IDEA GROUP PUBLISHING
701 E. Chocolate Avenue, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com
ITB8243

Chapter IV

Practical Case Study of a Web-Based Tutor Payment System

Tanguy Chateau and Cecile Leroy
Ecole Centrale d'Electronique, France

Johanna Wenny Rahayu
La Trobe University, Australia

David Taniar
Monash University, Australia

ABSTRACT

The emerging use of object-relational databases with Web technologies has only recently begun. This chapter discusses a practical realization of an application using this technology. The aim is to show readers how to construct a full application from a design using object-oriented features up to the implementation. In this chapter, we highlight important or difficult stages with an emphasis on the mapping of object design into Oracle 8i and the use of stored procedures with the extended features for objects manipulation of Oracle 8i. This enables developers to construct professional Web applications achieving a high modularity and evolution capacity with an accelerated development phase in comparison with the traditional approach.

INTRODUCTION

This chapter is dedicated to the study of a practical case. We will show a way of implementing a Web-based application using an object-relational database. To justify the use of an object-relational database rather than an object-oriented database or a relational database, it is important to take into account the constant need of companies to produce faster applications. The object-oriented model with its characteristics of modularity, reusability, and extendibility, and its ability to stay the nearest as possible from the real world, is very suitable for these requirements. However, object-oriented databases are still not widely deployed and most companies want to be able to use the object technology without having to change their database systems. Moreover, because developers know relational systems very well, applications may be developed faster. This is the reason that we chose a system combining the advantages of relational and object-oriented features. We chose Oracle 8*i*. Since version 8, Oracle has provided some object functionalities, such as objects, ref types, collection types, nested objects, etc. This new era of DBMS (Database Management Systems), where relational DBMS is enhanced with some object-oriented features, is often known as *Object-Relational DBMS* (Stonebraker & Moore, 1996). Consequently, database design using object-oriented modeling needs to be transformed into an object-relational database schema for implementation (Rahayu and Chang, 1993).

In this case study, the Web part of the application is implemented using PHP (McCarty, 2001). This allows us to demonstrate the use of a scripting language to access a database and present the information to users. PHP has also been chosen as much for its ease of use as for its ability to demonstrate how to use a language combining presentation and logic facilities. The implementation of the database and the way to construct queries are also explained and demonstrated using PL/SQL (Urman, 1997).

The rest of this chapter is organized as follows. Firstly, we describe the case study and the database design. This is then followed by the implementation architecture. The database layer implementation using the transformation methodology is later described. The components of the two logic layers, the database and the application logic respectively, are then detailed, as well as the presentation layer. Finally, some discussions and conclusions are given.

TUTOR PAYMENT SYSTEM: A CASE STUDY

In this section, we briefly describe a tutor payment system, and the design aspect of the system, using an object-oriented modeling.

Problem Descriptions

The case study is an online tutor claim system, which allows casual tutors to submit claims of payment for their work. This payment claim is done fortnightly as the payment in the Australian University system is carried out every fortnight. The casual tutors are students, mostly postgraduates hired by a university department for each subject in each semester. They will help lecturers in their work, doing marking, labs, tutorials, and consultations.

Prior to this online system, tutors have had to lodge a blue claim form every fortnight to be approved by the associated lecturer in order to be processed in the financial system. This form contains details of activities (e.g., lab, meeting, preparation, marking, etc.), date and time, and duration. With this online system, payment claims can be made through the Web. Database maintenance is carried out by the general office personnel (i.e. administrator) who maintains tutor details, subject details, semester, budget, etc. Moreover, the general office generates a report of claims and balances. Each lecturer normally has a certain budget, which can be used to hire casual tutors for the semester. Each lecturer still needs to approve/reject claims made by his/her tutors. Once a claim is approved, it is checked by the general office before being sent to the Director of Undergraduate Studies (DUDS) for approval, then the budget allocated for the tutor is appropriately reduced. Tutors must also be able to check the state of their claims. Lecturers have to be able to approve or reject the claims and check the claims that they have already approved.

Design: Object-Oriented Model

As in any application, a primary phase of design is necessary (Oestereich B., 1999). The design is a step involving a recursive approach. It requires that the designer be able to view the application as concepts that are fitting together. A good approach described in "Developing Software with UML" (Oestereich B. 1999), the one we follow here, is the rational one. It allows the designer to find the components of the application and has the advantage of being able to give a very modular structure, allowing the development of the components in separate teams.

Figure 1 shows an object-oriented diagram of the online tutor claim system. It consists of 10 classes. Class Staff forms an *inheritance* hierarchy with classes Tutor, Lecturer, GO (General Office) and DUDS (Director of Undergraduate Studies) as its subclasses because they have some common attributes: Staff ID, Name, user ID (login), password, and email. Also notice that the inheritance is a *union inheritance*, meaning that the instances of class staff may be one or more (or none) of its subclasses. For example, a lecturer may also be a DUDS.

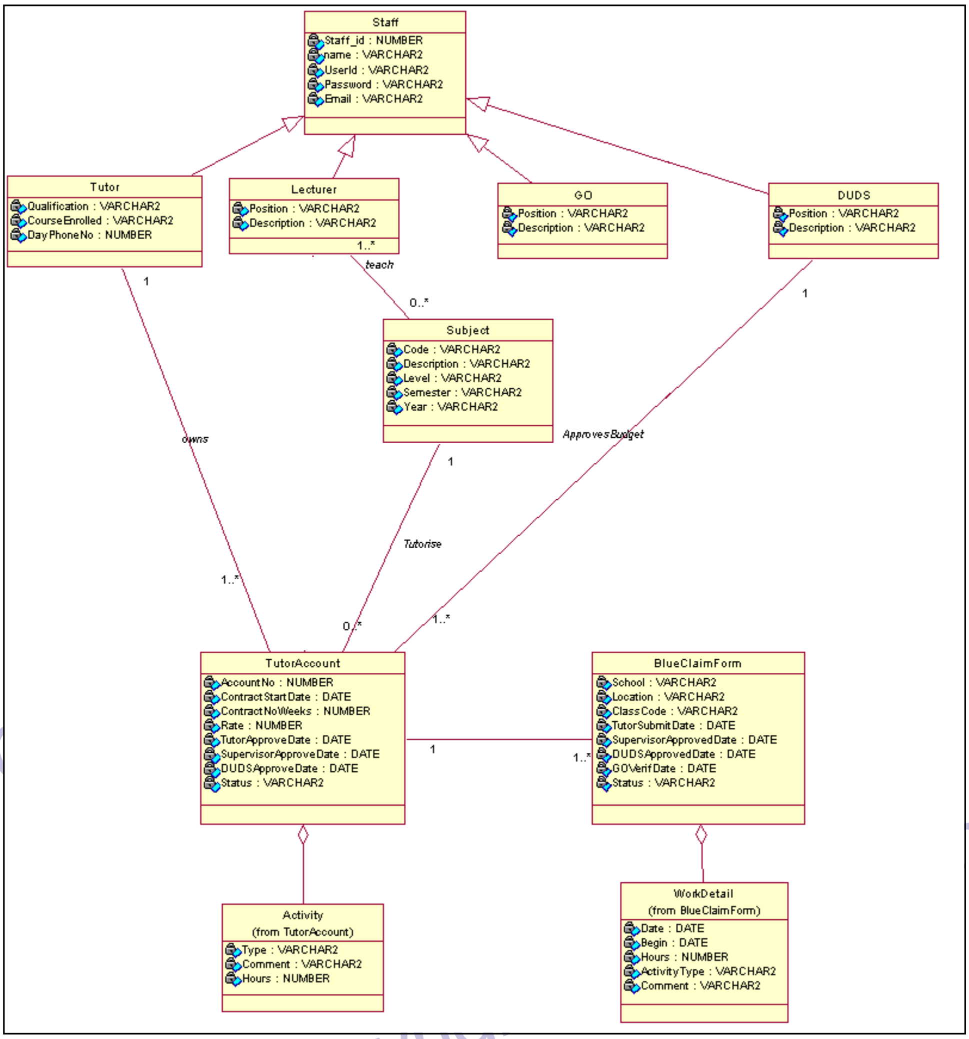
In the diagram, there are two *aggregation* (whole-part) hierarchies. One aggregation is where class TutorAccount consists of class Activity, and the other aggregation is where class BlueClaimForm consists of WorkDetail.

The *association* relationships are of cardinality *one*-to-*many* or *many*-to-*many*. For example, a *one*-to-*many* is between Tutor and TutorAccount, where a tutor may have many TutorAccount (one TutorAccount per subject). A *many*-to-*many* relationship is between Lecturer and Subject, where a lecturer may teach many subjects, and one subject may be taught by many lecturers.

IMPLEMENTATION ARCHITECTURE

We use a three-tier architecture for the application that can separate the presentation of the information from the retrieval of information and the data storage. Figure 2 shows a three-tier architecture.

Figure 1: Class diagram



In the next four sections, we are going to elaborate upon each layer in more detail. First of all, the database layer will gather all the information needed to map the object-oriented design into Oracle8*i*. Each of the relations (inheritance, aggregation, associations) will be explained and when necessary, a comparison with pure relational databases will be made. Then, the database logic layer will present how to use stored-procedures in PL/SQL for database manipulation. The explanation will begin with simple procedures, which become gradually more complex. Again, we will compare the traditional method for accessing data (directly with stand-alone queries) and our method of accessing data with procedures. In the application logic layer, we will describe how PHP interacts with procedures to manipulate the data. This layer allows us to keep the presentation layer and database layer separate. It plays the role of an interface between these two layers. Finally, the presentation layer is responsible for presenting the data to the user. It gives a way to change easily how the data are displayed and retrieved from the user. We will show with PHP a possible way of implementing this module.

