DEVELOPING APPLICATIONS USING PL/SQL PACKAGES AND UPDATEABLE VIEWS

John C. Lennon, ThinkSpark

Introduction

What are Packages?

Packages are used to store PL/SQL objects either in the database of in client-side (or application server) libraries. Any PL/SQL object can be stored in a package. The most commonly used are functions and procedures, but other objects such as cursors, variables PL/SQL table declarations etc. may be included in a package. This paper discusses only database packages. The examples provided are based on the simplified bookstore application schema shown in *figure 1*.

The Advantages of Using Packages

The first advantage of using packages is that they are an ideal way to group together PL/SQL objects that are in some way related to one another. This means that there are a lot fewer objects that have to be managed. For example, there may be ten or more functions and procedures associated with a table. Storing these in a package allows us to manage just one object instead of many. In addition, packages stored in the database offer very real performance gains. The first and most obvious gain is through a very significant reduction in network traffic. The less obvious, but even more significant, gain results from how Oracle implements stored packages in the database. When any object in a package is first referenced, the entire package is compiled, validated, and loaded into memory in the Shared Global Area (SGA). The whole package is thus made available for future calls to any of its elements. This applies not just to the current session but to all other sessions. The package remains in the SGA indefinitely unless it is removed by the least-used algorithm to free up memory. PL/SQL does not have to retrieve elements from the disk when they are called. It is possible to pin a package into the SGA. However, this practice is not recommended. Only rarely used packages are likely to be affected, and packages that fall into this category are unlikely to have a significant effect on the overall performance of the application.

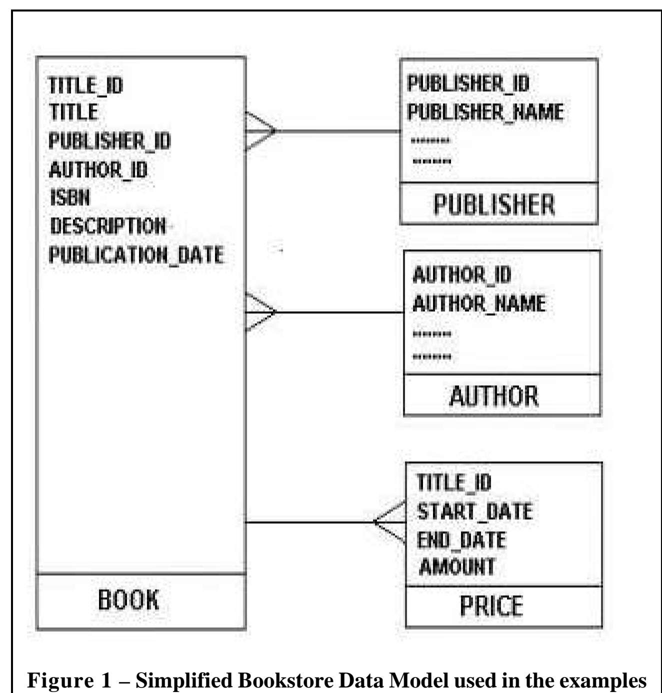

Figure 1 – Simplified Bookstore Data Model used in the examples

Modularity

When developing any type of application, it generally pays to keep pieces of code in self-explanatory modules that can perform a specific task on their own. This helps to keep your code readable, easy to debug and maintain and easy to reuse. It also makes it easier for several developers to work on the same project at the same time. The result can be significantly reduced development time.

Code Sharing

Packages allow code to be shared between different types of applications, even applications written with different development tools. For example, applications written using Oracle Forms, PowerBuilder, Visual Basic and Cold Fusion could all use the same database package. This not only speeds development but also ensures that every application enforces business rules in exactly the same way. This is of particular importance if applications are being developed by several teams. In addition, any changes in business rules require code revision in only one place. A recent update to a client's business rules in a large and complex application resulted in modifications having to be made to over seventy forms and took several weeks. As part of this exercise it was decided to convert the forms to use packages. The result of this is that any similar changes in

the future will be able to be implemented in about ten percent of the time. If the changes to be implemented affect only database objects and the package headers are unaltered there is not even a need to re-compile any application server modules.

Security

Packages can provide a great advantage in terms of security. By creating packages that encapsulate tables (see *figure 2*) those tables can be completely hidden from users and even developers. Access to, and manipulation of, data is restricted to only what is allowed by the functions and procedures contained in the packages. The level of security that can be achieved is quite high, as is described later in this paper.

