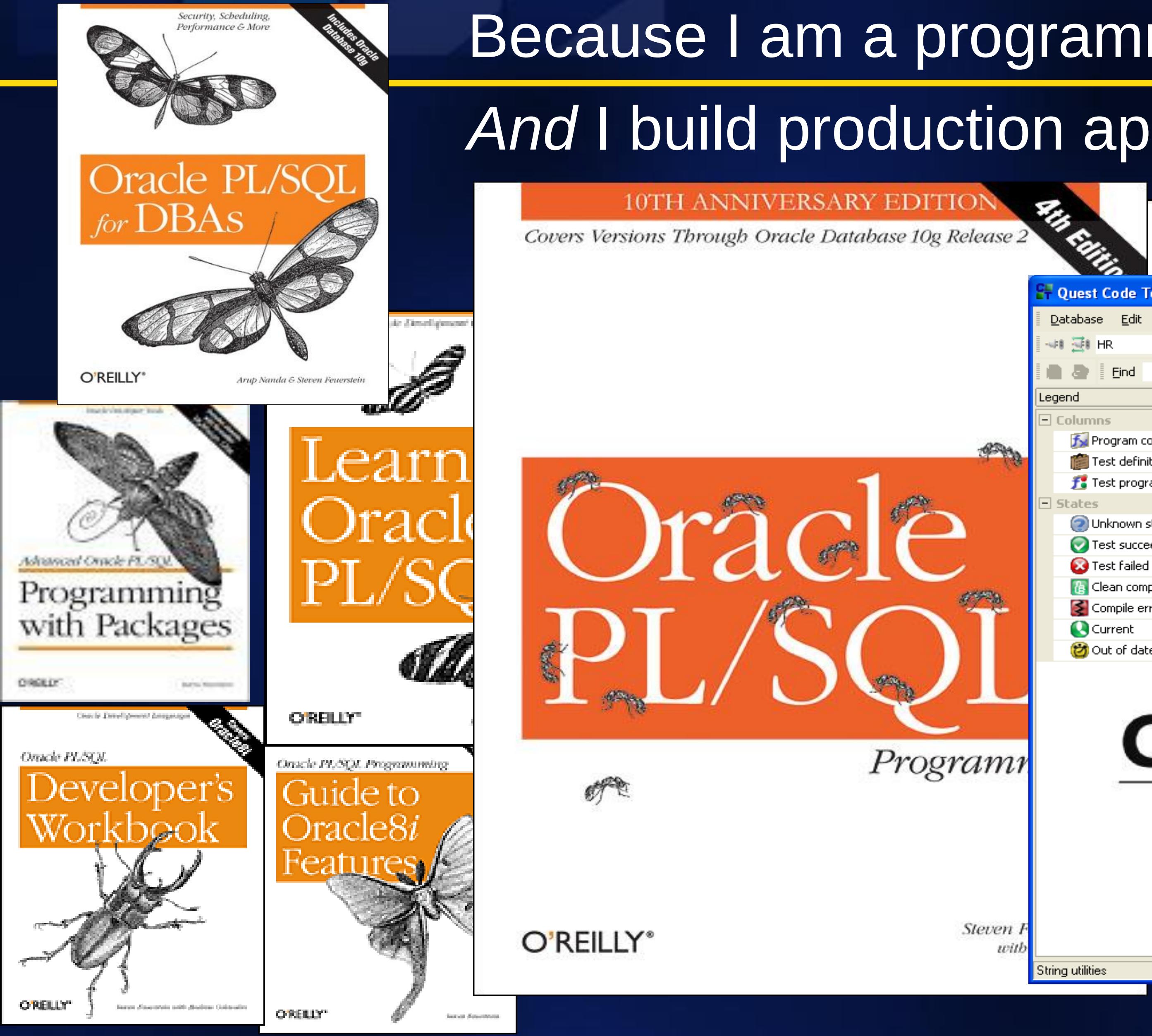
Making the Most of Oracle PL/SQL Error Management Features

