Effective use of Oracle Tools

Writing bug free code

Francis John 30 November, 2002

INTRODUCTION

Philosophy

The main thrust of this document is to attempt to give the reader a thorough grounding in some mental models and techniques that will make him or her a more effective programmer with the Oracle developer tools. That said, these techniques should help anyone who wishes to be a more effective programmer.

The emphasis in the text is that there is always a better way to do things and that a good programmer is *lazy*, in the sense that a solution with a lot of similar repeated code is bad and a solution with long streams of repeated operations is error prone, as is a non modular module (this will be explained later). Similarly objects like cursors should only be closed once in one well defined place. Constructive laziness means doing things once and doing them right.

This text also attempts to inculcate the reader with the idea that it is possible to write bug free code by adopting various habits of layout and cliche that will help.

Re-learning the obvious

In this context the obvious is what you see when you mental model of what you are trying to do is reasonably complete. After using the Oracle tools for ten years and watching them develop over that period the author can remember how some things used to be done and see underneath what is happening. I would argue that this makes coding easier. An over simple mental model of the solution space gives over complex, buggy solutions.

No limits

There are no limits to what can be done with the tools if you work at it. They were built to work with a relational database but if you are peverse enough you can make them work in a non relational way. You can join data from disparate sources, write horrible mangled pieces of unreadable code that will do anything that the host environment will allow. The power of the tools gives you unlimited freedom to get it wrong in a spectacular way and they also give you the freedom to get it right.

To write bug free code you need to approach the coding process with a correct attitude. This means that you need to be open to new ideas and willing to throw away things that do not work without fear.

CASE STUDY 1 : REAL PROGRAMMERS USE QUALITY CONTROL

The case against Quality Control

When I had just graduated I suffered from the big failing of most graduate programmers. I thought that I owned the code I wrote. I also thought I was perfect, although I was willing to learn more.

Most of the people who don't like quality control have this attitude. It works fine for undergraduate or hobbyist projects. I would be unhappy if the system that ran an X-ray machine at my local hospital had been written in such a way. I would feel litiginous if my children were X-rayed by it.

Keeping track of what you did and why you did it is bureaucratic and time consuming so no-one wants to do it. So is planning the project, its interfaces, what the user sees and so on. Far better to start coding now because you feel you are achieving something.

The case for Quality Control

When you work alone the only person you have to worry about looking at the code is yourself, and you wrote it, right? Wrong. Speaking as a professional, when you write thousands of lines of code and documentation, shifting several design specs, forms, reports, system overviews, quality plans etc. in a month means that you are lucky to remember what you wrote yesterday.

If you work as part of a team then you must tell the other team members what they need to know enought to look after the code you wrote (no it isn't *your code*, its *team code,* in fact *company code*). Also when you test you test to your notion of what the user will do in the best of all possible worlds. Another person testing will attempt to find the worst and catch the eviscerating bug that could have cost millions.

Don't leave a mess behind. In simple terms the effort up front involved in documenting what the system is supposed to do and how it did it, plus good comments in the code itself means that the people following after who have to maintain the system will not have to consider rewriting it to fix the bugs. Anyway, haven't you got any pride?

Simple minded Quality Control

Quality control systems are built around a systems development model. In simple terms this means that at each stage of the model you choose (there are many models) you fill in a check list that says the requirements of that stage have been met. The requirements depend upon the level of detail required. The list may say that a specification has been written, but a good QA system will have a check list for the specification ensuring it is as complete as possible.

Middle minded Quality Control

In addition to check lists you will need procedures. These specify the order things are done and who (what role) does them. Defining environments and their content is also required.

Industrial strength Quality Control

The number of teams and their general roles and responsibilities. The structure of the teams, the kinds of skills their memebers need, what roles there are within the teams. A time line with deliverable objects in it. A plan using a project management tool with resource allocation and holidays in it. A project manager.

Measurement

One of the useful things about a quality control system is that you can measure what you have achieved. You can also record, after implementation, what kind of bugs came and analyse them so that on the next project of the type you are tr

IDEAS A : DEFENSIVE PROGRAMMING

What is Defensive Programming?