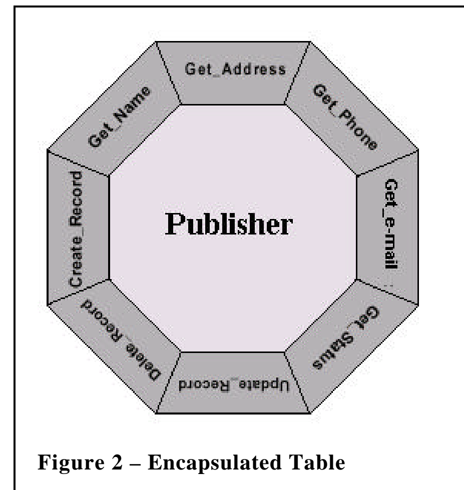

Figure 2 – Encapsulated Table

Implementation

Consensus

The first hurdle to overcome is gaining acceptance among developers of the architecture. Unfortunately, from this standpoint at least, development tools like Oracle Forms and Cold Fusion make it very easy to build modules which do not conform to the methodology described here and which are often not terribly efficient when deployed in the real world. If developers cannot be persuaded to work within a disciplined and controlled environment the methodology cannot succeed.

Planning

A methodology such as this cannot be successfully implemented on any sort of ad-hoc basis. Careful analysis of the business and data models and the requirements is needed to ensure success. There is a lot of "upfront" work required developing the packages so that at least the headers are in place when needed by the developers. This also requires fostering an understanding with the client and/or end users as, inevitably, there is going to be a longer lag than they might expect before they see the first deliverable rolled out.

Documentation

A problem with object orientation and code re-use has always been avoiding duplication of effort. Detailed documentation is necessary to ensure that the whole development team is aware of what code is available. Keeping the documentation up to date, often on a daily basis, as new objects are created also requires considerable effort. Also, probably the greatest stumbling block, is getting developers to actually read this carefully prepared documentation.

The Architecture

The basic architecture is illustrated in *figure 3*. The aim here is to move SQL (and anything else that references database objects) off of the other tiers and into the database. Ideally the user interface tools should need no knowledge of the fact that they are dealing with a relational database and the developers should need little or no knowledge of SQL.

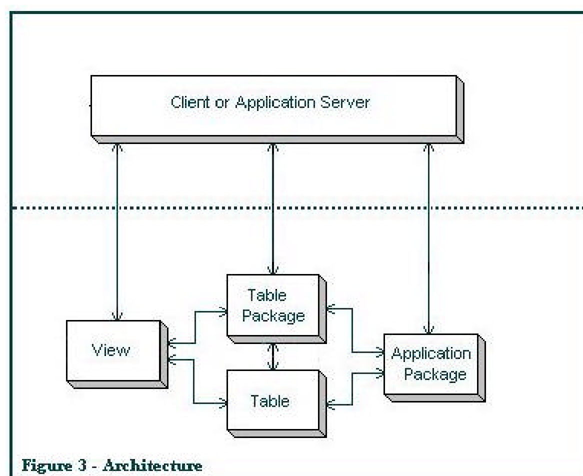

Figure 3 - Architecture

The Application Package

These packages contain common functions and procedures, i.e., those that are used by more than one module and that are not normally related to a single table. These include the application start up process, which sets up the environment and security variables when a user logs into the application, and many of the procedures used to enforce business rules. Although standard table packages contain functions that are used for specialized validation, the application package might include some generic validation routines, using dynamic SQL. Complex code is then in one place and, because there is just one generic routine, more time can be spent optimizing the routine and making it as generic and robust as possible.

Table Packages

A package is developed for each of the tables and possibly some of the views in an application. These packages contain standard procedures and functions to perform queries, inserts, deletes and updates on their associated tables. *Figure 4* shows the specification or header for a typical table package. Other functions and procedures are added to the basic package to support the requirements of a particular table. As an example of a typical package, for a table EMPLOYEE, the Get_Unique_Key function would probably return the employee number. We would add another function, Get_Employee_Name, and might also find one for Get_Manager_Name useful. A major advantage is that the developer does not need to know all of the complexities of the table's data relationships like the hierarchy linking employee and manager, but only the particular function's parameters and return value. Overloaded names can also be used. For example, the package for the PRICE table might have two functions called find_price, one with just one parameter, the title_id, the other with two, title_id and a date. The first would return the current book price, the second the price on a particular date. The developer does not need to know that there are two functions, only which parameters to pass.

