PL/SQL Coding Style Guidelines Introducing Error Handling, Tracing and Coding Standards

Author: Vishal Gupta

Introduction

Presenter:

- Vishal Gupta

Topics:

- Reasons for guidelines
- Standards for error handling
- Standards for trace handling
- Header comments and versions
- Questions and answers

Reasons for guidelines

Advantages of using coding guidelines:

- **Uniform look and feel of all of the source code**
- **Framework can be used for implementing new modules**
- **Faster ramp-up time for new developers**
- **Proven methods and structures are reused which leads to more stable code**

Standards for Error Handling

All PL/SQL code has to conform to the error handling standards

- Error handler in each procedure/function.
- In large procedures multiple error handlers should be defined
- Secondary error handlers should be defined in case an error can occur inside of an error handler
- Procedures should forward errors to the calling procedure
- Standard error_struct (structure) should be used for error handling
- Each error that is raised in code should be documented in configuration table
- Documentation and example

Categorization of Error Handling

All Defined errors should be categorized as follows:

- I = Info (These are just for information level Process continues in regular flow)
- W = Warning (Warning Exceptions/Logs should work like to handle certain situations of processing steps these are not Business Critical and process continues)
- E = Error (Any Exception that reports Application Error Should Terminate processing)
- F = Fatal error (Any Undefined Exception, that is trapped and handled external to application using WHEN OTHERS should be recorded as FATAL error and should terminate the execution of process)

Error Handling Cont…

Error Handling Configuration will look like as:

FIELD_NAME	EXAMPLE	DESCRIPTION
ERROR_CODE	DROP-000001	This is the unique identifier of Error Code referenced in code exception
ERROR_TYPE	I, W, E, F	I = Info, W = Warning, E = Error, F = Fatal error
ERROR_TEXT	No data found in <%s1>!	Where “No data found in” is generic text and %s1 will be replaced by parameters in generic error handling package passed from the code.
USER_CREATTED	123	ID of the application user who created this record should references to user tables
CREATE_DATE	12/12/2008 20.12.00PM	Date time record created

Error Dependencies:

- Primary errors (P) — This is the error that should occurs at first. It gets a 'Master-ID'.
- Secondary errors (S) — These are errors that happen in any kind of cleanup or exception handling of a primary error. This error should reference to the 'Master-ID'.
- Forwarded errors (F) — These are errors that should be just passed through the exception handlers on the way up to the caller (in most of the cases the application).

Example

ID	Master-ID	Kind of Error
1	1	P
2	2	P
3	2	F
4	2	F
5	5	P
6	5	S
7	5	S
8	5	F
9	5	F
10	5	F

Key information recorded for Error Log:

MACHINE
OS_USER
SESSION_USER
DATE
MODULE
FUNCTION
ORIGINATION_TYPE
ERROR_CODE
PARAMETER_LIST
CONTEXT

Avoid hard-coding of -20,NNN Errors

```
PACKAGE errnums
IS
   en_general_error CONSTANT NUMBER := -20000;
   exc_general_error EXCEPTION;
   PRAGMA EXCEPTION_INIT
     (exc_general_error, -20000);

   en_must_be_18 CONSTANT NUMBER := -20001;
   exc_must_be_18 EXCEPTION;
   PRAGMA EXCEPTION_INIT
     (exc_must_be_18, -20001);

   en_sal_too_low CONSTANT NUMBER := -20002;
   exc_sal_too_low EXCEPTION;
   PRAGMA EXCEPTION_INIT
     (exc_sal_too_low , -20002);

   max_error_used CONSTANT NUMBER := -20002;

END errnums;
```

Should give numbers names and associate them with named exceptions.

Using the standard raise program

Rather than have individual programmers call RAISE_APPLICATION_ERROR, simply call the standard raise program.
Benefits:

- Easier to avoid hard-coding of numbers.
- Support positive error numbers!

Raising errors by name

Use an error name (literal value).

- The code compiles *now*.
- Later, define that error in the repository.
- No central point of failure.

