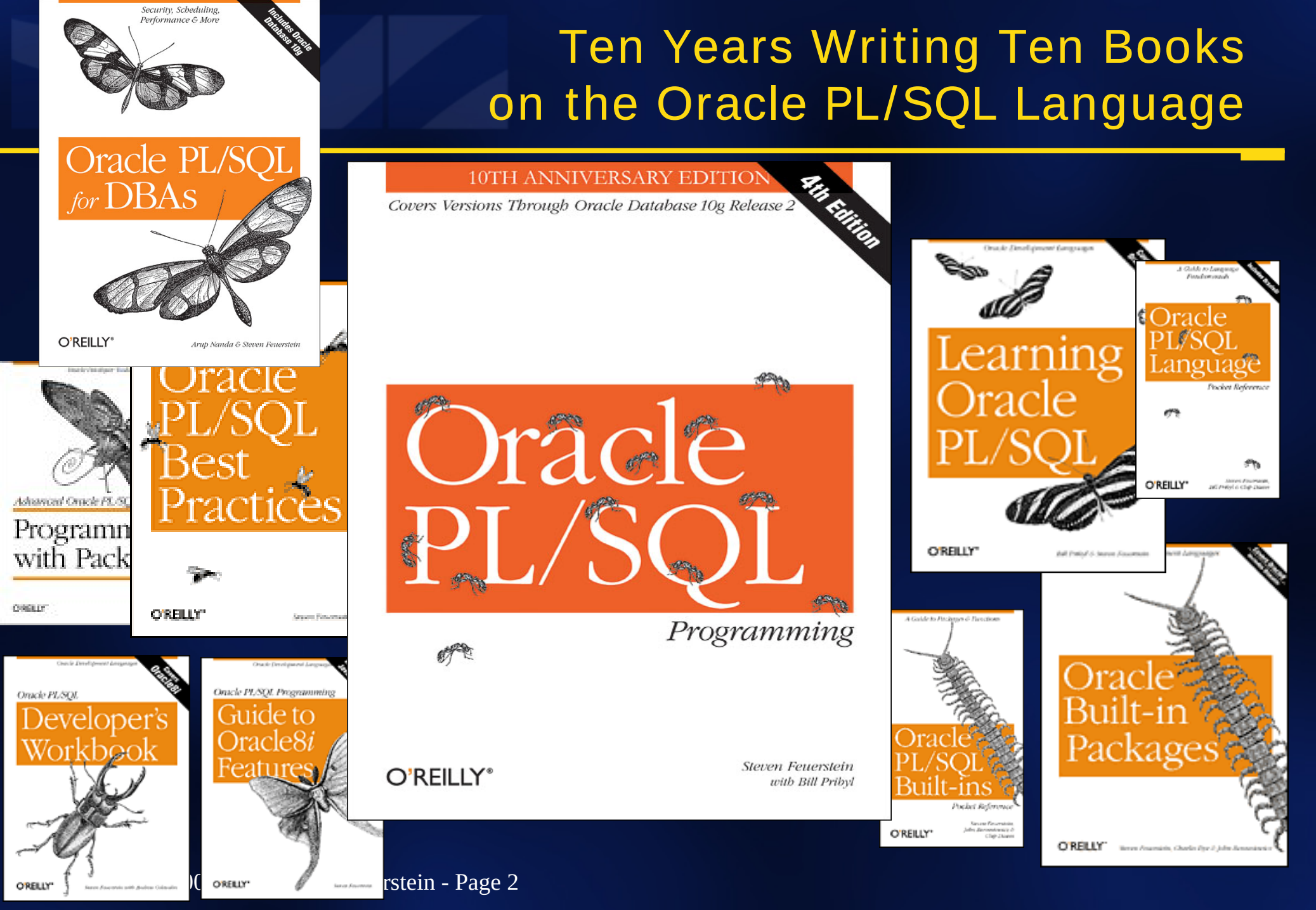
Best Practice PL/SQL

Making the Best Use of the Best Features of Oracle PL/SQL

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

Ten Years Writing Ten Books on the Oracle PL/SQL Language



How to benefit most from this class

- Watch, listen, *ask questions*.
- Download the training materials and supporting scripts:
  - http://oracleplsqlprogramming.com/resources.html
  - "Demo zip": all the scripts I run in my class available at http://oracleplsqlprogramming.com/downloads/demo.zip

filename_from_demo_zip.sql

- Use these materials as an accelerator as you venture into new territory and need to apply new techniques.
- Play games! Keep your brain fresh and active by mixing hard work with challenging games
  - MasterMind and Set (www.setgame.com)

Critical elements of PL/SQL Best Practices

- Build your development toolbox
- Unit test PL/SQL programs
- Optimize SQL in PL/SQL programs
- Manage errors effectively and consistently
- Write readable, maintainable code

High Level, High Impact

Important principles...

