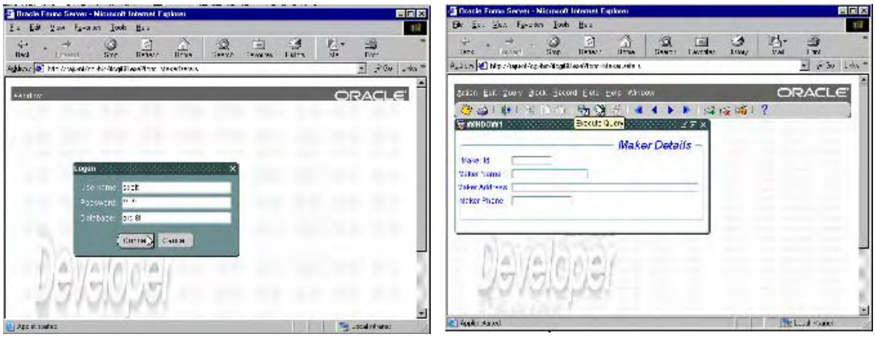


Figure 8.24 shows how to insert a new record using a form. By clicking the Insert Record button, the record is automatically inserted into the table. By clicking the delete button, we can delete a particular record from the database.

We have shown how to run a form using the traditional client-server mode. The next method is running it through the applet viewer. By doing this, we can see how the form works when it is deployed on the Web. We can do this by clicking Run from the Web button in Layout Editor, or by selecting Web from the menu Program under Run From. The window is shown in Figure 8.25.

Finally, the form can also run from a Web browser. By calling the server name using a URL (uniform resource locator), users can access the form from the server easily and it works similarly to the way it works in the traditional client-server mode. Note that the URL will depend on the setup of the server.

Figure 8.26. Form run time through a Web browser



a. b.

If we do not enter the user details in the URL, we will first have to make a connection (see Figure 8.26a). On the completion of the connection, the form will be displayed (see Figure 8.26b). This form works similarly to the previous two run methods.

Creating a Form Using a Custom Form

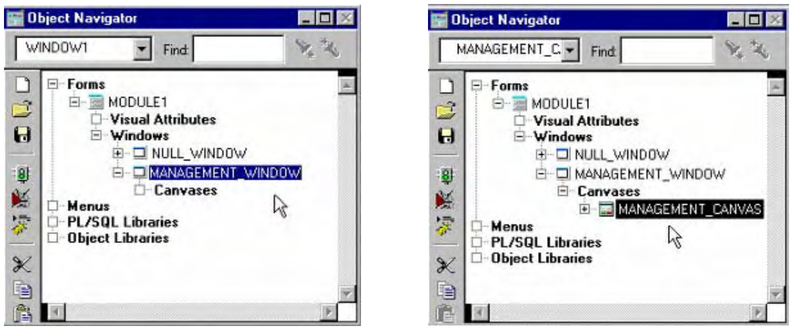
In this section we will demonstrate how to create a form using a custom form. A custom form is usually applicable for the forms that integrate several tables together. It can also be used when the users want to have more freedom in the form design. For example, we will use a custom form for the Management_T object and its subclasses Directors_T and Managers_T.

As before, by choosing Form Builder from the Program menu, the welcome screen (see Figure 8.13) should come up. Instead of choosing Data Block Wizard, we select to build a form manually. The form window with the Object Navigator should then appear, and we are ready to start building the form.

First, by changing the view from the ownership view (which is the default) to the visual view, we will see a slightly different Object Navigator. With this visual view, we can then change the name of the window created. In this case, we choose the name Management_Window (see Figure 8.27a).