DATABASE LAYER

There are three class relationships: *associations*, *inheritance*, and *aggregation*. In the following sections, we will show how to map each of these relationships. The mapping process consists of transforming the classes shown in Figure 1 into tables in Oracle. The object-relational rules used here are adopted from the object-relational transformation methodology developed by Rahayu et al. (Rahayu et al., 1993, 1995, 1998, 1999, 2000, 2001, 2002). The transformation process is summarized as follows.

Transformation of One-*to*-Many *Associations*

There are four *one*-to-*many* relationships in the object-diagram as shown in Figure 1. To describe this transformation, we use the association between Tutor and TutorAccount. The first step is to create an object type for the '*one*-part' of the relation, and then an object type for the '*many*-part'. The TutorAccount class needs to refer to the Tutor to know exactly to which Tutor it belongs. To do so, we add a *REF* in the many-part (in the TutorAccount type) pointing to Tutor type as shown in Figure 3. Note that the concept of REF in Oracle 8*i* is similar to the concept of logical pointers in object-oriented databases. REF is, however, different from the traditional references, which normally exist in a primary key (PK) – foreign key (FK) relationship.

The next step after the creation of types is to create the corresponding tables. These tables include additional information such as primary keys, which were not declared in the type creation, because type is not a table, and hence does not have a primary key. Figure 4 shows the two tables based on the types created in Figure 3.

In this design, the relationships that exist between tutor, tutor account and subject have been defined like this: a tutor may have many tutor accounts, one for each subject, but cannot have two tutor accounts for the same subject. In a conventional relational database, the attributes TA_Tutor and TA_Subject would

Figure 2: Three-tier architecture

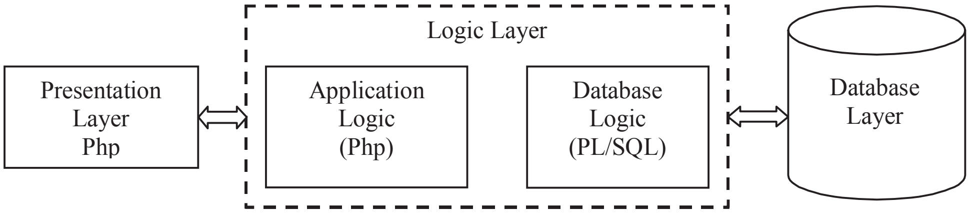


Figure 3: Type creation for a one-to-many association

```
CREATE OR REPLACE TYPE Tutor_T AS OBJECT
(
      Tutor_Qualif                VARCHAR2(20),
      Tutor_CourseEnrolled        VARCHAR2(20),
      Tutor_DayPhoneNo            VARCHAR2(20)
)
/
CREATE OR REPLACE TYPE TutorAccount_T AS OBJECT
(
      TA_Oid                NUMBER(4),
      TA_Rate               NUMBER(4,2),
      TA_StartDate          DATE,
      TA_NoWeeks            NUMBER(2),
      TA_TutAppDate         DATE,
      TA_SupAppDate         DATE,
      TA_DUDSAppDate        DATE,
      TA_Status             VARCHAR2(2),
      TA_Tutor              REF    Tutor_T
)
/
```

Figure 4: Table creation for a one-to-many association

```
CREATE TABLE Tutor OF Tutor_T
(
      Staff_Id      NOT NULL              PRIMARY KEY
);

CREATE TABLE TutorAccount OF TutorAccount_T
(
      TA_Oid        NOT NULL              PRIMARY KEY,
      TA_StartDate  NOT NULL,
      TA_NoWeeks    NOT NULL,
      TA_Rate       NOT NULL,
      TA_Tutor      SCOPE IS Tutor        –optional
);
```

have been, respectively, foreign keys of Tutor and Subject as well as the primary key of TutorAccount. Here is a reminder of how the tables would have looked:

Tutor (*TA_Tutor*, Qualification, CourseEnrolled, DayPhoneNo)
Subject (*TA_Subject*, Code, Description, Level, Year, Semester)
TutorAccount (*TA_Tutor**, *TA_Subject**, ContractStartDate, [other arguments])

Unfortunately, this is incompatible with the use of the object-oriented features of Oracle (i.e., REF): TA_Tutor and TA_Subject are REF and cannot also be primary keys. The solution is to implement this constraint in the business level (in the stored-procedures or in PHP). We chose to implement it in PHP with the help of a combo-box, which displays only the subjects that are not already taken by the tutor.

Transformation of Many-*to*-Many *Associations*

The only *many*-to-*many* association in our design is between Lecturer and Subject. For this mapping, in addition to the creation for a type and a table for each class, we need to create a new table Teach to associate these two classes. Figure 5 shows type and table creation for Subject and Lecturer (Permanent Staff), and a third table, called Teach. Notice how *REF* is used in table Teach to associate with the two other tables.

Transformation of Aggregations

In the transformation of an aggregation relationship, we use the BlueClaimForm and WorkDetail classes. This relationship is created by using a *nested table*. We choose to use a nested table rather than clusters because BlueClaimForm and WorkDetail are tightly linked together (i.e., *existent-dependant*), which means that

Figure 5: Many-to-many association

```
CREATE OR REPLACE TYPE Subject_T AS OBJECT
(
       Subject_Oid          NUMBER(4),
       Subject_Code         VARCHAR2(8),
       Subject_Desc         VARCHAR2(60),
       Subject_Level        NUMBER(1),     – 1st, 2nd, 3d or 4th year
       Subject_Sem          NUMBER(1),     – semester 1 or 2
       Subject_Year         NUMBER(4)
)
/
CREATE OR REPLACE TYPE PermStaff_T AS OBJECT
(
       Staff_Id             NUMBER(8),
       PermStaff_Position   VARCHAR2(10)
)
/
CREATE TABLE Subject OF Subject_T
(
       Subject_Oid          NOT NULL      PRIMARY KEY,
       Subject_Code         NOT NULL,
       Subject_Year         NOT NULL
);

CREATE TABLE Lecturer OF PermStaff_T
(
   Staff_id  NOT NULL  PRIMARY KEY   REFERENCES Staff (Staff_id)
);

CREATE TABLE Teach
(
       Teach_Subject        REF    Subject_T SCOPE IS Subject,
       Teach_lecturer       REF    PermStaff_T SCOPE IS Lecturer
);
```

we cannot have a WorkDetail without an associated BlueClaimForm. The limitation of nested tables is that only one level of aggregation is allowed[1]. Figure 6 shows the type and table creation for the above aggregation. Notice also how the nested table is created.

Transformation of Inheritance

The inheritance hierarchy shown in Figure 1 has class Staff as a superclass and classes Tutor, Lecturer, GO (General Office) and DUDS (Director of Undergraduate Studies) as subclasses. Because Oracle 8*i* does not provide any features for implementing inheritance[2], we need to use a combination of primary key and foreign key (Rahayu et al, 2000, 2002). The subclasses are related to the superclass by the Staff_Id attribute as a foreign key. It would have been necessary to add a Staff_Type attribute in the superclass if it were a *mutual-exclusion* inheritance (i.e. if DUDS

Figure 6: Aggregation using a nested table

```
CREATE OR REPLACE TYPE WorkDetail_T AS OBJECT
(
       Work_Oid                   NUMBER(4),
       Work_Date                  DATE,
       Work_Begin                 DATE,
       Work_NoHours               NUMBER(4),
       Work_ActivityType          VARCHAR2(5),
       Work_Comment               VARCHAR2(100)
)
/
CREATE OR REPLACE TYPE WorkDetailTable_T AS TABLE OF
WorkDetail_T
/
CREATE OR REPLACE TYPE BlueClaimForm_T AS OBJECT
(
       BCF_Oid                    NUMBER(4),
       BCF_School                 VARCHAR2(20),
       BCF_Location               VARCHAR2(20),
       BCF_ClassifCode            NUMBER(4),
       BCF_TutorSubmitDate        DATE,
       BCF_SupAppDate             DATE,
       BCF_DUDSAppDate            DATE,
       BCF_GOVerifDate            DATE,
       BCF_Status                 VARCHAR2(2),
       BCF_WorkDetails            WorkDetailTable_T
)
/
CREATE TABLE BlueClaimForm OF BlueClaimForm_T
(
       BCF_Oid                    NOT NULL      PRIMARY KEY,
       BCF_TutorSubmitDate        NOT NULL
)

NESTED TABLE BCF_WorkDetails STORE AS BCF_WorkDetails_Tab;
```

were not also a Lecturer) or a *partition* (i.e. a staff must be of one and only one of the subtypes). But in our case, since it is a *union* inheritance, this is not necessary. Figure 7 shows the type and table creation for the inheritance relationship. Note that only one object type PermStaff_T was created for DUDS, GO and Lecturer because they have the same attributes.

Figure 7: Inheritance relationships

```
CREATE OR REPLACE TYPE Staff_T AS OBJECT
(
     Staff_Id                 NUMBER(8),
     Staff_Userid             VARCHAR2(8),
     Staff_Name               VARCHAR2(40),
     Staff_Password           VARCHAR2(20),
     Staff_Email              VARCHAR2(60)
)
/
CREATE OR REPLACE TYPE Tutor_T AS OBJECT
(
     Staff_Id                 NUMBER(8),
     Tutor_Qualif             VARCHAR2(20),
     Tutor_CourseEnrolled     VARCHAR2(20),
     Tutor_DayPhoneNo         VARCHAR2(20)
)
/
CREATE OR REPLACE TYPE PermStaff_T AS OBJECT
(
     Staff_Id                 NUMBER(8),
     PermStaff_Position       VARCHAR2(10)
)
/
CREATE TABLE Staff OF Staff_T
(
     Staff_Id NOT NULL        PRIMARY KEY
);
CREATE TABLE Tutor OF Tutor_T
(
Staff_Id    NOT NULL      PRIMARY KEY   REFERENCES Staff (Staff_Id)
);
CREATE TABLE Lecturer OF PermStaff_T
(
Staff_Id    NOT NULL      PRIMARY KEY   REFERENCES Staff (Staff_Id)
);
CREATE TABLE GO OF PermStaff_T
(
Staff_Id    NOT NULL      PRIMARY KEY   REFERENCES Staff (Staff_Id)
);
CREATE TABLE DUDS OF PermStaff_T
(
Staff_Id    NOT NULL      PRIMARY KEY   REFERENCES Staff (Staff_Id)
);
```

LOGIC LAYER: DATABASE LOGIC (PL/SQL)

The database logic layer contains all the procedures necessary to insert and retrieve data from the database. It is the only access point to the database for the users (the Web-application). In other words, a user cannot create applications that run unauthorized operations. In the following sections, we will examine several stored-procedures which gradually become more complex.