- Assume everything will change.
- Aim for a single point of definition (a SPOD).

Top four tips....

- Drink lots of water.
- Write tiny chunks of code.
- Stop writing so much SQL.
- Stop guessing and start testing.

Drink lots of water!

- And lots less coffee.
- OK, don't go cold turkey on caffeine.
- But drink lots (and lots more) water.
  - Coffee dehydrates you and a dehydrated brain just doesn’t work as effectively.
- Generally, we need to take care of our host body, so that our brain can keep on earning that big fat paycheck!
  - Get away from your computer, take breaks.
  - Exercise and stretch.

> Build your development toolbox

- You need first and foremost a powerful IDE.
  - There are many to choose from, varying greatly in price and functionality.
- Other useful tools...
  - Avoid writing code; instead rely on code generation, reusable libraries, etc.
  - Test your code using unit testing tools.
- And some useful utilities...
  - Performance analysis/comparison
  - Memory usage analysis

Quest CodeGen Utility
Quest Code Tester

Performance Analysis and Comparison

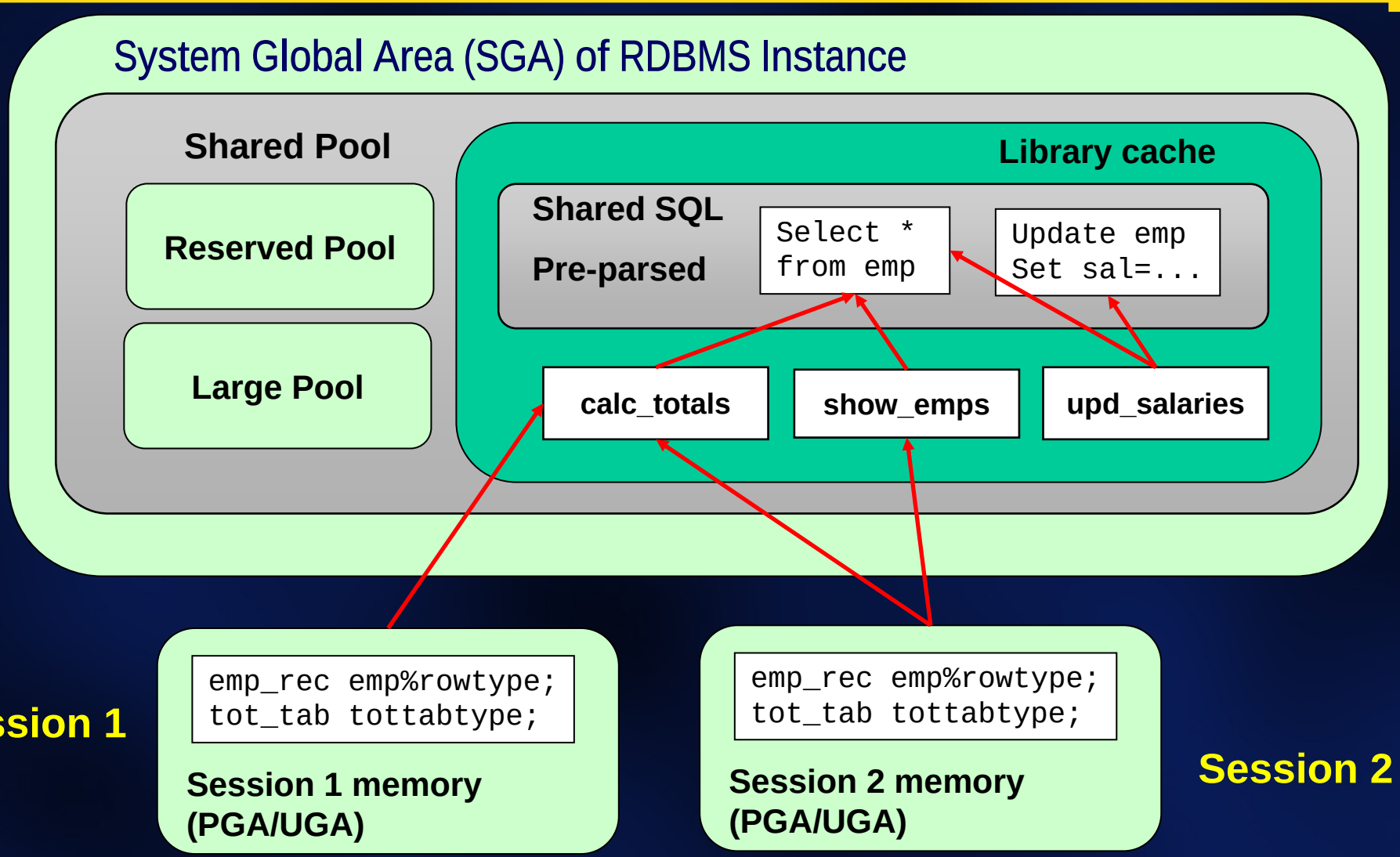
- Several options are available...
  - TKPROF
  - SQL*Plus SET TIMING ON
  - DBMS_UTILITY.GET_TIME/GET_CPU_TIME
  - SYSTIMESTAMP
- The seminar "demo zip" offers several encapsulations of a DBMS_UTILITY-based performance analysis script.
  - DBMS_UTILITY.GET_CPU_TIME helps you answer that question down to the 100th of a second.