Under the window, we then create a canvas (see Figure 8.27b). As in the previous section, this canvas will be used to display the records in the form. By choosing the Property Palette of the window and the canvas, we can change the

Figure 8.27. Changing window name and creating a canvas in Object Navigator



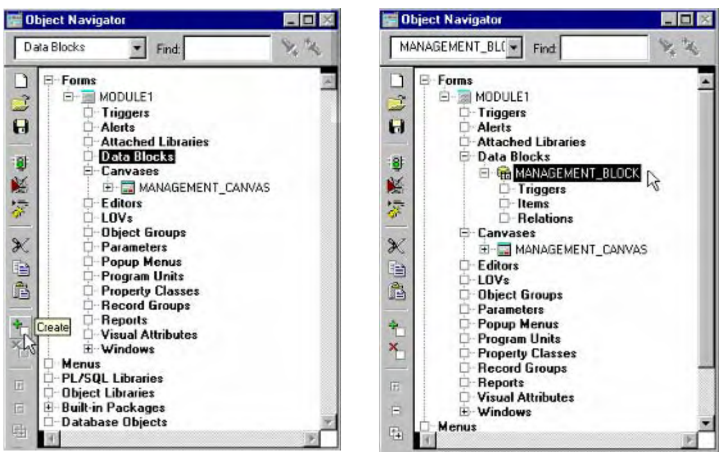
a. b.

details such as the title, the size, the scroll bar, and so forth. In our case, we choose to use the title Management Details for the window we will work on.

Now we are ready to create a special data block that is not associated with a specific database table, which is called the control block. It is recommended to design the control block first before we start building the custom form. In our case, the control block will contain records from the table Management and its subclass tables.

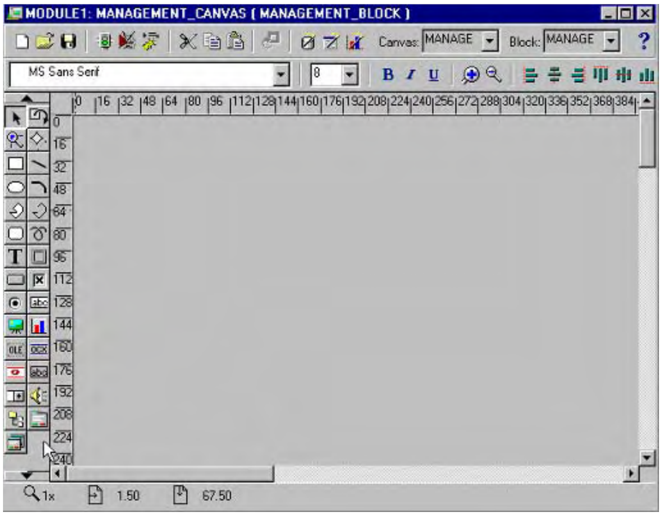
To create the control block, we have to change the view from the visual view to the ownership view under View menu. On the Object Navigator (see Figure 8.28a), we then highlight the item data block, and by clicking the Add button

Figure 8.28. Creating a data block in Object Navigator



a. b.

Figure 8.29. Empty new canvas



on the toolbar, we can start creating the new data block. A pop-up window will appear; we select to build a new data block manually.

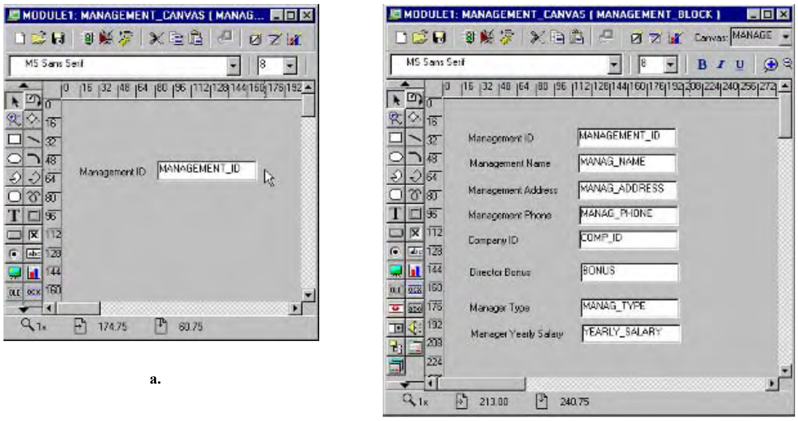
The block will appear under the data block item on the Object Navigator. We can change the name of the data block to Management_Block (see Figure 8.28b). Note that under the new data block there is a menu named Items. Items in this case are components inside the data blocks. We can create the items on the canvas that we created earlier.