Simple Insertion

Figure 8 shows a simple procedure that consists of creating a new subject. The Add_New_Subject procedure receives necessary parameters, which are the actual value of each attribute in table Subject. The procedure then calls the Insert Into statement to actually insert these attribute values into the corresponding attributes.

The only particularity of this procedure is the use of a sequence to increment the Subject_Oid automatically. Note that we first need to create the sequence as follows:

```
CREATE SEQUENCE Subject_Oid_Seq;
```

Simple Retrieval

We will now explain the procedure for retrieving a list of data. Two major points have to be taken into account: (*i*) the use of the cursor in retrieving a list of information; and subsequently, (*ii*) the use of a package. Firstly, we need to create a package header that allows the declaration of a new cursor and contains the procedure's header. Figure 9 shows a package specification, which contains a procedure called Get_Tutor_Info that requires two parameters, where the second parameter is also an output parameter.

After a package specification is defined, we need to declare the package body. Figure 10 gives the package body of the package specification shown in Figure 9. Basically, in the Get_Tutor_Info procedure, a cursor that contains the SQL to retrieve the requested data is open.

Figure 8: Simple procedure for insertion

```
PROCEDURE Add_New_Subject
(    P_Code    Subject.Subject_Code%TYPE,
     P_Desc    Subject.Subject_Desc%TYPE,
     P_Level   Subject.Subject_Level%TYPE,
     P_Sem     Subject.Subject_Sem%TYPE,
     P_Year    Subject.Subject_Year%TYPE
) IS
BEGIN
INSERT INTO Subject
(Subject_Oid, Subject_Code, Subject_Desc,Subject_Level,
Subject_Sem, Subject_Year)
VALUES
(Subject_Oid_Seq.NEXTVAL,P_Code,P_Desc, P_Level, P_Sem, P_Year);
END Add_New_Subject;
```

Figure 9: Package specification

```
CREATE OR REPLACE PACKAGE TPS AS
TYPE TPS_Cursor IS REF CURSOR;
PROCEDURE Get_Tutor_Info
(
     P_Staffid       Tutor.Staff_Id%TYPE,
     P_Tutor_Curs    IN OUT TPS_Cursor
     );
END TPS;
```

Figure 10: Package body for simple retrieval

```
CREATE OR REPLACE PACKAGE BODY TPS
AS
PROCEDURE Get_Tutor_Info(
               P_Staffid     Tutor.Staff_Id%TYPE,
               P_Tutor_Curs  IN OUT TPS_Cursor
     )
     IS
     BEGIN
               OPEN P_Tutor_Curs FOR
                      SELECT S.Staff_Name AS Name,
                             S.Staff_Userid AS Userid,
                             T.Tutor_Qualif AS Qualif
                      FROM   Tutor T,
                             Staff S
                      WHERE  S.Staff_Id = T.Staff_Id
                      AND    T.Staff_Id = P_Staffid;
     END Get_Tutor_Info;
END TPS;
```

Insertion in a Nested Table

The insertion of data into a nested table with procedures does not impose any particular difficulty. However, we have to take into account the case where the nested table does not contain any information prior to the insertion of data. Therefore, we first need to test in the procedure the existence of the nested table. To avoid this, we could automatically create a line with blank information in the nested table to insure its existence. Figure 11 shows how to test the existence of information in the nested table before proceeding to the insertion.

We do not insert the information in the table BlueClaimForm and in the nested table WorkDetail at the same time. This is because we cannot tell in advance the number of rows that the user will need to insert in the nested table for a particular blue claim form. It is better to make it possible for users to use a specific procedure in order to insert information in the nested table.

Another point is the use of function Max_Work_Oid instead of using a sequence. The only aim of this function is to provide a 'sequence' of numbers for a

Figure 11: Insertion into a nested table

```
PROCEDURE Add_New_Workdetail
(    P_Bcf_Oid        BlueClaimForm.Bcf_Oid%TYPE,
     P_Date           DATE,
     P_Begin          VARCHAR2,
     P_Hours          NUMBER,
     P_Acttype        VARCHAR2,
     P_Comment        VARCHAR2
)
IS
BEGIN
     DECLARE
              WDET   WorkDetailTable_T;
     BEGIN
              SELECT Bcf_WorkDetails INTO WDET
              FROM BlueClaimForm
              WHERE Bcf_Oid = P_Bcf_Oid;
              IF WDET IS NULL THEN
                     UPDATE BlueClaimForm
                     SET Bcf_WorkDetails =
                            WorkDetailTable_T(WorkDetail_T
                                   (1,
                                   P_Date,
                                   TO_DATE('01-01-01' ||
                                   P_Begin,'DD-MM-YY HH24:MI'),
                                   P_Hours,
                                   P_ActType,
                                   P_Comment))
                     WHERE Bcf_Oid = P_Bcf_Oid;
              ELSE
                     INSERT INTO THE (
                                   SELECT Bcf_WorkDetails
                                   FROM BlueClaimForm
                                   WHERE BCF_Oid = P_BCF_Oid)
                            (Work_Oid,
                            Work_Date,
                            Work_Begin,
                            Work_NoHours,
                            Work_ActivityType,
                            Work_Comment)
                     VALUES
                            (MAX_Works_Oid(P_BCF_Oid),
                            P_Date,
                            TO_DATE('01-01-01' || P_Begin,
                            'DD-MM-YY HH24:MI'),
                            P_Hours,
                            P_ActType,
                            P_Comment);
                     END IF;
     END;
END Add_New_WorkDetail;
```

given blue claim form. If we had used a single sequence, the number would have increased in function of all the blue claim forms inserted, which is not what we wanted! We would have had to construct dynamically those sequences needed to obtain the same result as we did with our function.

Retrieval from a Nested Table

To illustrate retrieval from a nest table, we are going to take the case of tutor's activities. The activities information is kept in the nested table Activities of Tutor Account. The Figure 12 shows how to access these data.

Without using a procedure, it is also possible to use the technique of Figure 13 called "unnesting technique".

Insertion with REF

In this section, we will show how to insert some data in a table containing REF. To illustrate this, we will use the following example: for a given TutorAccount (i.e., for a specific Tutor and for a specific Subject), a Tutor wants to submit a new BlueClaimForm. The process consists of inserting into the BlueClaimForm table the value of the attributes (i.e., school, classification code, tutorSubmitDate, …). These values are passed as parameters to a procedure called Add_New_BCF. The TutorAccount ID is also given as a parameter to know to which TutorAccount this BlueClaimForm is associated with. Figure 14 shows the procedure Add_New_BCF.

There are two main points to take into consideration. Firstly, in a stored procedure, we need to define a variable (in Figure 14 it is called TA_REF) to be able to use REF. In a stand-alone query (i.e., not in a stored-procedure), we should use REF in the INSERT INTO command, as is shown in Figure 15. The bold part of Figure 16 presents how to use REF in a stored procedure (using an intermediate variable TA_REF).

Figure 12: Retrieval from a nested table

```
PROCEDURE Get_Activities( P_Ta_Oid     Tutoraccount.Ta_Oid%TYPE,
                          P_Act_Curs   IN OUT Tps_Cursor)
IS
BEGIN
OPEN P_Act_Curs FOR
     SELECT A.Activity_Hours AS Hours
     FROM THE (SELECT  T.Ta_Activities
                FROM    TutorAccount T
                WHERE   T.Ta_Oid = P_Ta_Oid) A;
END Get_Activities;
```

Figure 13: Retrieval with unnesting technique

```
SELECT A.Activity_Hours AS Hours
FROM TutorAccount T, TABLE(T.Ta_Activities) A
WHERE T.Ta_Oid  =  P_Ta_Oid;
```

Figure 14: Insertion with REF

```
PROCEDURE Add_New_Bcf
(
P_School      BlueClaimForm.Bcf_School%TYPE,
P_Loc         BlueClaimForm.Bcf_Location%TYPE,
P_Code        BlueClaimForm.Bcf_ClassifCode%TYPE,
P_Tsd         BlueClaimForm.Bcf_TutorSubmitDate%TYPE,
P_Ta_Oid      TutorAccount.Ta_Oid%TYPE,
P_CursOid     IN OUT TPS_Cursor
)
IS
BEGIN
    DECLARE
              V_BCF_Oid     NUMBER;
              TA_REF        REF    TutorAccount_T;
    BEGIN
              SELECT REF(T) INTO TA_REF
              FROM TutorAccount T
              WHERE T.TA_Oid = P_TA_Oid;

              SELECT BCF_Oid_Seq.NEXTVAL INTO V_Bcf_Oid FROM DUAL;

              OPEN P_CursOid FOR
              SELECT BCF_Oid_SEQ.CurVal AS BCFOid FROM DUAL;

              INSERT INTO BlueClaimForm(
                                          BCF_Oid,
                                          BCF_School,
                                          BCF_Location,
                                          BCF_ClassifCode,
                                          BCF_TutorSubmitDate,
                                          BCF_Status,
                                          BCF_TutorAccount)
    VALUES ( V_BCF_Oid,
                            P_School,
                            P_Loc,
                            P_Code,
                            P_Tsd,
                            'S1',
                            TA_REF);
    END;
END ADD_NEW_BCF;
```

Figure 15: The usual way for using REF

```
 INSERT INTO BlueClaimForm( – same as above)
 VALUES (V_BCF_Oid, P_School, P_Loc, P_Code, P_TSD, 'S1',
              (SELECT REF(T)
              FROM TutorAccount T
              WHERE T.TA_Oid = P_TA_Oid));
```

Figure 16: Using REF in a procedure

```
PROCEDURE Add_New_BCF
(
P_TA_Oid       TutorAccount.TA_Oid%TYPE,
P_CursOid      IN OUT TPS_Cursor
)
IS
BEGIN
     DECLARE
               TA_REF        REF    TutorAccount_T;
     BEGIN
     SELECT REF(T) INTO TA_REF
     FROM TutorAccount T
     WHERE T.TA_Oid = P_TA_Oid;

     INSERT INTO BlueClaimForm(BCF_TutorAccount)
     VALUES (TA_REF);
     END;
END Add_New_BCF;
```

The second point is the use of cursor P_CURSOID. As we use two different procedures to insert data in the BlueClaimForm and in the nested table WorkDetail associated with this BlueClaimForm, we need to retrieve BCF_Oid created with the AddNewBCF procedure.