Downsides: risk of typos, runtime notification of "undefined error."

Summary: an Exception Handling Architecture

Make sure it is understood how it all works

- Exception handling is tricky stuff

Set standards before start coding

- It's not the kind of thing can easily added in later stages

Use standard infrastructure components

- Everyone and all programs need to handle errors the same way

Take full advantage of error management features.

- SAVE EXCEPTIONS, DBMS_ERRLOG, DBMS_UTILITY.FORMAT_ERROR_BACKTRACE...

Don't accept the limitations of Oracle's current implementation.

- lots can be done to improve the situation.

Trace Handling

Tracing should be used for tracking error situations and process analysis during development (But can be activated in production on need from configuration tables). It should be implemented with in PL/SQL packages and has to be used in a strong defined way in every other PL/SQL package/function or procedure. Hence It should be highly configurable and easy to use.

All objects belonging to the tracing module need to be placed in a common schema

Standards for Trace Handling

All PL/SQL code has to conform to the trace handling standards

- Trace handling using the Tracing package
- As a minimum trace of procedure entry/exit
- Use appropriate trace level for the trace statement being implemented
- Configuring the trace setup (What, When, & Module piece of code need to be traced, at what level and from which machine/host)
- Reviewing the trace output (Table, File)
- Documentation and example

Levels of Trace Handling

Definition of Trace Levels:
Trace levels can be defined in a range from 0 to 8, where 0 is the same as "No Trace"
Trace should be used using pre-defined trace levels for the following parts of the application.

- *Level 1*: has to be set in any case of an error, this goes hand in hand with the call of the error handling functions
- *Level 2*: at the beginning and end of every procedure or function
  - trace2('Entering procedure my_proc');
  - trace2('Leaving procedure my_proc');
- *Level 3*: list of procedure parameters
  - trace3('p_param1: '||p_param1);
  - trace3('p_param2: '||p_param2);
- *Level 4*: any local variables
  - trace4('v_start_seq: '||v_start_seq);
  - trace4('v_datestamp: '||to_char(v_datestamp,
  - 'mm.dd.yyyy hh24:mi'));
- *Level 5*: before and after calling a sub-procedure
  - trace5('calling my_proc');
  - my_proc(4711, sysdate, p_error_rec);
  - trace5('return from my_proc');

Levels of Trace Handling

- *Level 6*: variables inside loops with depth 1
  - loop
  - trace6('loop nr. '||to_char(v_loop_counter));
  - <do something>
  - v_loop_counter := v_loop_counter+1;
  - if (v_loop_counter > 100)
  - then
  - trace6('loop finished');
  - end if;
  - end loop;
- *Level 7*: variables inside loops with depth 2
- *Level 8*: variables inside loops with depth 3

Header comments and versions

All PL/SQL packages, functions and procedures have to contain a standard header

- Package header with copyright information
- Function and procedure header contains:
  - Description and parameter list definition
  - Revision history including author, date, CR, change description
- Documentation and examples

Bottleneck of traditional Exception Handling

Hard to avoid code repetition in handlers

```
WHEN NO_DATA_FOUND THEN
   INSERT INTO errlog
     VALUES ( SQLCODE
            , 'No company for id ' || TO_CHAR ( v_id )
            , 'fixdebt', SYSDATE, USER );
WHEN OTHERS THEN
   INSERT INTO errlog
     VALUES (SQLCODE, SQLERRM, 'fixdebt', SYSDATE, USER );
   RAISE;
END;
```

If every developer writes exception handler code on their own, you end up with an unmanageable situation.

- Different logging mechanisms, no standards for error message text, inconsistent handling of the same errors, etc.

Questions?

Authors

This presentation was prepared by:

Vishal Gupta,
10200 Park Meadows

Dr. Lone Tree CO, 80124
Tel: +1.646.203.3520
E-mail: gupt.vishal@gmail.com

Recommendations

- **Introduce PL/SQL Coding Guidelines**