Now we can open the canvas by selecting Layout Editor under the Tools menu. This canvas will be an empty canvas (see Figure 8.29). We are ready to put the items on the canvas by utilizing the toolbar on the left side of the canvas.

First, select the text item button from the toolbar and put it on the canvas. Next, do the same thing for a text button. For each item, we can change the details through the Property Palette. Figure 8.30a shows the two items with their details having been changed. The text item management_ID will show the manag_ID of the Management table. Therefore, we have to make sure that the data type and other properties match the data in the database tables.

We have to do this process for each item that we want to display on the form. Figure 8.30b shows the canvas with all items added to it. The attributes from tables Director and Manager are also included in this form. Note that we are not allowed to do this using the data-block form.

Figure 8.30. Adding items to the canvas



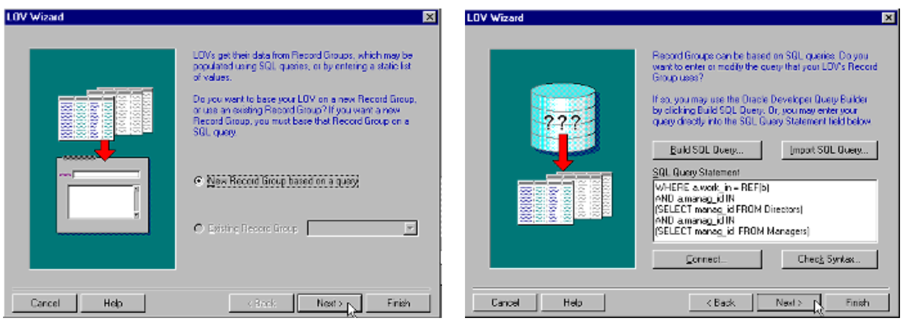
a.　　b.

At this stage, we can save the module with the name Management_Details and start linking the items to the database tables. For this purpose, one way provided by Oracle™ is by using the LOV (list of values) Wizard. The LOV is a list of legal values that can be used in a form field. It is useful for making data entry easier and avoiding errors. For the custom form, it is the way to link the separate tables together.

In our Management form, we need four different tables. The first three tables are obvious and are the Management table and its subclasses tables. The last table needed is the Company table because the data in the Management table might have references (through ref) to the data in the Company table.

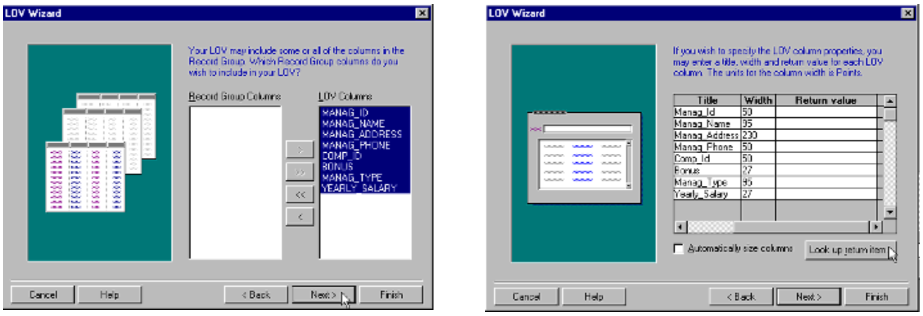
To start creating a LOV, we select LOV Wizard from the Tools menu. A first LOV Wizard window will appear (see Figure 8.31a). Keep the radio button on

Figure 8.31. Welcome screen and entering and SQL statement in LOV Wizard



a.　　b.