Figure 17 may seem to be a solution but has two drawbacks. Firstly, an affectation like the one in bold is not supported by Oracle when used in a stored procedure. Furthermore, the OUT parameter provokes an error in PHP. It works well until the P_BCFOid reaches 9 but as soon as the number is greater than 9, PHP collapses with an error: "OCIStmtExecute ORA-06502: PL/SQL: Numeric or Value error, character string buffer too small". The solution in Figure 18 was to use the Dual table and a cursor.

Figure 17: Return value of a procedure with an OUT parameter

```
PROCEDURE Add_New_BCF
(
P_TA_Oid       TutorAccount.TA_Oid%TYPE,
P_BCFOid       OUT    NUMBER
)
IS
  BEGIN
     P_BCFOid := BCF_Seq.NEXTVAL;

     INSERT INTO BlueClaimForm(BCF_Oid)
     VALUES (P_BCF_Oid);
  END;
END Add_New_Bcf;
```

Figure 18: Return values in a procedure with a cursor

```
PROCEDURE Add_New_Bcf
(
P_CursOid     IN OUT TPS_Cursor
)
IS
BEGIN
     DECLARE
               V_BCF_Oid     NUMBER;

               BEGIN
               SELECT BCF_Oid_Seq.NEXTVAL INTO V_BCF_Oid FROM DUAL;

               OPEN P_CursOid FOR
               SELECT BCF_Oid_Seq.CurrVal AS BCFOid FROM DUAL;

               INSERT INTO BlueClaimForm(BCF_Oid)
               VALUES (V_BCF_Ood);
     END;
END Add_New_Bcf;
```

Member Functions

We use some functions to execute specific operations like the calculation of budget. It would have been worth encapsulating these functions in the class as member functions (A member function is a function that may only be invoked by the object it belongs to). This would have enforced the concept of object-encapsulation.

A member function can be declared in a create type, as shown in Figure 19. This works well, as long as we do not use stored procedures; otherwise, an "invalid number error" occurs. To resolve this problem, we create a standalone function (see Figure 20). The parameter Tutor Account ID is now required to indicate on which tutor account the operations must be carried out. Note that we now declare a variable V_Budget. Since we are no longer inside a class, we need to perform a selection in the TutorAccount table to calculate the budget.

LOGIC LAYER: APPLICATION LOGIC (PHP)

The access module in PHP is composed of functions that can be gathered in a PHP class. Each function has the aim to access one or more stored procedures from the Oracle database. There are two examples of functions: one to call a procedure that inserts data, the other one to retrieve a list of data (the content of a cursor sent back by the stored procedure).

Figure 21 shows how to call the procedure to insert a new tutor. The function takes the parameters we are going to insert into the tutor table, and a parameter for the connection to the database ($connection has been obtained using the function ocilogon). The function first creates a variable $sql with the code to call the stored

Figure 19: A member function

```
CREATE OR REPLACE TYPE TutorAccount_T AS OBJECT
(
     TA_Oid                  NUMBER(4),
     TA_Rate         NUMBER(4,2),
     MEMBER FUNCTION Calc_Budget RETURN NUMBER
)
/
CREATE OR REPLACE TYPE BODY TutorAccount_T IS
     MEMBER FUNCTION Calc_Budget RETURN NUMBER IS
                 V_HoursWeek   NUMBER;
     BEGIN
— Calc_Hours calculate the number of hours worked per week
                 V_HoursWeek := Calc_Hours(TA_Oid);
                 RETURN TA_NoWeeks*V_HoursWeek*TA_Rate;
     END Calc_Budget;
END
/
```

Figure 20: Standalone function

```
FUNCTION Calc_Budget(P_TA_Oid TutorAccount.TA_Oid%TYPE) RETURN NUMBER
IS
     V_Budget NUMBER;
     V_HoursWeek     NUMBER;
BEGIN
     V_HoursWeek := Calc_Hours(P_TA_Oid);
     SELECT (TA_NoWeeks*V_HoursWeek*TA_Rate) INTO V_Budget
     FROM TutorAccount
     WHERE TA_Oid = P_TA_Oid;

     RETURN V_Budget;
END Calc_Budget;
```

procedure Insert_Tutor of the package TPS. The statement begins with the keyword BEGIN and ends with the keyword END followed by a semi colon. Once this statement is created, the function executes a PHP function OCIParse to create the executable code and keep it in variable $stmt. Then, it is necessary to bind the PHP variable to the parameters of the procedure. This is done with the OCIBindByName function. Finally, the executable code of $stmt is executed by OCIExecute and memory space is freed by OCIFreeStatement. For more information on PHP functions see the PHP official Website and the McCarty's book (2001).

Figure 22 describes how to call a procedure for retrieving the list of all subjects from the database. We will see how to call this function and use the cursor in the presentation part. The Get_Subjects stored-procedure has a cursor for parameter. This sort of parameter needs to be bind into a special variable in PHP. The variable must be declared using the OCINewCursor function. Once the declaration is done,

Figure 21: PHP function calling a PL/SQL prcedure for insertion

```
function InsertTutor($connection,
                     $staffid,
                     $qualif,
                     $enrol,
                     $phone)
{
     – PL/SQL to call the procedure INSERT_TUTOR from the package TPS.
       $sql = “BEGIN “ .
                “TPS.Insert_Tutor(:P_StaffId, :P_Qualif, :P_Enrol,
     :P_Phone);” .
              “END;” ;
     – Prepare the statement to call the procedure
       $stmt = OCIParse($connection, $sql);
     – Bind the PHP variables with the parameters of the procedure
       OCIBindByName($stmt, “:P_StaffId”, &$staffid, -1);
       OCIBindbyname($stmt, “:P_Qualif”, &$qualif, -1);
       OCIBindbyname($stmt, “:P_Enrol”, &$enrol, -1);
       OCIBindbyname($stmt, “:P_Phone”, &$phone, -1);
     – Execute the prepared statement
       OCIExecute($stmt);
     – Free the statement
       OCIFreeStatement($stmt);
}
```

it is possible to bind the PHP variable created ($cursor in Figure 22) with the parameter of the stored-procedure (“:S_C” in Figure 22). In the stored procedure, the cursor stores the result of the query proceed (a select statement). The PHP variable also needs to be a cursor so that it is possible to iterate through the list of values returned (by the “:S_C” cursor). Moreover, the command OCIExecute must be proceeded on the cursor created in PHP (OCIExecute($cursor)), once the stored procedure has been executed. Finally, the memory space reserved for the statement may be freed (OCIFreeStatement($stmt)), and the function returns the PHP cursor variable. This variable will be used when the GetAllSubjects($connection) function is called. It will contain all the values returned by the query, and it will be possible to iterate through the values, get a particular value and get the number of values in the variable. We will see an example of such a call in the presentation layer (Figure 27).

As stated earlier in this chapter, our implementation architecture is based on the use of stored procedures. The importance of stored procedure can be highlighted by describing its opposite. First of all, let us examine the InsertTutor function as shown in Figure 23. The bold part of the function shows how it is different from the call of a stored-procedure: Now the statement is not a call to a stored procedure, but a query to be executed directly on the database (an INSERT). The parameters of the query need to be bound to PHP variables (OCIBindByName), as it also would have been done for a stored procedure. However, each time a client is going to run this function, the whole statement (in the $sql variable) will be transmitted. As it is only a simple query, the amount of information transmitted in comparison with a stored procedure

Figure 22: PHP funtion calling a PL/SQL procedure for retrieving values

```
function GetAllSubjects($connection)
{
     – write the PL/SQL statement
       $sql =  "BEGIN ".
               "TPS.Get_Subjects(:S_C); ".
               "END; ";
     – Declare the cursor
       $cursor = OCINewCursor($connection);
     – Prepare the statement for execution
       $stmt = OCIParse($connection, $sql);
     – Bind the PL/SQL cursor with the PHP variable
       OCIBindByName($stmt, ":S_C", &$cursor, -1, OCI_B_CURSOR);
     – Execute the statement prepared
       OCIExecute($stmt);
     – Execute the cursor
       OCIExecute($cursor);
     – Free the prepared statement
       OCIFreeStatement($stmt);
     – Return the cursor (it will be used by the caller of the function)
       return $cursor;
}
```

is quite the same. However, if the statement were more complex (per example with a loop), the statement for the stored procedure would be transmitted only once (doing the loop inside the stored procedure), whereas the statement for the query would be transmitted the number of iterations of the loop.