As much as anything this is an attitude of mind. If you are aware of the causes of common bugs then you can write your code such that they are avoided. A simple example is where you avoid

Closure

Completeness

Coupling

Binding

Protect yourself

The main protector of your own head is your attitude. Code defensively. Declare strings and magic numbers as constants only once in a central place. Always have an else on your ifs at the end of a long series of if ... elsif blocks, because if it isn't supposed to get there then make sure it errors if it does. Assume the worst and you can't go wrong.

Some other simple things you can do are to always put parameters on your cursors. This helps particularly if you are editing a procedure or basing a new one on an old one, it prevents you leaving global variables where local ones should be and creating subtle bugs that are hard to find because the code *looks* correct to your eyes but is not correct in terms of what it delivers.

Under used features of functions and procedures

Inside a function or procedure you can declare a function or procedure that is local to it. This also works for anonymous blocks. When you are writing a procedure or package you can break each procedure down further into smaller procedures so that each step you take to the solution is a small accurate one rather than a two hundred line untested until you finished one which could be all over the place.

A simple example of this is the *McInitca*p function given here. The requirement is for a function that works like the supplied Intitcap function, which capitalises the first letter after a space or at the beginning of a string, for example:

```
select initcap('fish')
from dual;

Fish
```

However, this does not work for names like De'Ville, McClarren and so on, where you will get:

```
select initcap('de''ville')
from dual;

De'ville
```

Which is not what you want. The problem is also one of some subtlety, using a function like *replace* may cause unforseen side effects as it replaces every occurrence in the string to be processed e.g.

```
select replace('macmacmadam','macm','MacM')
from dual

MacMacMadam
```

Another problem with *replace* is that you will have to list all possible combinations of the capitalising prefix and the letter to be capitalised in you replacements. This is both time consuming and error prone. Taking some time to analyze the problem we can see that we want something that has some intelligence about the possible prefixes a name could have and then uppercase the following letter if the *beginning* of the name matches the prefix. To do this we need a code fragment like:

```
p_name := substr(p_name,1,v_prefix_length)
                        ||  initcap(substr(p_name,v_prefix_length+1,1))
                        ||  substr(p_name,v_prefix_length+2) ;
```

Where the string to be processed is called p_name (it's a parameter) and the prefix length has been calculated beforehand. The next thing we need to do is find out if the prefix is in the string before processing it. As a first cut we could try something like the following:

```
create or replace function mcInitCap( p_name in varchar2 ) return varchar2 is
retval varchar2(50) ;
v_prefix_length number ;
v_prefix varchar2(10);
begin
    retval := initcap( p_name ) ;
    v_prefix := 'Mc';
    v_prefix_length := length(v_prefix);
    if instr( p_name, v_prefix ) > 0 then
            retval := substr(retval,1,v_prefix_length)
                        ||  initcap(substr(retval,v_prefix_length+1,1))
                        ||  substr(retval,v_prefix_length+2) ;
    end if ;
    v_prefix := 'Mac';
    v_prefix_length := length(v_prefix);
    if instr( retval, v_prefix ) > 0 then
            retval := substr(retval,1,v_prefix_length)
                        ||  initcap(substr(retval,v_prefix_length+1,1))
                        ||  substr(retval,v_prefix_length+2) ;
    end if ;
    v_prefix := 'De''';
    v_prefix_length := length(v_prefix);
    if instr( retval, v_prefix ) > 0 then
            retval := substr(retval,1,v_prefix_length)
                        ||  initcap(substr(retval,v_prefix_length+1,1))
                        ||  substr(retval,v_prefix_length+2) ;
    end if ;
    -- And so on ...
    return retval ;
end ;
```

This breaks several rules of good programming practice, as you can see the code fragment that processes the string is repeated all over the place. I have already factored the code a little in that the v_prefix variable is set and then reused, rather than the string being hard coded. This is part of the habit of minimising the places an error can occur. If you are forced to cut and paste to do repetetive tasks like this then it is a good technique. The *v_prefix_length* variable appears to be unnecessary as you can see how long the prefix is by eye but then you would have to change it every time, which is error prone.