Depending on which front-end tools or tools is to be used there might also be TYPE declarations in the header for any or all of RECORDS, TABLES of RECORDS and REF CURSORS. Using Oracle Forms, for instance, with blocks based on the package would require all three.

```
Package Publisher_Pkg IS
  FUNCTION Get_Name(p_Publisher_Id NUMBER) RETURN VARCHAR2;
   Pragma RESTRICT_REFERENCES(Get_Name,WNDS,WNPS);
  FUNCTION Get_Id(p_Publisher_Name VARCHAR2) RETURN NUMBER;
   Pragma RESTRICT_REFERENCES(Get_Id,WNDS,WNPS);
  FUNCTION Get_Phone(p_Publisher_Id NUMBER) RETURN VARCHAR2;
  FUNCTION Get_Credit_Limit(p_Publisher_Id NUMBER) RETURN NUMBER;
  FUNCTION Get_Status(p_Publisher_Id NUMBER) RETURN VARCHAR2;
  FUNCTION Delete_Record(p_Publisher_Id NUMBER) RETURN BOOLEAN;
  FUNCTION Create_Record(p_Publisher_Id VARCHAR2, p_Name VARCHAR2
                        ,p_Address_1 VARCHAR2 ,p_Address_2 VARCHAR2
                        ,p_City VARCHAR2 ,p_State VARCHAR2,
                        ,p_Zipcode VARCHAR2 ,P_Telephone VARCHAR2
                        ,p_Status VARCHAR2 ,p_Credit_Limit VARCHAR2)
                        RETURN BOOLEAN;
  FUNCTION Update_Record(p_Publisher_Id VARCHAR2, p_Name VARCHAR2
                        ,p_Address_1 VARCHAR2 ,p_Address_2 VARCHAR2
                        ,p_City VARCHAR2 ,p_State VARCHAR2,
                        ,p_Zipcode VARCHAR2 ,P_Telephone VARCHAR2
                        ,p_Status VARCHAR2 ,p_Credit_Limit VARCHAR2)
                        RETURN BOOLEAN;
   PROCEDURE Get_Address(p_Publisher_Id IN NUMBER
                        ,p_Address_1 OUT VARCHAR2 ,p_Address_2 OUT VARCHAR2
                        ,p_City OUT VARCHAR2 ,p_State OUT VARCHAR2,
                        ,p_Zipcode OUT VARCHAR2 ,P_Telephone OUT VARCHAR2
                        ,p_Status OUT VARCHAR2 ,p_Credit_Limit OUT VARCHAR2);
END Publisher_Pkg;
```

Figure 4 – Table Package Header

Using Views

General

Desirable as allowing access to data only through packages might be there is sometimes a need to allow users to interact more directly with the data. There may be a legitimate requirement for more experienced users to be able to view and report on the data using ad-hoc query tools.

Views have long been recognized as a means of achieving this while still maintaining control over what the user can access. For example, columns containing sensitive data that a user does not need to see can be omitted from views of that data. Views can also be used to present information in a format that is meaningful to the users while shielding them from the complexity of the relational data model.

```
Create or replace view book_vu AS
SELECT b.title
      ,p.publisher_name PUBLISHER
      ,a.author_name    AUTHOR
      ,b.isbn
      ,b.description
      ,to_char(b.publication_date,'YYYY') YEAR
      ,c.amount                           PRICE
  FROM book b
      ,publisher p
      ,author a
      ,price c
WHERE p.publisher_id = b.publisher_id
  AND a.author_id    = b.author_id
  AND c.title_id     = b.title_id
  AND c.start_date = (SELECT MAX(start_date)
                        FROM price
                       WHERE title_id = b.title_id)
```

Figure 5 – View of BOOK with lookup values

This is illustrated in *figure 5*. The information that the user needs to see can be queried as if it were contained in a single table (*figure 6*). The complexity of the joins to three other tables is hidden from the user

```
SQL>Select * from book_vu
TITLE                       PUBLISHER      AUTHOR           ISBN          DESCRIPTION                               YEAR  PRICE
--------------------------- -------------- ---------------- ------------- ---------------------------------------- ----- -----
Oracle PL/SQL Programming   Oracle Press   Scott Urman      0-07-882176-2 The essential guide for every Programmer 1996  34.95
HTML for the World Wide Web Peachpit Press Elizabeth Castro 0-201-69696-7 Teach yourself HTML the easy way         1998  17.95
Oracle JDeveloper 3 Handbook Oracle Press  Dorsey/Koletzke  0-072-12716-3 JDeveloper 3 Handbook                    2001  39.99
Building Oracle XML Apps    O'Reilly       Steve Muench     1-565-92691-9 CD Rom Edition                           2000  35.96
Oracle8i Data Warehousing   Oracle Press   M. Corey et al   0-072-12675-2 Perform data warehouse analysis          2001  39.99
Oracle Pl/SQL 101           Oracle Press   Chris Allen      0-072-12606-X Learn to use SQL and PL/SQL              2000  31.99
```