Figure 23 gives an example of performing one operation; that is, inserting a record into table Tutor. Now consider a procedure with more than one operation. Figure 24 shows UpdateSubject procedure that contains several SQL statements. In this stored procedure, four update statements are executed. Figure 25 shows the PHP that calls this procedure. As has been shown previously, the SQL statement to run the stored procedure is built first and stored in a PHP variable. Then, the statement is prepared for execution (OCIParse) and each of the parameters of the stored procedure is linked to a PHP variable containing the values to pass to the function. Finally, the prepared statement is executed (OCIExecute) and the memory space reserved for execution is freed. These steps are executed only once despite the fact that four different SQL statements will be executed on the database.

On the other hand, Figure 26 shows what would have been necessary if a stored procedure had not been used. In this case, the steps (defining the query in a PHP variable, parsing the query before execution, binding the parameters of the query to PHP variables, executing the prepared statement and freeing the memory) need to be repeated for each of the four SQL statements. This example illustrates our point with a few simple queries to perform a task; however, the more complex the task, the more interesting the use of stored procedures becomes.

Figure 23: PHP function doing an insertion

```
function InsertTutor($connection,
                     $staffid,
                     $qualif,
                     $enrol,
                     $phone)
{
    – PL/SQL to call the procedure INSERT_TUTOR from the package TPS.
      $sql = “ INSERT INTO TUTOR “ .
     “(STAFF_ID,TUTOR_QUALIF,TUTOR_COURSEENROLLED,TUTOR_DAYPHONENO)”.
             “ VALUES “.
    “ (:P_StaffId, : P_Qualif, : P_Enrol, :P_Phone);” ;
    – Prepare the statement to call the procedure
      $stmt = OCIParse($connection, $sql);
    – Bind the PHP variables with the parameters
      OCIBindByName($stmt, “:P_StaffId”, &$staffid, -1);
      OCIBindbyname($stmt, “:P_Qualif”, &$qualif, -1);
      OCIBindbyname($stmt, “:P_Enrol”, &$enrol, -1);
      OCIBindbyname($stmt, “:P_Phone”, &$phone, -1);
    – Execute the prepared statement
      OCIExecute($stmt);
    – Free the statement
      OCIFreeStatement($stmt);
}
```

Figure 24: Multi-operations procedure

```
PROCEDURE UpdateSubject(P_Oid      Subject.Subject_Oid%TYPE,
                        P_Code     Subject.Subject_Code%TYPE,
                        P_Level    Subject.Subject_Level%TYPE,
                        P_Sem      Subject.Subject_Sem%TYPE,
                        P_Desc     Subject.Subject_Desc%TYPE
                       )
IS
BEGIN
    UPDATE Subject
    SET Subject_Code = P_Code
    WHERE Subject_Oid = P_Oid;

    UPDATE Subject
    SET Subject_Level = P_Level
    WHERE Subject_Oid = P_Oid;

    UPDATE Subject
    SET Subject_Sem = P_Sem
    WHERE Subject_Oid = P_Oid;

    UPDATE Subject
    SET Subject_Desc = P_Desc
    WHERE Subject_Oid = P_Oid;
END UpdateSubject;
```

Figure 25: PHP function calling the UpdateSubject procedure

```
function updateSubject($connection, $subjectoid, $code, $desc)
{
/* The subject code used at the University was composed by several
    information. For example CSE42ADB means a fourth year subject,
    second semester in the computer science departement in advanced
    databases. So to determine the level and semester we extract the
    third and fourth character of the subject code.*/
    $level = $code{3};
    $sem = $code{4};
    $sql = "BEGIN " .
            " TPS.UpdateSubject(:P_Oid, :P_Code, :P_Level, :P_Sem,
    :P_Desc); " .
            "END; " ;
    $stmt = OCIParse($connection, $sql);
    OCIBindByName($stmt, ":P_Oid", &$subjectoid, -1);
    OCIBindByName($stmt, ":P_Code", &$code, -1);
    OCIBindByName($stmt, ":P_Desc", &$desc, -1);
    OCIBindByName($stmt, ":P_Level", &$level,-1);
    OCIBindByname($stmt, ":P_Sem", &$sem, -1);
    OCIExecute($stmt);
    OCIFreeStatement($stmt);
}
```

PRESENTATION LAYER

This section shows how to use logic components to present and retrieve data to/from the user. The presentation component is responsible for displaying information it receives from the logic component (i.e., using tables, graphics, text, …), as well as retrieving the data and actions of the user (i.e., using forms). Then, we are going to show how to call the function getAllSubjects to present a list of subjects to the user and insertTutor to insert the information entered within a form by the user.

Figure 27 shows how to display a selection list with all the subjects. The SELECT tag allows us to build a combo-box on a Web page. The content of the combo-box is defined with the OPTION tag. In this example, the content of the combo-box is the list of all the subjects available. To obtain the list of all the subjects, the function GetAllSubjects is called. This function executes a stored procedure to retrieve the list of the subjects in a cursor and returns the cursor. The OCIFetch function in PHP positions the index of the cursor on the current row and returns false if there are no more rows available in the cursor. The values of the row are retrieved with the function OCIResult taking as parameters the cursor (positioned on the current row) and the column's name of the value.

The OPTION tag is constructed and echoed while there are values available.

Figure 28 details the call of the insertTutor function defined above. The first part of the code (HTML) builds a part of the form displayed for the user (Figure 29). The INPUT tags have names that are the same as the PHP names used as parameters of the inserTutor function. Then, the user enters its information in the text fields.

Figure 26: UpdateSubject function (not calling a PL/SQL procedure)

```
function updateSubject($connection, $subjectoid, $code, $desc)
{
             $level = $code{3};
    $sem = $code{4};
    $sql = “UPDATE Subject” .
             “SET Subject_Code = :P_Code” .
             “WHERE Subject_Oid = :P_Oid;”;

    $stmt = OCIParse($connection, $sql);

    OCIBindByName($stmt, “:P_Oid”, &$subjectoid, -1);
    OCIBindByName($stmt, “:P_Code”, &$code, -1);
    OCIExecute($stmt);
    OCIFreeStatement($stmt);

    $sql = “UPDATE Subject” .
             “SET Subject_Level = :P_Level” .
             “WHERE Subject_Oid = :P_Oid;”;

    $stmt = OCIParse($connection, $sql);
    OCIBindByName($stmt, “:P_LEVEL”, &$level,-1);
    OCIBindByName($stmt, “:P_Oid”, &$subjectoid, -1);
    OCIExecute($stmt);
    OCIFreeStatement($stmt);

    $sql = “UPDATE Subject” .
             “SET Subject_Sem = :P_Sem” .
             “WHERE Subject_Oid = :P_Oid;”;

    $stmt = OCIParse($connection, $sql);
    OCIBindByname($stmt, “:P_SEM”, &$sem, -1);
    OCIBindByName($stmt, “:P_Oid”, &$subjectoid, -1);
    OCIExecute($stmt);
    OCIFreeStatement($stmt);

    $sql = “UPDATE Subject” .
             “SET Subject_Level = :P_Level” .
             “WHERE Subject_Oid = :P_Oid;”;

    $stmt = OCIParse($connection, $sql);
    OCIBindByName($stmt, “:P_Desc”, &$desc, -1);
    OCIBindByName($stmt, “:P_Oid”, &$subjectoid, -1);
    OCIExecute($stmt);
    OCIFreeStatement($stmt);
}
```

When the user clicks on the button to submit the form, the data is stored in the PHP variables associated with each field’s name, and the insertTutor function is called with these variables.

Figure 27: Using a PHP function retrieving information

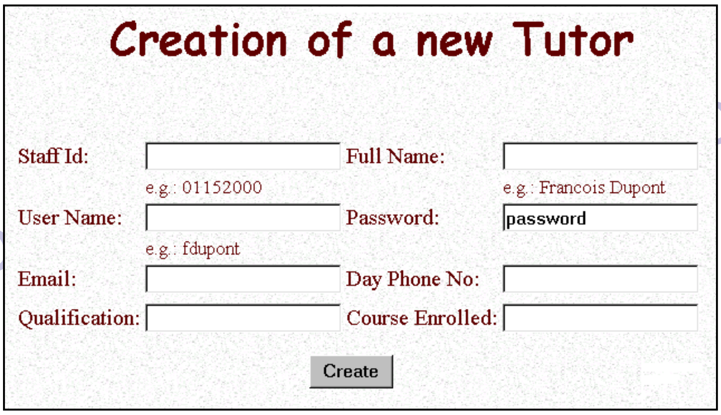
```
<SELECT>
<?php
/* Call the function getAllSubjects with connection parameter and put
     the return value (the content of the query result) in the variable
     $cursor */
$cursor=GetAllSubjects($connection);
/* Loop on all the elements of the cursor */
while(OCIFetch(&$cursor))
{
/* SubjectOid and SubjectCode are the name defined in the procedure
     */
$subjectoid = OCIResult($cursor, "SubjectOid");
$subjectcode = OCIResult($cursor, "SubjectCode");
/* add the retrieve element in the select */
echo "<OPTION VALUE= $subjectoid> $subjectcode </OPTION>";
}
/* Free the cursor */
OCIFreeCursor($cursor);
?>
</SELECT>
```

Figure 28: Using a PHP function to insert information

```
// First we define a simple form to retrieve the information
<FORM NAME="insertnewstaff" ACTION="addnewStaff.php" METHOD="POST">
<TABLE><TR>
<TD>Staff id: </TD>
<TD><INPUT TYPE="TEXT" NAME="id"></TD> </TR>
<TR><TD>Qualification: </TD>
<TD><INPUT TYPE="TEXT" NAME="qualif"></TD>
<TD>Course Enrolled: </TD>
<TD><INPUT TYPE="TEXT" NAME= "enrol"></TD>
</TR><TR>
<TD>Day Phone No: </TD><TD><INPUT TYPE="TEXT" NAME= "phone"></TD>
</TR> // add the button to submit the form here.
</TABLE>
</FORM>
// When the form is submitted, it calls the file AddNewStaff.php that
     contains the following code.
<?php
insertTutor($connection, $id, $qualif, $enrol, $phone);
?>
```