It seems obvious that the repeated code could be factored into two highly dependant routines as follows:

```
create or replace procedure mcReplace( p_prefix in varchar2, p_name in out varchar2 ) is
v_prefix_length number := length( p_prefix ) ;
begin
    if instr( p_name, p_prefix ) > 0 then
        p_name := substr(p_name,1,v_prefix_length)
                    ||  initcap(substr(p_name,v_prefix_length+1,1))
                    ||  substr(p_name,v_prefix_length+2) ;
    end if ;
end McReplace ;
create or replace function mcInitCap( p_name in varchar2 ) return varchar2 is
retval varchar2(50) ;
begin
    retval := initcap( p_name ) ;
    McReplace( 'Mc', retval ) ;
    McReplace( 'Mac', retval ) ;
    McReplace( 'De''',retval ) ;
    McReplace( 'Du',retval ) ;
    return retval ;
end ;
```

Now we have two procedural bits of code that are *tightly coupled*. Briefly this means that either they are meaningless without each other or unable to work at all without each other. The *McReplace* procedure is useless on its' own and in a sane environment it would be best to prevent people seeing it and maybe using it for other purposes because then, if you wanted to change the implementation details, you may not be allowed to by others. Those of you who are very familiar with PL/SQL will be able to say that we could create a package and hide the *McReplace* procedure in that, just giving a public interface to the *McInitcap* function. There is an even easier way to do this:

```
create or replace function mcInitCap( p_name in varchar2 ) return varchar2 is
retval varchar2(50) ;
    procedure mcReplace( p_prefix in varchar2, p_name in out varchar2 ) is
    v_prefix_length number := length( p_prefix ) ;
    begin
        if instr( p_name, p_prefix ) > 0 then
            p_name := substr(p_name,1,v_prefix_length)
                      ||  initcap(substr(p_name,v_prefix_length+1,1))
                      ||  substr(p_name,v_prefix_length+2) ;
        end if ;
    end McReplace ;
begin
    retval := initcap( p_name ) ;
    McReplace( 'Mc', retval ) ;
    McReplace( 'Mac', retval ) ;
    McReplace( 'De'',retval ) ;
    McReplace( 'Du',retval ) ;
    return retval ;
end ;
```

We have factored out the repeated code and made it generic. The internals of the function are not public. This is a better solution, although a package would have done the job.

The above solution is still not correct, imagine what it would have done to the names Mace Macneice, Duval for example. You could modify the current solution to ignore anything that is smaller than the prefix+1 in length. The only final solution would be to have a table which will allow you to look up what the translation should be and even this may not work as the same spelling could be capitalised differently by different people. Once human beings get in to the equation life becomes harder.

Validation cursors: changing the standard model

The standard way to write a validation routine is as follows:

```
function f_item_exists(
       p_item_number in varchar2
,      p_org_id in number ) return boolean is
cursor c_check_item( c_item_number varchar2, c_org_id number ) is
select 1
from   mtl_system_items
where  segment1 = c_item_number
and    organization_id = c_org_id
;
v_testit number ;
begin
       open c_check_item( p_item_nunber, p_org_id ) ;
       fetch c_check_item into v_testit ;
       if c_check_item%notfound then
              close c_check_item ;
              return false ;
       end if ;
       close c_check_item ;
       return true ;
end f_item_exists ;
```

This code is not clean for the following reasons:

- It returns from two different places. If this were an anonymous block the return false would have been replaced with *raise form_trigger_failure* or the equivalent server side call to *raise_application_error*.
- It closes the cursor twice

Neither of these things is good, but you find code like this all over the place. Instead look at the following, cleaner example:

```
function f_item_exists(
        p_item_number in varchar2
,       p_org_id in number ) return boolean is
cursor c_check_item( c_item_number varchar2, c_org_id number ) is
select 1
from    mtl_system_items
where   segment1 = c_item_number
and     organization_id = c_org_id
;
v_testit number ;
b_foundit boolean ;
begin
        open c_check_item( p_item_nunber, p_org_id ) ;
        fetch c_check_item into v_testit ;
        b_foundit := c_check_item%found ;
        close c_check_item ;
        return b_foundit ;
end f_item_exists ;
```