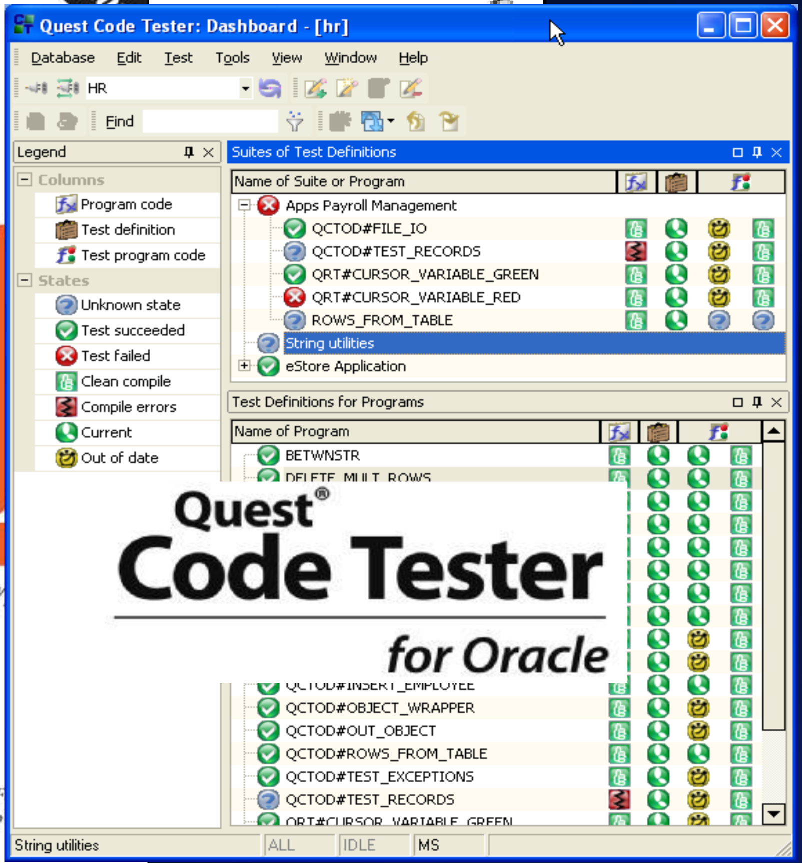
Steven Feuerstein
PL/SQL Evangelist
Quest Software
steven.feuerstein@quest.com

So...why listen to *me*?

Because I am a programmer obsessed...

And I build production applications....


Security, Scheduling, Performance & More
Includes Oracle Database 10g
Oracle PL/SQL for DBAs
O'REILLY
Arup Nanda & Steven Feuerstein
Advanced Oracle PL/SQL
Programming with Packages
Learn Oracle PL/SQL
10TH ANNIVERSARY EDITION
Covers Versions Through Oracle Database 10g Release 2
4th Edition
Oracle PL/SQL Programming
Steven F
Oracle PL/SQL
Developer's Workbook
Oracle PL/SQL Programming
Guide to Oracle8i Features


Quest Code Tester: Dashboard - [hr]
Database Edit Test Tools View Window Help
HR
Find
Quest® Code Tester for Oracle

Legend

- Columns
  - Program code
  - Test definition
  - Test program code
- States
  - Unknown state
  - Test succeeded
  - Test failed
  - Clean compile
  - Compile errors
  - Current
  - Out of date

Suites of Test Definitions

Name of Suite or Program

- Apps Payroll Management
  - QCTOD#FILE_IO
  - QCTOD#TEST_RECORDS
  - QRT#CURSOR_VARIABLE_GREEN
  - QRT#CURSOR_VARIABLE_RED
  - ROWS_FROM_TABLE
- String utilities
- eStore Application

Test Definitions for Programs

Name of Program

- BETWNSTR
- DELETE_MULT_ROWS
- QCTOD#INSERT_EMPLOYEE
- QCTOD#OBJECT_WRAPPER
- QCTOD#OUT_OBJECT
- QCTOD#ROWS_FROM_TABLE
- QCTOD#TEST_EXCEPTIONS
- QCTOD#TEST_RECORDS
- QRT#CURSOR_VARIABLE_GREEN

String utilities | ALL | IDLE | MS

How to benefit most from this session

- **Watch, listen, *ask questions*. Then afterwards....**
- **Download and use any of my the training materials, available at my "cyber home" on Toad World, a portal for Toad Users and PL/SQL developers:**

PL/SQL Obsession **http://www.ToadWorld.com/SF**

- **Download and use any of my scripts (examples, performance scripts, reusable code) from the demo.zip, available from the same place.**

filename_from_demo_zip.sql

- **You have my permission to use *all* these materials to do internal trainings and build your own applications.**
  - **But they should not considered production ready.**
  - **You must test them and modify them to fit your needs.**

Manage errors effectively and consistently

- A significant challenge in any programming environment.
  - Ideally, errors are raised, handled, logged and communicated in a consistent, robust manner
- Some special issues for PL/SQL developers
  - The EXCEPTION datatype
  - How to find the line on which the error is raised?
  - Communication with non-PL/SQL host environments

Achieving ideal error management

- Define your requirements clearly
- Understand PL/SQL error management features and make full use of what PL/SQL has to offer
- Apply best practices.
  - **Compensate for PL/SQL weaknesses**
  - **Single point of definition: use reusable components to ensure consistent, robust error management**

PL/SQL error management features

- Defining exceptions
- Raising exceptions
- Handing exceptions
- Exceptions and DML