Figure 6 – Querying the View

If the user were not permitted to see the price we could simply create another view omitting the amount column and the join to the PRICE table which results in the query shown in *figure 7*.

```
SQL>Select * from book_vu2
TITLE                       PUBLISHER      AUTHOR           ISBN          DESCRIPTION                              YEAR
--------------------------- -------------- ---------------- ------------- ---------------------------------------- -----
Oracle PL/SQL Programming   Oracle Press   Scott Urman      0-07-882176-2 The essential guide for every Programmer 1996
HTML for the World Wide Web Peachpit Press Elizabeth Castro 0-201-69696-7 Teach yourself HTML the easy way         1998
Oracle JDeveloper 3 Handbook Oracle Press  Dorsey/Koletzke  0-072-12716-3 JDeveloper 3 Handbook                    2001
Building Oracle XML Apps    O'Reilly       Steve Muench     1-565-92691-9 CD Rom Edition                           2000
Oracle8i Data Warehousing   Oracle Press   M. Corey et al   0-072-12675-2 Perform data warehouse analysis          2001
Oracle Pl/SQL 101           Oracle Press   Chris Allen      0-072-12606-X Learn to use SQL and PL/SQL              2000
```

Figure 7 – Querying the View with price omitted

Updateable Views

The simplest updateable view is one created over a single table. Updates and deletes can be handled using the view. Inserts are also possible, provided the view includes any and all mandatory columns

An updateable view may also be created over two or more joined tables. In this case only the columns derived from one key-preserved table can support DML. Querying the table USER_UPDATABLE_COLUMNS shows which columns support what DML.

```
Create or replace view book_vu AS
SELECT b.title
      ,b.publisher_id
      ,p.publisher_name
      ,b.author_id
      ,a.author_name
      ,b.isbn
      ,b.description
      ,b.publication_date
  FROM book b
      ,publisher p
      ,author a
WHERE p.publisher_id = b.publisher_id
  AND a.author_id    = b.author_id
```

Figure 8 – View of BOOK with lookup values

Having created the view in *figure 8* querying USER_UPDATABLE_COLUMNS results in:

```
TABLE_NAME COLUMN_NAME      UPD INS DEL
---------- ---------------  --- --- ---
BOOK_VU    TITLE_ID         YES YES YES
BOOK_VU    TITLE            YES YES YES
BOOK_VU    PUBLISHER_ID     YES YES YES
BOOK_VU    PUBLISHER_NAME   NO  NO  NO
BOOK_VU    AUTHOR_ID        YES YES YES
BOOK_VU    AUTHOR_NAME      NO  NO  NO
BOOK_VU    ISBN             YES YES YES
BOOK_VU    DESCRIPTION      YES YES YES
BOOK_VU    PUBLICATION_DATE YES YES YES
```

As we might have expected all the columns derived from the table BOOK support update, insert and delete. The columns PUBLISHER_NAME and AUTHOR_NAME, which are lookups from the other tables, do not support any DML.

This view could be quite useful for creating and updating information about books. However, it does not include price which is in a separate table because the application stores historical data. Adding a join to the table PRICE results in that table becoming the key-preserved table.

```
TABLE_NAME COLUMN_NAME      UPD INS DEL
---------- ---------------  --- --- ---
BOOK_VU    TITLE_ID         NO  NO  NO
BOOK_VU    TITLE            NO  NO  NO
BOOK_VU    PUBLISHER_ID     NO  NO  NO
BOOK_VU    PUBLISHER_NAME   NO  NO  NO
BOOK_VU    AUTHOR_ID        NO  NO  NO
BOOK_VU    AUTHOR_NAME      NO  NO  NO
BOOK_VU    ISBN             NO  NO  NO
BOOK_VU    DESCRIPTION      NO  NO  NO
BOOK_VU    PUBLICATION_DATE NO  NO  NO
BOOK_VU    AMOUNT           YES YES YES
BOOK_VU    START_DATE       YES YES YES
BOOK_VU    END_DATE         YES YES YES
```

Only the columns derived from PRICE now support DML which is not what is required.

INSTEAD OF Triggers

INSTEAD OF triggers were introduced in Oracle8 and allow any view to be utilized as if it is an updateable view. The downside is that all of the programming logic has to be handcrafted. As this can be quite complex it requires the knowledge and experience of a seasoned PL/SQL programmer.