Two simple examples illustrating how to display information of the database in a Web page have just been studied. We are now going to give some tips that could enhance the presentation and the quality of a Web-site as well as examples of the result obtained for this application. The presentation's first rule is to simplify the user's task as much as possible. The number of clicks the user has to perform to

Figure 29: Creation of a new tutor (insertion)

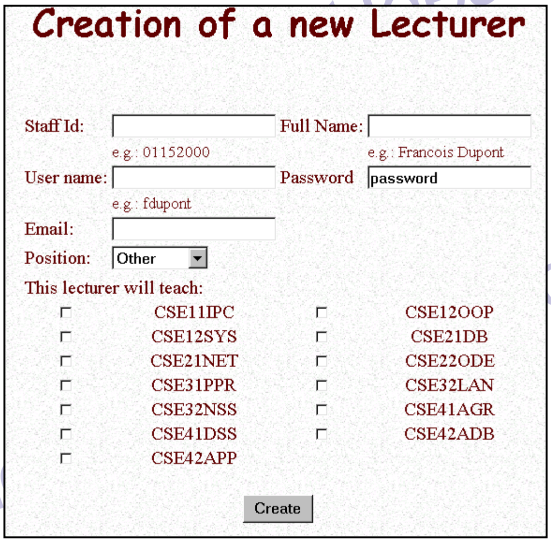

Creation of a new Tutor

Staff Id:	 e.g.: 01152000	Full Name:	 e.g.: Francois Dupont
User Name:	 e.g.: fdupont	Password:	password
Email:		Day Phone No:	
Qualification:		Course Enrolled:	

access something, or the number of fields he has to fill by himself, must be reduced to an absolute minimum. For example, the use of combo-boxes or check-boxes is usually preferable to text fields. The coherence of the database may be enhanced with this because it restricts the user's choice for entering his data. As an example, a date field will never have the same form if left to the user's initiative, but the use of combo-boxes (one for the day, one for the month and another one for the year) will allow the construction of the date format in a consistent manner. Figure 30 shows how we have used combo-boxes to propose a list of available positions for lecturers, and check-boxes for the choice of the subjects a lecturer may teach.

It is important to always recall the information the user has already entered, in particular when there is more than one step in a process. The current information

Figure 30: Creation of a new lecturer


Creation of a new Lecturer

Staff Id:	 e.g.: 01152000	Full Name:	 e.g.: Francois Dupont
User name:	 e.g.: fdupont	Password	password
Email:			
Position:	Other		

This lecturer will teach:

☐	CSE11IPC	☐	CSE12OOP
☐	CSE12SYS	☐	CSE21DB
☐	CSE21NET	☐	CSE22ODE
☐	CSE31PPR	☐	CSE32LAN
☐	CSE32NSS	☐	CSE41AGR
☐	CSE41DSS	☐	CSE42ADB
☐	CSE42APP		

should nevertheless be clearly displayed. Figure 31 represents the creation of a new Blue Claim Form and follows the previous advices.

The display of summaries in tables is often a problem for the user when these tables have many lines and columns. The user will have difficulty finding the right line or the right column, and it is generally very annoying to be obliged to go back to the top of a table each time you want to see the title of the columns. We therefore recommend that titles be repeated (all the 15 lines per example taking into account the fact that some users may not have very high-definition screens). Figures 32 and 33 are, respectively, the representation and the source code that illustrate this point. To build the table, we need to retrieve the data from the database, then as explained above, we use a function that returns a cursor with rows of data (the function runs a stored procedure on the database that fills the cursor variable with rows of data). For each row of data in the cursor, we print a line with the values of the row and if the counter ($nbrows) initialized at zero at the beginning and incremented for each line is odd, the color for the line is set to grey ($color = #E0E0E0) else the color of the row is set to white ($color = #FFFFFF). To define the color of a line we set the value of the parameter BGCOLOR of the tag TR to $color (BGCOLOR = "<?php Echo $color ?>").

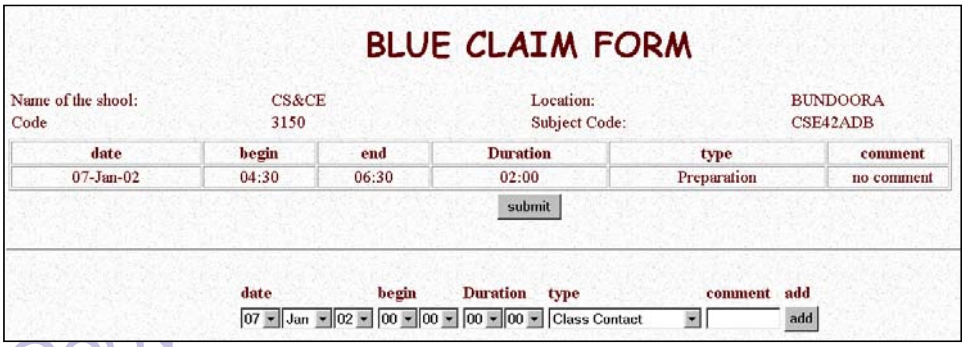
We will now consider more complex fields that show each of the points we have previously discussed.

Figure 34 shows a report that is given to the general office. This report summarises each of the tutor claims for the period, the total amount that has been paid to date, and the current balance as well as many other useful pieces of information.

Figure 35 shows the first screen that a tutor sees when he logs into the system. This gives a summary of all claims that he has made so far with their status shown by a color code. It is possible to view the details of a claim (clicking on the view button), or to update a claim either rejected or not already approved (clicking on the update button).

Figure 36 presents the screen shown to the tutor after clicking on the update button of Figure 35. All the previous data entered in the claim have been selected by default in combo-boxes or inserted in text fields. The tutor may delete a row by

Figure 31: Creation of a new blue claim form

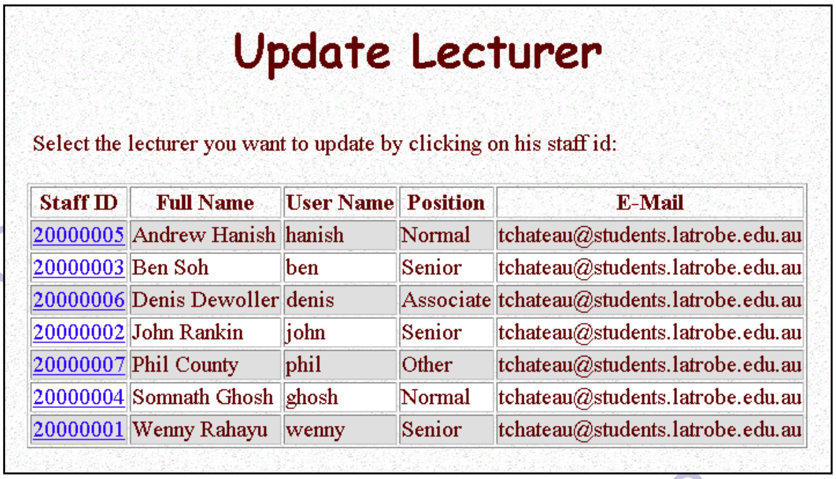


BLUE CLAIM FORM

Name of the shool:	CS&CE	Location:	BUNDOORA
Code	3150	Subject Code:	CSE42ADB

date	begin	end	Duration	type	comment
07-Jan-02	04:30	06:30	02:00	Preparation	no comment

Figure 32: A table using alternate colours to display lines



Update Lecturer

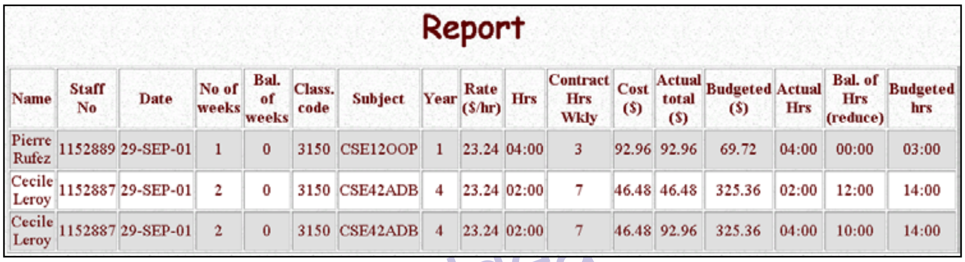
Select the lecturer you want to update by clicking on his staff id:

Staff ID	Full Name	User Name	Position	E-Mail
20000005	Andrew Hanish	hanish	Normal	tchateau@students.latrobe.edu.au
20000003	Ben Soh	ben	Senior	tchateau@students.latrobe.edu.au
20000006	Denis Dewoller	denis	Associate	tchateau@students.latrobe.edu.au
20000002	John Rankin	john	Senior	tchateau@students.latrobe.edu.au
20000007	Phil County	phil	Other	tchateau@students.latrobe.edu.au
20000004	Somnath Ghosh	ghosh	Normal	tchateau@students.latrobe.edu.au
20000001	Wenny Rahayu	wenny	Senior	tchateau@students.latrobe.edu.au

Figure 33: How to alternate the color of the lines in a table

```
<TABLE BORDER="1" ALIGN="CENTER">
    <TR>
      <TH>Staff ID</TH>
      // Idem for other titles
    </TR>
<?php
    $nbrows = 0;
    // The function getAllLecturers retrieve the information on
    lecturers
    $cursorInfo = getAllLecturers($connection);
    while(OCIFetch($cursorInfo))
    {
            $nbrows++;
            if($nbrows%2)
                  $color = "#E0E0E0";
    else
                  $color = "#FFFFFF";
            $lectid = OCIResult($cursorInfo, "STAFFID");
            $lectname = OCIResult($cursorInfo, "NAME");
            // Idem for other arguments
?>
            <TR BGCOLOR="<?php echo $color ?>" >
            <TD><?php echo $lectid ?></TD>
            <TD><?php echo $lectname ?></TD>
            // Idem for other cells
            </TR>
<?php
    }
    OCIFreeCursor($cursorInfo);
?>
</TABLE>
```