Figure 8.32. Selecting LOV columns in LOV Wizard



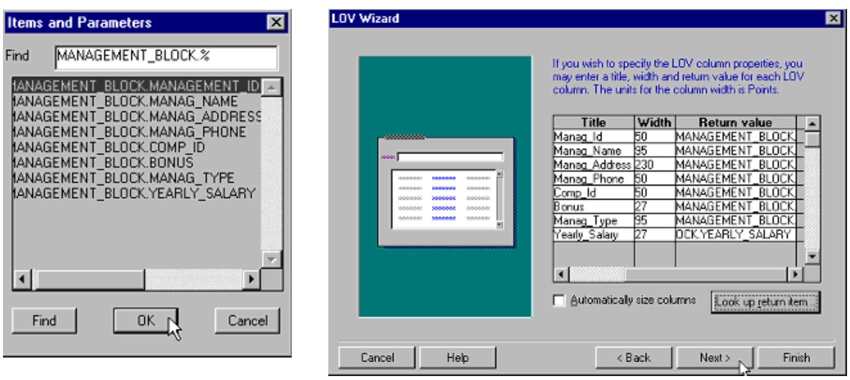
a. b.

creating a new record group. The next window (see Figure 8.31b) enables the users to specify the query statement for grouping the data in the LOV. We can also use Query Builder for this purpose by pressing the query-builder button. In this case, we link the three tables together using the references or object references.

The next window (see Figure 8.32a) shows the record group columns created. We can choose to transform the record into LOV columns by using the arrow as in the data-block form. For our example, we choose to transform all records.

In the next window (see Figure 8.32b), the properties for each column are displayed. We can change them according to the needs. The important thing to consider in this step is to choose the right look-up return item.

Figure 8.33. Return item in LOV and the complete return value



a. b.

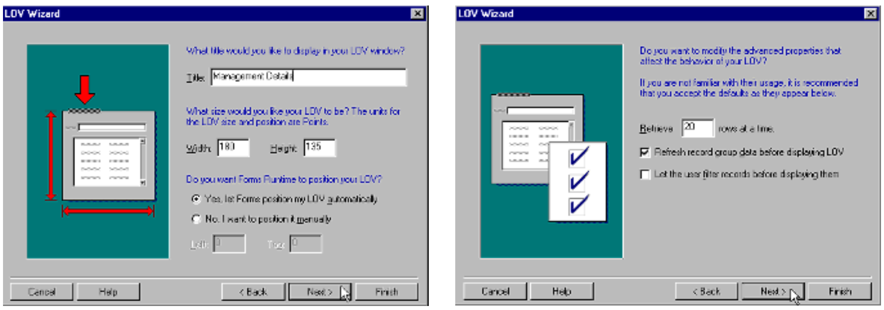
By putting the cursor on the column of a particular item, we can click the "Look up return item" button to choose which value to return to the specified item. Figure 8.33a shows the window to choose the return item.

By clicking the OK button, the particular item Manag_ID will return the value of Management_ID in the record group. It will be done for each single item (that needs return values) in the block as is shown in Figure 8.33b.

The next window (see Figure 8.34a) allows us to add the title of the LOV window, and the window after that (see Figure 8.34b) allows us to determine the number of data retrieved at a time on the advanced-options page.

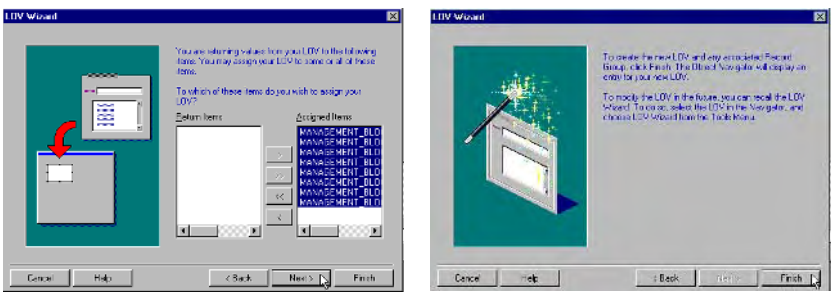
In the next window, we can assign the return items from the LOV columns. Only assigned items will be displayed on the form, and for this case, we select all of the columns (see Figure 8.35a). This is the last step of LOV Wizard, and at the end, the end screen should appear (see Figure 8.35b).