INSERTING

As an example let us assume our imaginary bookseller has just received a consignment of books and wants to update his stock information using the information from the delivery note. For the purposes of this exercise we are going to assume that the data is "clean" and requires no validation. Our bookseller is going to use the simple data entry screen shown in *figure 9.*

Title Author Publisher ISBN Publication Date
Description Price Price Date
SAVE

Figure 9 – Data Entry Screen

First we need to create an INSTEAD OF INSERT trigger. The syntax for this is:

```
Create or replace TRIGGER book_vu_ins
INSTEAD OF INSERT ON book_vu
FOR EACH ROW
BEGIN
………
………
………
………
END:
```

When our bookseller hits SAVE we're ready to insert into our four tables BOOK, AUTHOR, PUBLISHER, PRICE. Unfortunately it's not quite that easy. Before we can insert into BOOK, we need values for title_id, publisher_id and author_id. Title_id should be trivial if we've built a package to encapsulate BOOK and included a function to fetch the next sequence number.

```
:new.title_id := book_pkg.next_id;
```

The author and publisher are a little more difficult. There may or may not be existing records for these. If there are we just want to get the associated primary keys, if one or both does not exist we want to create a new record or records and return the primary key. This can be achieved using functions for each table as illustrated in *figure 10*. The name is passed to the function and the table is queried using the name. If the record does not exist a new record is created and, in either case, the id is returned.

```
:new.author_id     := author_pkg.insert_and_get_id(:new.author_name);
:new.publisher_id := publisher.insert_and_get_id(:new.publisher_name);
```

The new records for BOOK and PRICE can now be inserted.

UPDATING

This is somewhat similar in complexity to an insert. Firstly, the new and old values need to be compared. If the values of the non-foreign key lookup values of BOOK have changed then BOOK can simply be updated with the new values. If the publisher's or author's names have changed then the same functions as used in the insert will need to be called to get the new foreign key values, inserting new records if necessary. The business rules then could possibly call for checking if the previous publisher or author has any more books and if not deleting them. If the price or the start date has changed then the PRICE table will either be updated or have a new record created. Again business rules could play a part here, depending on how historical data is handled.

```
FUNCTION insert_and_get_id(p_Name VARCHAR2)
 RETURN NUMBER
IS
 cursor cur_id IS
   SELECT publisher_id
     FROM publisher
    WHERE publisher_name = p_Name;
 v_id NUMBER;
BEGIN
  OPEN cur_id;
    FETCH cur_id
    INTO v_id;
  CLOSE cur_id;

  IF v_id IS NULL THEN
    -- fetch the next sequence using
    -- the function publisher.next_id
    -- the package name is not needed
    -- within the same package

    v_id := next_id;

   INSERT INTO publisher
          VALUES(v_id
                ,p_Name);
  END IF;
  RETURN v_Id;
END;
```

Figure 10

DELETING

This is fairly trivial compared to inserting or updating. The records from PRICE and BOOK are simply deleted. If cascade delete is enabled this just entails deleting the BOOK records. However, as mentioned above, it may be necessary to delete the author and/or the publisher if the business rules demand it.

Security

There is no such thing as a totally secure computer system or application of any type on any platform. To achieve 100% security, it would be necessary to deny access to anyone at anytime and for any reason, which somewhat restricts the usefulness of the system to say the least. We, therefore, have to compromise and concentrate our efforts on protecting the data and preventing unauthorized access.

Oracle offers multiple levels of security, ranging from simple to complex. At the simplest level, we can control access with usernames and passwords either at the operating system level or within the database. Access to database objects can be controlled through grants, roles, procedures, triggers and views. At the highest level, and at extra cost, Oracle offers options such as Trusted Oracle (primarily used by government), the Advanced Networking Option and Oracle Application Server.

When an application is developed using packages and views to access the data there is no reason to give any grants on the tables to any user. They are then not accessible to anyone except the schema owner. To guard against a hacker gaining access logged in as the owner the CONNECT privilege can be revoked from this account, preventing logon. To further protect access to the data, the packages should be WRAPped, thus concealing the source code.

About The Author

John Lennon is a consultant with the Las Vegas practice of ThinkSpark. Originally from England; John resided for some years in South Africa before moving to the United States in 1992. He has many years of experience with applications development, primarily in engineering and utility environments. John has been working with Oracle products for over a decade. He has presented at user group meetings, international and regional Oracle users' conferences, including previous ODTUG conferences, and has published articles in Oracle technical journals in the United States and Great Britain.

John C. Lennon
e-mail: johnlennon@ieee.org or john@the-lennons.com
website: www.the-lennons.com