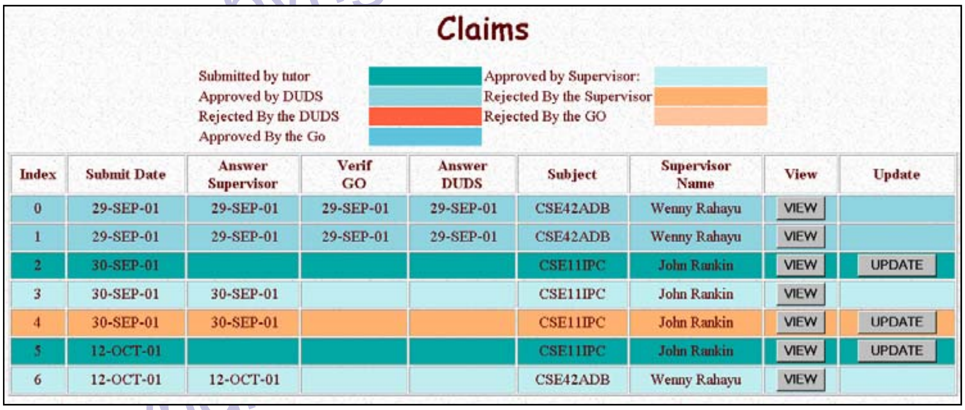
Figure 34: The report shown to the general office



Report

Name	Staff No	Date	No of weeks	Bal. of weeks	Class. code	Subject	Year	Rate ($/hr)	Hrs	Contract Hrs Wkly	Cost ($)	Actual total ($)	Budgeted ($)	Actual Hrs	Bal. of Hrs (reduce)	Budgeted hrs
Pierre Rufez	1152889	29-SEP-01	1	0	3150	CSE12OOP	1	23.24	04:00	3	92.96	92.96	69.72	04:00	00:00	03:00
Cecile Leroy	1152887	29-SEP-01	2	0	3150	CSE42ADB	4	23.24	02:00	7	46.48	46.48	325.36	02:00	12:00	14:00
Cecile Leroy	1152887	29-SEP-01	2	0	3150	CSE42ADB	4	23.24	02:00	7	46.48	92.96	325.36	04:00	10:00	14:00

Figure 35: The claims status shown to the tutors

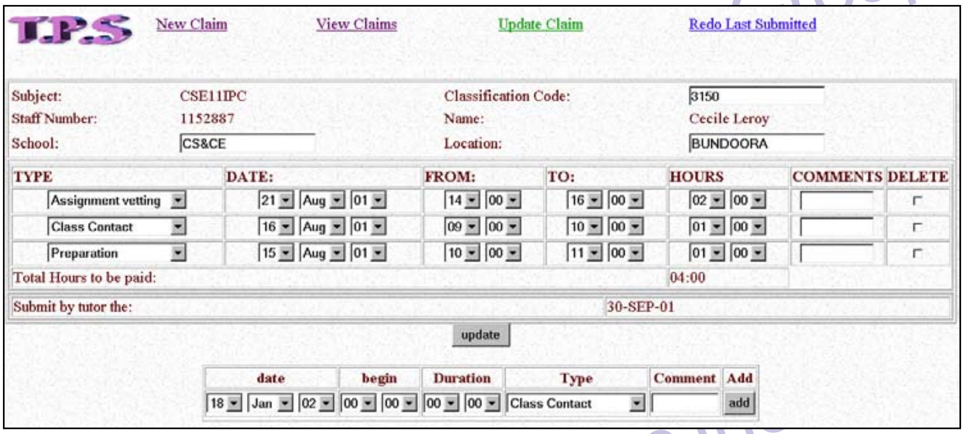


Claims

Submitted by tutor	🟦	Approved by Supervisor:	🟦
Approved by DUDS	🟦	Rejected By the Supervisor	🟧
Rejected By the DUDS	🟥	Rejected By the GO	🟧
Approved By the Go	🟦		

Index	Submit Date	Answer Supervisor	Verif GO	Answer DUDS	Subject	Supervisor Name	View	Update
0	29-SEP-01	29-SEP-01	29-SEP-01	29-SEP-01	CSE42ADB	Wenny Rahayu	VIEW	
1	29-SEP-01	29-SEP-01	29-SEP-01	29-SEP-01	CSE42ADB	Wenny Rahayu	VIEW	
2	30-SEP-01				CSE11IPC	John Rankin	VIEW	UPDATE
3	30-SEP-01	30-SEP-01			CSE11IPC	John Rankin	VIEW	
4	30-SEP-01	30-SEP-01			CSE11IPC	John Rankin	VIEW	UPDATE
5	12-OCT-01				CSE11IPC	John Rankin	VIEW	UPDATE
6	12-OCT-01	12-OCT-01			CSE42ADB	Wenny Rahayu	VIEW	

Figure 36: The update of a blue claim form



Subject:	CSE11IPC	Classification Code:	3150
Staff Number:	1152887	Name:	Cecile Leroy
School:	CS&CE	Location:	BUNDOORA

TYPE	DATE:	FROM:	TO:	HOURS	COMMENTS	DELETE
Assignment vetting	21 Aug 01	14 00	16 00	02 00		☐
Class Contact	16 Aug 01	09 00	10 00	01 00		☐
Preparation	15 Aug 01	10 00	11 00	01 00		☐
Total Hours to be paid:				04:00		

Submit by tutor the: 30-SEP-01

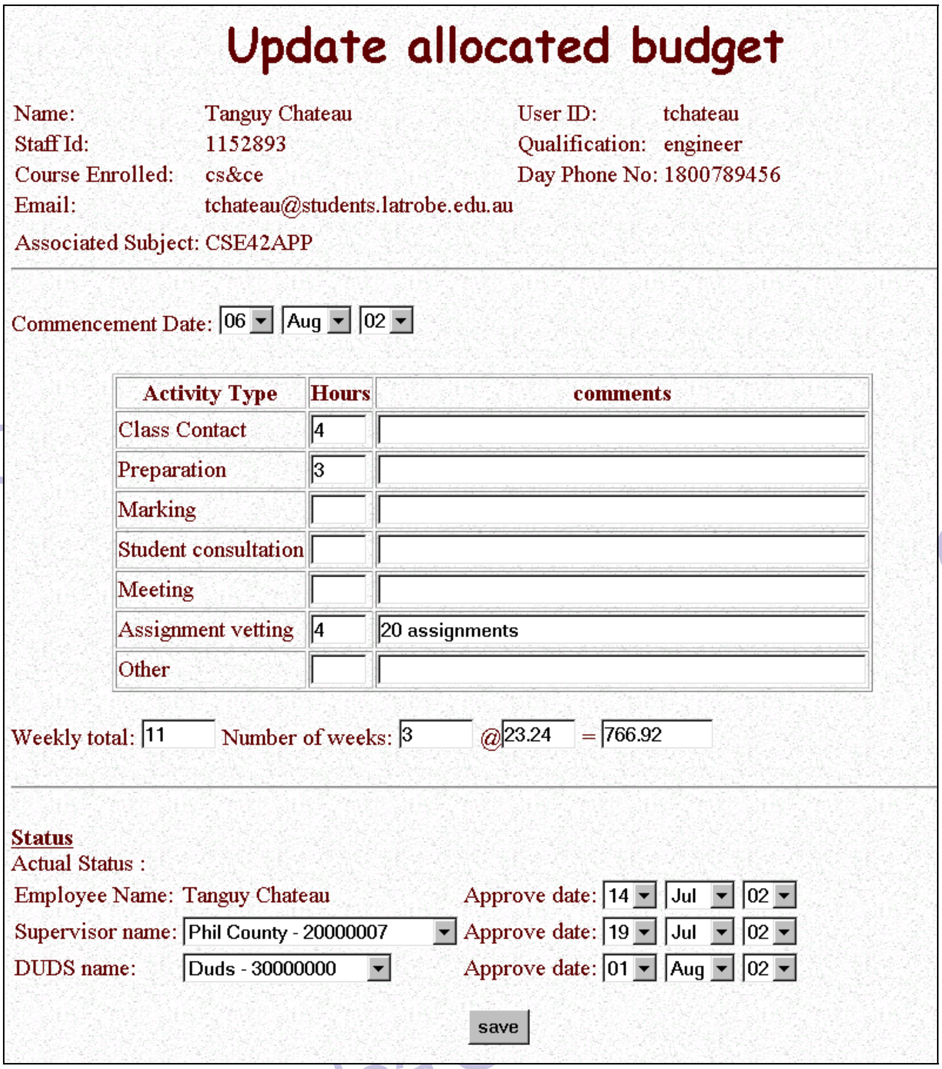
update

date	begin	Duration	Type	Comment	Add
18 Jan 02	00 00	00 00	Class Contact		add

checking the corresponding checkbox in the delete column. He may also add one or more new rows or modify the existing one. The user is restricted in the choices of values he may enter in the combo-boxes. The three columns from, to and hours are related so that a change in one of them is reflected in one of the others.

We allow the general office the possibility of updating a budget for a tutor. Figure 37 shows the form the general office encounters. The information related to the tutor is reiterated at the top of the screen. All the information entered during the creation of the budget is already shown: a change in the Activity table for the hours changed automatically, the weekly total number of hours and the amount of the budget. There again, when possible, the choices of the user have been restricted with combo-boxes (dates, supervisor name, DUDS name).