Quiz: Test your exception handling know-how

```
PACKAGE valerr
IS
  FUNCTION
    get RETURN VARCHAR2;
END valerr;

PACKAGE BODY valerr
IS
   v VARCHAR2(1) := ‘abc’;
   FUNCTION get RETURN VARCHAR2 IS
   BEGIN
      RETURN v;
   END;
BEGIN
   DBMS_OUTPUT.PUT_LINE (
     'Before I show you v...');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE (
     ‘Trapped the error!’);
END valerr;
```

```
SQL> EXECUTE p.l (valerr.get);
```

- I create the valerr package and then execute the command below. What is displayed on the screen?
- Key to remember: even if package initialization fails, Oracle marks the package as *initialized*.

valerr.pkg
valerr2.pkg

Defining Exceptions

- The EXCEPTION is a limited type of data.
  - **Has just two attributes: code and message.**
  - **You can RAISE and handle an exception, but it cannot be passed as an argument in a program.**
- Give names to error numbers with the EXCEPTION_INIT PRAGMA.

```
CREATE OR REPLACE PROCEDURE upd_for_dept (
   dept_in    IN   employee.department_id%TYPE
 , newsal_in IN   employee.salary%TYPE
)
IS
   bulk_errors   EXCEPTION;
   PRAGMA EXCEPTION_INIT (bulk_errors, -24381);
```

Raising Exceptions

- RAISE raises the specified exception by name.
  - RAISE; re-raises current exception. Callable only within the exception section.
- RAISE_APPLICATION_ERROR
  - Communicates an application specific error back to a non-PL/SQL host environment.
  - Error numbers restricted to the -20,999 - -20,000 range.

Using RAISE_APPLICATION_ERROR

```
RAISE_APPLICATION_ERROR
   (num binary_integer, msg varchar2,
    keeperrorstack boolean default FALSE);
```

- Communicate an error number and message to a non-PL/SQL host environment.
  - **The following code from a database triggers shows a typical (and problematic) usage of RAISE_APPLICATION_ERROR:**

```
IF :NEW.birthdate > ADD_MONTHS (SYSDATE, -1 * 18 * 12)
THEN
   RAISE_APPLICATION_ERROR
      (-20070, ‘Employee must be 18.’);
END IF;
```

Handling Exceptions

- The EXCEPTION section consolidates all error handling logic in a block.
  - **But only traps errors raised in the executable section of the block.**
- Several useful functions usually come into play:
  - **SQLCODE and SQLERRM**
  - **DBMS_UTILITY.FORMAT_ERROR_STACK**
  - **DBMS_UTILITY.FORMAT_ERROR_BACKTRACE**
- The DBMS_ERRLOG package
  - **Quick and easy logging of DML errors**
- The AFTER SERVERERROR trigger
  - **Instance-wide error handling**

DBMS_UTILITY error-related functions

- DBMS_UTILITY.FORMAT_CALL_STACK answers the question: "How did I get here?"
- Get the full error message with DBMS_UTILITY.FORMAT_ERROR_STACK
  - **SQLERRM might truncate the message.**
  - **Use SQLERRM went you want to obtain the message associated with an error number.**
- Find line number on which error was raised with DBMS_UTILITY.FORMAT_ERROR_BACKTRACE
  - **Introduced in Oracle10g Release 2, it returns the full stack of errors with line number information.**
  - **Formerly, this stack was available only if you let the error go unhandled.**

More on the BACKTRACE function

- When you re-raise your exception (RAISE;) or raise a different exception, subsequent BACKTRACE calls will point to *that* line.
  - **So before a re-raise, call BACKTRACE and store that information to avoid losing the original line number.**
- The BACKTRACE does not include the error message, so you will also want to call the FORMAT_ERROR_STACK function as well.

backtrace.sql
bt.pkg

DBMS_ERRLOG

- Allows DML statements to execute against all rows, even if an error occurs.
  - The LOG ERRORS clause specifies how logging should occur.
  - Use the DBMS_ERRLOG package to associate a log table with DML operations on a base table.
- Much faster than trapping errors, logging, and then continuing/recovering.
- Consider using LOG ERRORS with FORALL (instead of SAVE EXCEPTIONS) so that you can obtain all error information!
  - But there are some differences in behavior.

dbms_errlog.*
dbms_errlog_helper.sql
save_exc_vc_dbms_errlog.sql
cfl_to_bulk7.sql

The AFTER SERVERERROR trigger

- Provides a relatively simple way to use a single table and single procedure for exception handling in an entire instance.
- Drawbacks:
  - **Error must go unhandled out of your PL/SQL block for the trigger to kick in.**
  - **Does not fire for all errors (NO: -600, -1403, -1422...)**
- Most useful for non-PL/SQL front ends executing SQL statements directly.