This is much cleaner, it does everything it needs to do once. You may say that it will make little difference to you because the other code is not that much more complicated and easy to understand. Now suppose you have a somewhat artificial scenario where if you have to test the value of one item if another is first correct. Say we are looking at a cost type code and an inventory item:

```
function f_item_cost_exists(
        p_item_number in varchar2
,       p_org_id in number
,       p_cost_type in varchar2
) return boolean is
cursor c_check_item( c_item_number varchar2, c_org_id number ) is
select 1
from    mtl_system_items
where   segment1 = c_item_number
and     organization_id = c_org_id
;
cursor c_check_cost_type(  c_cost_type varchar2 ) is
select 1
from    cst_cost_types
where   cost_type = c_cost_type
;
v_testit number ;
begin
        open c_check_item( p_item_nunber, p_org_id ) ;
        fetch c_check_item into v_testit ;
        if c_check_item%notfound then
                close c_check_item ;
                return false ;
        end if ;
        open c_check_cost_type( p_cost_type ) ;
        fetch c_check_cost_type into v_testit ;
        if c_check_cost_type%notfound then ;
                close c_check_item ;
                close c_check_cost_type ;
                return false ;
        end if ;
        close c_check_item ;
        close c_check_cost_type ;
        return true ;
end f_item_cost_exists ;
```

Horrible! It closes the c_check_item cursor three times and the other twice. If you were checking three interdependencies then the situation will only get worse. Instead of carrying the open cursors to the end of the function close them straight away. Consider the following:

```
function f_item_cost_exists(
        p_item_number in varchar2
,       p_org_id in number
,       p_cost_type in varchar2
) return boolean is
cursor c_check_item( c_item_number varchar2, c_org_id number ) is
select 1
from    mtl_system_items
where   segment1 = c_item_number
and     organization_id = c_org_id
;
cursor c_check_cost_type(  c_cost_type varchar2 ) is
select 1
from    cst_cost_types
where   cost_type = c_cost_type
;
v_testit number ;
b_foundit boolean ;
begin
        open c_check_item( p_item_nunber, p_org_id ) ;
        fetch c_check_item into v_testit ;
        b_foundit := c_check_item%found ;
        close c_check_item ;
        if b_foundit then
                open c_check_cost_type( p_cost_type ) ;
                fetch c_check_cost_type into v_testit ;
                b_foundit := b_foundit and c_check_cost_type%found ;
                close c_check_cost_type ;
        end if ;
        return b_foundit ;
end f_item_cost_exists ;
```

This code is far cleaner and it does everything once. Additional tests can easily be added without causing any problem.

Simulating lexicals with *instr*

In the Oracle Reports tool you can use variables as parts of a query. For example, say you wanted to test for a list of item types depending upon what the user has entered you would like to put in something like:

```
select account_name
,       balance
from    accounts
where   account_type in ( &typelist )
```

Where the typelist variable will be a string that contains a user defined list that can change whenever the report is run. In Oracle reports whole pieces of SQL can be run by placing them in strings that can be inserted into the select statements that drive the report. You can change order by clauses, what is selected and add extra bits of SQL to the query.

In the other Oracle tools there is no such facility, and in Oracle Reports you cannot place & into PL/SQL blocks either. We can simulate the uses of lexicals by using the instr and decode functions. Decode is covered later.

Instr

Say you wish to be able to have a query that has a user-defined list of items as in the example earlier. Instr takes up to three arguments, the first one is the string to be searched, the second the string to look for and the third the number of times to find it. It will return the position in the string where it found the search string or zero if it did not. This changes the example to the following:

```
select account_name
,       balance
from    accounts
where   instr( :typelist, account_type ) > 0 )
```

This is obviously less efficient because it applies a function to a table column which means that it will not use indexes. It is functionally equivalent to the earlier statement provided that the account type codes are all the same length because instr('0003, 00031', account_type) will match account_type codes of '03' as well as '003', '0003', '0030' etc. and all of the permutations of '31', which is not what was intended. This does mean that you can use instr to simulate a *like* clause but as *like* uses indexes this is probably not a good idea in select statements. It does mean that you can simulate a *like* clause in an if statement, though, which is very useful.

Comment: Could also create a function f_inlist(list, target) which does parse long handed.