tmr*.ot
plvtmr.pkg
systimestamp_elapsed.sql
thisuser.*
emplu.pkg

Memory usage analysis

- Complex data structures (collections, objects, records) can take up substantial amounts of memory.
- You should be aware of the issue of memory consumption, know how to analyze memory usage, and adjust usage as needed.
- Let's review memory architecture and then examine how you can do your own analysis.

Refresher: PL/SQL in Shared Memory



Analyze and manage memory consumption

- Analyze through v$ dynamic views
  - Obtain "session pga memory" information from v_$sesstat.

mysess.sql
show_memory.sp
analyze_memory.sql
memory_analysis.sql

- Elements of PL/SQL that affect memory usage:
  - BULK COLLECT limit clause
  - The NOCOPY hint
  - Packaged variables
  - DBMS_SESSION programs: free_unused_user_memory, reset_package and modify_package_state

bulklimit.sql
nocopy*.*
emplu.*

> Unit test PL/SQL programs or....

Six Simple Steps to Unit Testing Happiness

Writing software is.....

Testing software is.....

Buggy software is....

- Embarrassing
- Expensive
- Deadly

Buggy software is embarrassing

- There can be as many as 20 to 30 bugs per 1,000 lines of software code. —Sustainable Computing Consortium
- 32% of organizations say that they release software with too many defects.—Cutter Consortium
- 38% of organizations believe they lack an adequate software quality assurance program.—Cutter Consortium
- 27% of organizations do not conduct any formal quality reviews.—Cutter Consortium
- Developers spend about 80% of development costs on identifying and correcting defects.—The National Institute of Standards and Technology

Buggy software is expensive - $60B per year in US alone!?

- **JUNE 25, 2002 (COMPUTERWORLD) - WASHINGTON -- Software bugs are costing the U.S. economy an estimated $59.5 billion each year. Of the total $59.5 billion cost, users incurred 64% of the cost and developers 36%.**
- **There are very few markets where "buyers are willing to accept products that they know are going to malfunction," said Gregory Tassey, the National Institute of Standards and Technology senior economist who headed the study. "But software is at the extreme end in terms of errors or bugs that are in the typical product when it is sold."**
- **Oh, yes and Y2K: $300B? $600B?**

Buggy software is deadly

- **2003** Software failure contributes to power outage across the Northeastern U.S. and Canada, killing 3 people.
- **2001** Five Panamanian cancer patients die following overdoses of radiation, amounts of which were determined by faulty use of software.
- **2000** Crash of a Marine Corps Osprey tilt-rotor aircraft partially blamed on “software anomaly" kills four soldiers.
- **1997** Radar that could have prevented Korean jet crash (killing 225) hobbled by software problem.
- **1995** American Airlines jet, descending into Cali, Colombia, crashes into a mountain, killing 159. Jury holds maker of flight-management system 17% responsible. A report by the University of Bielefeld in Germany found that the software presented insufficient and conflicting information to the pilots, who got lost.

How do we avoid buggy software?

- Clear and accurate requirements
- Careful design
- Excellent tools
- Best practices, standards, guidelines (that is, follow them)
- Code review
- Thorough testing

Wouldn't it be great if...