Figure 8.34. Display page settings in LOV Wizard



a. b.

Figure 8.35. Assigned items in LOV and end screen



a. b.

After creating the LOV, we can change the name of the record group and the LOV. It is mainly optional and is done for the sake of convenience. In this case, we rename them to Management_LOV (see Figure 8.36).

Next, we can add a button for the users to retrieve the record (see Figure 8.37). Recall that in Management_T we need one user-defined method to show the details of the management employees who have the roles of a director and a

Figure 8.36. Rename record group and LOV in Object Navigator

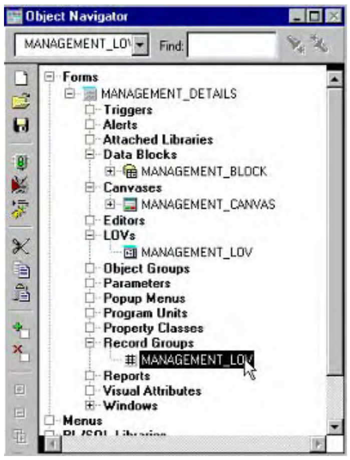
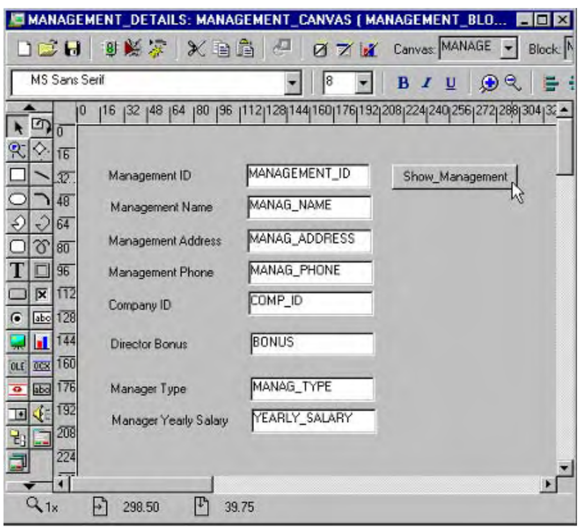


Figure 8.37. Adding a button to a custom form



manager at the same time. Actually, the method was implemented while we were creating the record group because the SQL statement in Figure 8.31b actually performs the same as the method. Therefore, using this button, we just need to retrieve the records.

Now we can add a trigger to that particular button. In this case, we will add a trigger every time the button is pressed. By choosing Smart Triggers under the Program menu, while the cursor is pointing to the button, we can be directed to the PL/SQL Editor window (see Figure 8.38).

Finally, before we run the application, we can design the look of the form. We can do this by changing the color, font, and so forth using the Property Palette or changing them directly through Layout Editor. Figure 8.39 shows the example of the same form with a better appearance.