Figure 37: Update of an allocated budget



Update allocated budget

Name:	Tanguy Chateau	User ID:	tchateau
Staff Id:	1152893	Qualification:	engineer
Course Enrolled:	cs&ce	Day Phone No:	1800789456
Email:	tchateau@students.latrobe.edu.au		

Associated Subject: CSE42APP

Commencement Date: 06 Aug 02

Activity Type	Hours	comments
Class Contact	4	
Preparation	3	
Marking		
Student consultation		
Meeting		
Assignment vetting	4	20 assignments
Other		

Weekly total: 11 Number of weeks: 3 @23.24 = 766.92

Status

Actual Status :

Employee Name:	Tanguy Chateau	Approve date:	14 Jul 02
Supervisor name:	Phil County - 20000007	Approve date:	19 Jul 02
DUDS name:	Duds - 30000000	Approve date:	01 Aug 02

DISCUSSIONS

In this section, we present discussions on two particular aspects of the implementation: (*i*) architectural paradigm as previously shown in Figure 2, and (*ii*) evaluation of an object-relational paradigm.

Architectural Paradigm

The benefits of the three-layer architecture need to be emphasized. This allows the architecture to be divided into several modules that will be easily replaceable with others. For example:

One might want to develop a user interface for the Web and another one for a standalone client. In this case, we will have to replace only the presentation module of the architecture instead of having to develop the whole application again.

Similarly, one might want many applications to have access to the database; in this case, only the logic layer of this application will need to be changed. They are all going to access the database through the interface given by the database access module.

Finally, any changes to the database could be done without having to change the applications as long as the interface is not changed (given by the procedure in our case); furthermore, these changes will be taken into account in all the applications dependent on this module.

From the perspective of software engineering, another advantage of such architecture is that it allows us to allocate different modules to different teams that can work simultaneously. The main requirements are a very clear design and a definition of the interfaces between the modules (i.e. name of the functions or procedures used, their parameters and what they return).

As demonstrated in Figure 2, PL/SQL has been used in the database logic layer using stored procedures. There are two ways to implement the queries: either by directly performing the query in PHP, or by using stored procedures called by PHP functions. We preferred the second method for several reasons:

- *For purposes of security*: By keeping an object-oriented point of view, stored procedures allow us to restrict the database operations that users can perform by allowing them to access data only through procedures and functions. The restriction of access is especially useful in the case of a Web-database, allowing us to create a "Web-user" for the database that will only have the rights to run the procedures and functions.
- *For performance*: Stored procedure can improve database performance because it reduces the amount of information that must be sent over network compared to issuing individual SQL statements.
- *For maintenance*: Stored procedures may be modified and the changes will be reflected in the application without a need to recompile.
- *For memory usage*: In the case of a Web-based application, the number of users is generally important. Then, as only one copy of the stored procedures

needs to be loaded into memory for execution by multiple users, it reduces the amount of memory needed.

Stored procedures may also be very useful when several applications are using the same database. They can be shared among applications, eliminating duplicate code, coding errors, increasing overall productivity, and also enhancing the integrity providing consistency of data access across all applications.

Stored procedures may be very useful, but they also have some drawbacks. We have to keep in mind that they do not allow transactional synchronization (two-phase commit) between stored procedures; the transaction process is handled only on single procedures. Another problem is the non-standardization of stored-procedures: each vendor has its own implementation and language, which is not compatible with others.

Evaluation of Object-Relational Paradigm

The online claim system uses *Oracle Extended Object Relational* (EOR) features to implement the online claim system. EOR allows more direct mapping between the modeling and the implementation. Some of the object-oriented features, like inheritance, are not directly supported by this version of Oracle. The ability to create custom data type makes it easy for us to write stored procedures that directly use that type to define a data type.

We found out that making changes to the EOR transformation is much easier than in an entity-relationship (ER), because in EOR we use REF to represent an association. In ER, if for example a Primary Key (PK) of a table is changed, we need to change the Foreign Key (FK) of all tables referencing to this PK; whereas in EOR, this is not necessary because of REF.

In performing a query through embedded SQL in PHP or through stored procedure, we found out the query is simpler and easier to modify. For example to query, join the BlueClaimForm class and Subject class, we need only to retrieve through path expression without the need to perform an explicit join (the BlueClaimForm class has a REF called BCF_TutorAccount on TutorAccount class, and the TutorAccount class has a REF called TA_Subject on Subject class). Below is an example:

```
Select bcf.BCF_TutorAccount.TA_Subject.Subject_Code
From BlueClaimForm bcf;
```

But, if we want only to retrieve BCF_TutorAccount from BlueClaimForm class, using ER makes it easier to retrieve and more understandable. The reference value stored in BCF_TutorAccount reference field in BlueClaimForm class is not legible and is difficult to debug even if the value stored is correct or valid. The reference value is stored as 16 bytes row object id.

Oracle EOR has a few limitations that we face in our implementations. We highlight them below.

Dangling Values

If the object to which a REF pointed is deleted, the REF will be left dangling (pointing to a nonexistent object). Oracle provides the IS-DANGLING predicate to check whether a REF is dangling. In our implementation, this will make it difficult to update rows that have references to another object. For example, a tutor has submitted a claim for a subject. If a staff member accidentally deleted that subject from the offered subject table, and later realized that the subject is needed and adds the same subject to this table, the subject REF by claims class will become dangling because the REF value points to the old row. This will not happen in Relational DBMS implementation because of referential integrity constraints.

In our implementation, there are three ways to solve the dangling values problem:

i. Explicitly check for dangling value, and set the REF value to NULL or delete that record.
ii. Do not allow deletion of a record if it is referenced by another object.
iii. Use the REFERENCES keyword to impose referential integrity, that is, to create a referential integrity (like relational implementation).

The three solutions above are too restricted and not practical. Firstly, the solution is visible if we can recreate the link (and we have a history of that link). The second solution is too inflexible, but in our opinion is the most useful method to impose integrity constraint. The third solution will violate the object-oriented concept, which does not allow the use of traditional ER referencing. It also creates a redundant SCOPE BY constraint.

Type Dependency Problem

This is the most challenging problem we face in our development. We cannot modify or alter an object that has other objects dependent on it (although this problem is also found in relational DBMS implementation, it is not a trivial problem). If we force the alteration, the dependant object will become invalid. It is good if the object can be altered and Oracle is still able to recreate the dependency. In an object-oriented implementation, this is a really problematic restriction because there are many dependencies in stored procedure, type or object table. It is also a good idea if Oracle is able to automatically drop the object dependency if we alter one object.

To solve this problem, we create a set of drop script to drop the whole database schemas and recreate again. The drop sequence must be correct, and no other object must be dependent upon the drop object. Then we must alter the table in SQL script files and recreate all the schemas again.

CONCLUSIONS

In this chapter, we have described important points for the construction of a Web application using an object-relational database. The importance of the design, not only of the database but also of the whole application, has been emphasized. The advantages of using an object-oriented approach have been explained, with a consistent focus on accelerating the development life cycle of applications. After the design of the application was explained, we highlighted both the crucial and challenging parts of the implementation. The emphasis has been on how to map the object-oriented design of the database into an object-relational database with an example for each of the relations identified (i.e. *one*-to-*many*, *many*-to-*many*, aggregation, inheritance). Moreover, the use of procedures has been explained with the particularities involved when we use the extended features of Oracle 8*i*.

The use of an object-relational database as a support for the Web application has had some advantages. Firstly, this has allowed the conception of the application with an object-oriented model and an easier implementation. This model allows a better modularity and evolution for the application. It allows the development of reusable components as well as a better modularity. Then, future evolution will be easier to take into account and future development could benefit from what has already been done, so that we fulfill the requirements of an object-oriented development: encapsulation, modularity and reusability.

ENDNOTES

1 This limitation is imposed by Oracle 8*i*. In a later version (Oracle 9), this limitation no longer exists.

2 Inheritance is supported in a later version of Oracle: Oracle 9*i*.

REFERENCES

Dorsey, P. and Hudicka, J. (1999). Oracle 8 design using UML. *Oracle Press*. New York: Osborne McGraw-Hill.

McCarty, W. (2001). *PHP4: A Beginner's Guide*. New York: McGraw-Hill/Osborne.

Oestereich, B. (1999). *Developing Software with UML: Object-Oriented Analysis and Design in Practice*. Object Technology Series: Booch, Jacobson, Rumbaugh, Addison Wesley.

Urman, S. (1997). *Oracle8 PL/SQL Programming*. New York: McGraw-Hill.

Stonebraker, M. and Moore, D. (1996). *Object-Relational DBMSs The Next Great Wave*. New York: Morgan Kaufmann Publisher.

Rahayu, J.W., Chang, E., Dillon, T.S., and Taniar, D. (2002). Relational database implementation of generic methods in an inheritance hierarchy. *International Journal of Computers and Their Applications*.

Rahayu, J.W., Chang, E., Dillon, T.S., and Taniar, D. (2001). Performance evaluation of the object-relational transformation methodology. *Data and Knowledge Engineering Journal*, *28*(3), 265-300.

Rahayu, J.W., Chang, E., Dillon, T.S., and 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. and Dillon, T. S. (1999). Composite indices as a mechanism for transforming multi-level composite objects into relational databases. *The OBJECT Journal (Best of OOIS'98)*, *5*(1). Hermes Science Publications.

Rahayu, J. W., Chang, E. and Dillon, T. S. (1998). Implementation of object-oriented association relationships in relational databases. *Proceedings of the International Database Engineering and Application Symposium*, IEEE Computer Society Press.

Rahayu, J. W., Chang E. and Dillon T. S. (1995). A methodology for the design of relational databases from object-oriented conceptual models incorporating collection types. *Proceedings of the 18th International Conference on Technology of Object-Oriented Languages and Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Rahayu, J. W. and Chang E. (1993). A methodology for transforming an object-oriented data model to a relational database. *Proceedings of the 12th International Conference on Technology of Object-Oriented Languages and Systems*. Englewood Cliffs, NJ: Prentice-Hall.