The Decode function

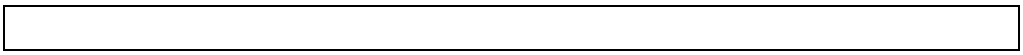
The decode function is one of the most powerful parts of the Oracle version of SQL. It is basically an if statement hidden in a function. It takes arguments of the form:

```
decode( accounts_type, 'A',6,'B',7,8)
```

This means that decode is to take the value in accounts_type and return 6 if it is A, 7 if it is B and 8 in all other cases. You can also test for null in the list and do something more than the standard nvl would do. Say that your table looks like this:

```
create table account_values
(       account_code    varchar2(20)    not null - compound key with region
,       region          varchar2(10)    not null
,       balance         number          not null
)
```

If you wanted to create a report of the form



You can create a report

CASE STUDY 2 : WRITING A SERVER-SIDE JOB HANDLER

Requirements

The requirements hwere are fairly standard. Usually it it not good to run reports locally on a client machine. This is because the machine becomes tied up running the report. It will also take longer to run the report because of the transfer of data between the client and the server. It would also be unusual if the power of the client machine was anywhere near that of the server. In many pre client/server systems it was good practice to place jobs in the background anyway to allow the user to continue doing something else.

Requirements analysis

First cut list of tables

Programs

Requests

Printers

Printer Definitions

(later)Parameter sets

First cut list of programs

submit

get status

runjob

(later)resubmit

scanjob

Further requirements

Hierarchies

Queues

Looping jobs

Obvious enhancements

Flaws

Security

IDEAS B : HOW TO WRITE MODULES

Factoring

This is the opaque art of design and coding. It is a simple concept. All that is required is the identification of discrete modules within an overall application and then subprograms within them. There are books about how to do this and they all present nice case studies but I contend that it is still a opaque art.

Modular modules

Forms model

Different approaches

Reusability

Objects and modules

HINTS & TIPS II: DEBUGGING SERVER-SIDE PL/SQL

Maintenance and Defence

Maintenance is the most difficult part of any job. The excitement of writing a new system gives way to the boredom of enhancing and fixing it. In order to write maintainable code it must be written to a standard. This means that there should be a standard layout and a standard way of handling errors. A new programmer feels that he or she is being imposed upon because layout is a personal thing but this is nonsense. A good set of layout and variable naming standards mean that you can get on with writing the program without having to worry about all that irrelevant stuff.

Closure

If you code defensively then you are making sure that all of the options have been taken care of. At a simple level this means always putting an else after every if. Certain languages like PL/SQL, because of its heritage from Ada, also have defensive elements built in to them. This is why an index loop variable is only scoped within that loop. The code is naturally block structured where if has end if and loop has end loop. This is in stark contrast to such overrated languages as C where a closing brace could be closing any control structure you like or indeed none.

Put_line

Debugging package using pipes

local Globals

CASE STUDY 3: ORACLE FORMS QUICK START

Introduction

Not a case study as such but a look at design considerations

Standards

Look & feel

Forms L&F not the same as windows standard but is standard across all platforms

Identifying libraries

The art of identifying reusable

Standard interface objects

Use classes for consistency

Calendar, standard lists

Buy in code QMS qs

Client server traffic

Smart client concepts

The server does the work not the client.

Ordering by joined columns easy. Surrogate keys (explained later) make ordering rather difficult if you *don't* use this model.

Object orientation

Procedures as objects, virtual tables the next wave

NC

Ultra-thin three tier model. Smart client still a good idea

HINTS & TIPS III : UNDERSTANDING SQL PROPERLY

Common sense SQL

Basic relational theory. Understanding select, project and join. What does the database try to do to meet your requirements. A query is give me the set where this set of conditions are true and united with another set. How a query is broken down (in an abstract model, not Oracle). Reasons behind combinatorial explosion and sum() etc. Not always giving sensible answers. Easy to understand if you have the model in your head.

Pure relational model always has a proper key to make rows distinct.

Surrogate keys - how this effects forms

CASE STUDY 4 : BUILDING A SIMPLE FAIL-OVER SYSTEM

Requirements

Requirements analysis

CASE STUDY 5 : BATCH LOADING DATA INTO TABLES

Requirements

User configurable data take on into a table or groups of tables

Dynamic configuration of take on

Requirements analysis

HINTS & TIPS V : FUN WITH DYNAMIC SQL

CASE STUDY 6 : WHY DID A CONFORMANT SYSTEM FAIL?

BITS THAT NEED TO GO SOMEWHERE

Version control

Creative use of editors

SQL*Loader tricks, getting item id’s where values are supplied elsewhere