Figure 8.38. Trigger in PL/SQL Editor

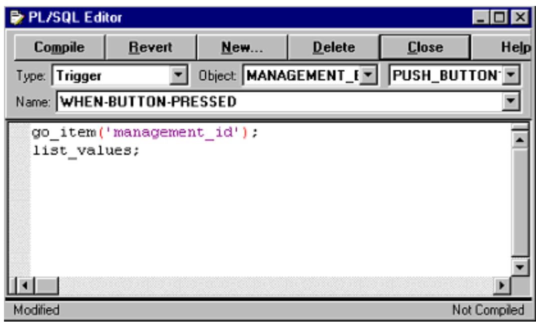


Figure 8.39. Custom form after editing

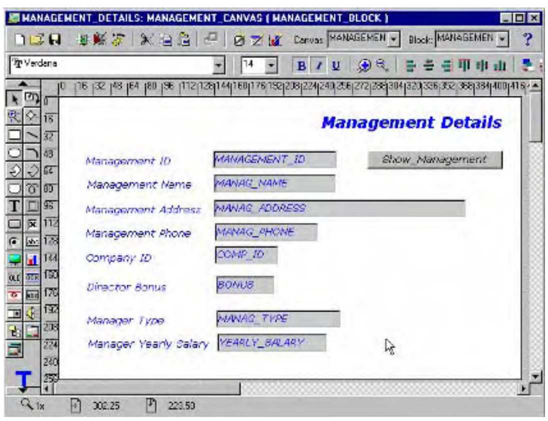
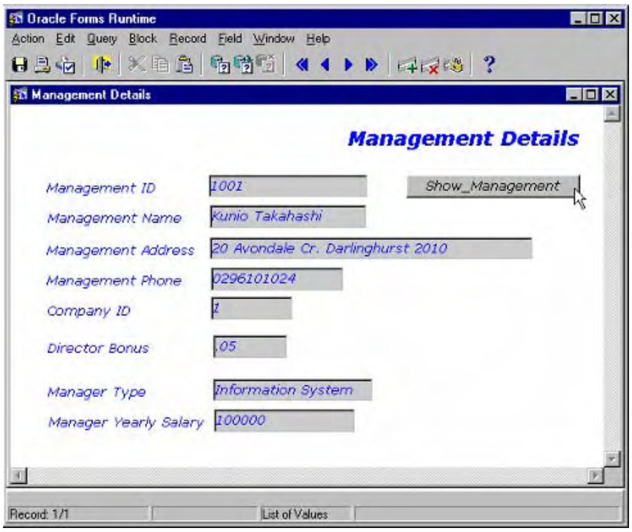


Figure 8.40. Custom form running from the client-server application



Now we are ready to run the application. Like using Data Block Wizard, the application can be run through different ways. For example, we can run it through the traditional client-server application (see Figure 8.40). Note that after we press the Show_Management button, windows will appear that list all the data retrieved.

Summary

This chapter has demonstrated an implementation of a comprehensive case study whereby object-oriented Oracle™ has been used to design and implement the tables and methods, and Oracle™ Developer was used to build the user interface of the system.

The forms implementation in this chapter was done using a very simple form application using Oracle™ Developer. With more PL/SQL applications, we can design a more interactive and powerful development tool.

About the Authors

Johanna Wenny Rahayu is an associate professor in the Department of Computer Science and Computer Engineering at La Trobe University, Australia. Her major research is in the area of object-relational databases, Web databases, and the Semantic Web. She has published more than 70 papers that have appeared in international journals and conference proceedings. She has edited three books, which form a series in Web applications, covering Web databases, Web information systems, and Web semantics. Currently, she is involved in a number of large projects on software development in collaboration with several industry partners in Australia.

David Taniar earned a PhD in databases from Victoria University, Australia (1997). He is now a senior lecturer at Monash University, Australia. He has published more than 100 research articles and edited a number of books in a Web technology series. He is on the editorial board of a number of international journals including *Data Warehousing and Mining*, *Business Intelligence and Data Mining*, *Mobile Information Systems*, *Mobile Multimedia*, *Web Information Systems*, and *Web and Grid Services*. He has been elected as a fellow of the Institute for Management of Information Systems (UK).

Eric Pardede earned his Master's in Information Technology from La Trobe University, Australia (2002). At the same university, he is currently a PhD candidate under the supervision of Dr. Wenny Rahayu and Dr. David Taniar. He has been working as a research and teaching assistant at Monash University and La Trobe University. He has published several research articles that have appeared in international journals and conference proceedings. His research area is in data modeling and query optimization for object-relational databases and XML (extensible markup language) databases.

Index