afterservererror.sql

Exceptions and DML

- DML statements generally are *not* rolled back when an exception is raised.
  - **This gives you more control over your transaction.**
- Rollbacks occur with...
  - **Unhandled exception from the outermost PL/SQL block;**
  - **Exit from autonomous transaction without commit/rollback;**
  - **Other serious errors, such as "Rollback segment too small".**
- Corollary: error logs should rely on autonomous transactions to avoid sharing the same transaction as the application.
  - **Log information is committed, while leaving the business transaction unresolved.**

log8i.pkg

Best practices for error management

- Compensate for PL/SQL weaknesses.
- Avoid hard-coding of error numbers and messages.
- Application-level code should not contain:
  - **RAISE_APPLICATION_ERROR: don't leave it to the developer to decide *how* to raise.**
  - **PRAGMA EXCEPTION_INIT: avoid duplication of error definitions.**
- Build and use shared components for raising, handling and logging errors.

Compensate for PL/SQL weaknesses

- The EXCEPTION datatype does not allow you to store the full set of information about an error.
  - **What was the context in which the error occurred?**
- Difficult to ensure execution of common error handling logic.
  - **Usually end up with lots of repetition.**
  - **No "finally" section available in PL/SQL - yet.**
- Restrictions on how you can specify the error
  - **Only 1000 for application-specific errors....**

Addressing the limitations of EXCEPTION

- When an error occurs....
  - **Sure, it's nice to know what the error code is.**
  - **But what I care most about is what *caused* this particular error to be raised.**
- Think in terms of *instances* of an error.
  - **What caused this error?**
  - **What were the application-specific values or context in which the error occurred?**
- The challenge becomes: how do I get and save all that critical application information?

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 everyone writes their own exception handler code, you end up with an unmanageable situation.**
  - **Different logging mechanisms, no standards for error message text, inconsistent handling of the same errors, etc.**

"Proof of concept" exception manager package

```
PACKAGE errpkg
IS
   PROCEDURE raise (err_in IN PLS_INTEGER);
   PROCEDURE raise (err_in in VARCHAR2);

   PROCEDURE record_and_stop (
      err_in IN PLS_INTEGER := SQLCODE
     ,msg_in IN VARCHAR2 := NULL);


   PROCEDURE record_and_continue (
      err_in IN PLS_INTEGER := SQLCODE
     ,msg_in IN VARCHAR2 := NULL);

END errpkg;
```

Raise the exception *for* you

Record and Stop

Record and Continue

errpkg.pkg

Invoking standard handlers

- Developers should call *only* a pre-defined handler inside an exception section.
  - **Much easier to write consistent, high-quality code**
  - **They don't have to make decisions about the form of the log and how the process should be stopped**

```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      errpkg.record_and_continue (
         SQLCODE,
         ' No company for id ' || TO_CHAR (v_id));

   WHEN OTHERS THEN
      errpkg.record_and_stop;
END;
```

The developer simply *describes* the desired action.

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;
```

- Give your error numbers names and associate them with named exceptions.

But don't write this code manually!

msginfo.pkg
msginfo.fmb/fmx

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-codings of numbers.**
  - **Support positive error numbers!**
- Let's revisit that earlier trigger logic using the error manager and related elements...

```
PROCEDURE validate_emp (birthdate_in IN DATE) IS
BEGIN
   IF ADD_MONTHS (SYSDATE, 18 * 12 * -1) < birthdate_in
   THEN
      errpkg.raise (errnums.en_too_young);
   END IF;
END;
```

No more hard-coded strings or numbers.

From proof of concept to real code

- One option: the Quest Error Manager, which you can download from PL/SQL Obsession.
- Offers a simple API to....
  - Raise, handle, log errors
  - Traces application execution and enhances DBMS_OUTPUT.PUT_LINE
  - Assert conditions
- Addresses the limitations of EXCEPTION.

QEM deals with *instances* of exceptions

- An error is a row in the error table, with many more attributes than simply code and message, including:
  - **Dynamic message (substitution variables)**
  - **Help message (how to recover from the problem)**
- An error instance is one particular occurrence of an error.
  - **Associated with it are one or more values that reflect the context in which the error was raised.**

The Quest Error Manager API

- High-level API for all error mgt operations:
  - **REGISTER_ERROR: register the fact that an error has occurred, retrieve an error instance handle.**
  - **RAISE_ERROR: Register the error, and then re-raise the exception to stop the calling program from continuing.**
  - **ADD_CONTEXT: Add unlimited number of name-value pairs to an error instance.**
  - **GET_ERROR_INFO: Retrieve information about latest (or specified) error.**

qem_demo*.sql

Summary on error management in PL/SQL

- Make sure you understand how it all works
  - **Exception handling is tricky stuff**
- Set standards before you start coding
  - **It's not the kind of thing you can easily add in later**
- 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.
  - **You can do lots to improve the situation.**