- It was easy to construct tests
  - An agreed-upon and effective approach to test construction that everyone can understand and follow
- It was easy to run tests
  - And see the results, instantly and automatically.
- Testing were completely integrated into my development, QA, and maintenance processes
  - No program goes to QA until it passes its unit tests
  - Anyone can maintain with confidence, because my test suite automatically validates my changes

Different types of testing

- There are many types of testing: functional/system tests, stress tests, unit tests.
- A "unit test" is the test of a single unit of code.
  - Also known as "programmer tests"
- Unit tests are the responsibility of developers - that is, us, the people in this room.
  - Not fundamentally a job for the QA department, which generally focuses on functional and system tests.

Truth or Dare

- How do you (or your team) *unit test* your PL/SQL code today?

? – We use automated testing software.

? – We have a formal test process that we each follow, but otherwise a manual process.

? – Everyone does their own thing and we hope for the best.

? – Our users test our code.

Unit testing reality

- Let's face it: we PL/SQL developers don't spend nearly enough time unit testing our code.
  - For the most part, we run a script that displays output on the screen and then we stare at all until we decide if the test succeeded or failed.
- There are some understandable reasons:
  - Very few tools and utilities have been available, *to date*, for PL/SQL testing.
  - Managers don't give us enough time to prepare and execute tests.

Typical Testing

- DBMS_OUTPUT.PUT_LINE - unit testing mechanism of choice?

```
BEGIN
   DBMS_OUTPUT.PUT_LINE (betwnstr (NULL, 3, 5, true));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefgh', 0, 5, true));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefgh', 3, 5, true));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefgh', -3, -5, true));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefgh', NULL, 5, true));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefgh', 3, NULL, true));
   DBMS_OUTPUT.PUT_LINE (betwnstr ('abcdefgh', 3, 100, true));
END;
```

betwnstr.sf
betwnstr.tst

Problems with Typical Testing

- Almost entirely ad hoc
  - No comprehensive effort to compile test cases
  - No infrastructure to record cases and administer tests
- Difficult to verify correctness
  - Non-automated verification is slow and error-prone.
- Relies on the user community to test
  - Since we are never really sure we've tested properly, we rely on our users (or, we are lucky, the QA department) to finish our job

There has got to be a better way!

Moving towards a Better Way

- Change from within: your code will not test itself.
  - You must accept the responsibility and then be disciplined (sigh...that's not fun at all).
  - Commit to testing and watch the way you write your code change.
- Change from without: new possibilities are on the horizon!
  - utPLSQL
  - Quest Code Tester for Oracle
- Ah, but what about those six, simple steps?

http://utplsql.sourceforge.net/

http://www.ToadWorld.com

Six Simple Steps to Unit Testing Happiness

- 1. Describe fully the required functionality of the program.
- 2. Define the header of the program (name, parameter list, return value).
- 3. Elaborate the test cases for the program.
- 4. Build test code that implements all test cases.
- 5. Write the program unit.
- 6. Test, debug, fix, test, debug, fix, test, debug....
- *Then*...*repeat* steps 3-6 for each enhancement and bug report.

Describe required functionality

- I need a variation of SUBSTR that will return the portion of a string between specified start and end locations.
- Some specific requirements:
  - It should work like SUBSTR as much as makes sense (treat a start location of 0 as 1, for example; if the end location is past the end of the string, the treat it as the end of the string).
  - Negative start and end should return a substring at the *end* of the string.
  - Allow the user to specify whether or not the endpoints should be included.

Define the program specification

```
FUNCTION betwnstr (
   string_in       IN   VARCHAR2
 , start_in        IN   PLS_INTEGER
 , end_in          IN   PLS_INTEGER
 , inclusive_in    IN   BOOLEAN DEFAULT TRUE
)
   RETURN VARCHAR2 DETERMINISTIC
```

- My specification or header should be compatible with all requirements.
  - I also self-document that the function is deterministic: no side effects.
- I can (and will) now create a compile-able stub for the program. Why do that?
  - Because I can then fully define and *implement* my test code!

TNT or TDD? Elaborate the test cases

- *Before I write my program,* I will come up with as many of the test cases as possible -- and write my test code.
  - This is known as "test-driven development". TDD is a very hot topic among developers and is associated with Agile Software (http://agilemanifesto.org/) and Extreme Programming.
- Putting aside the fancy names and methodologies, TDD makes perfect sense -- when you stop to think about it.

If you write your program before you define your tests, how do you know you when you're done?

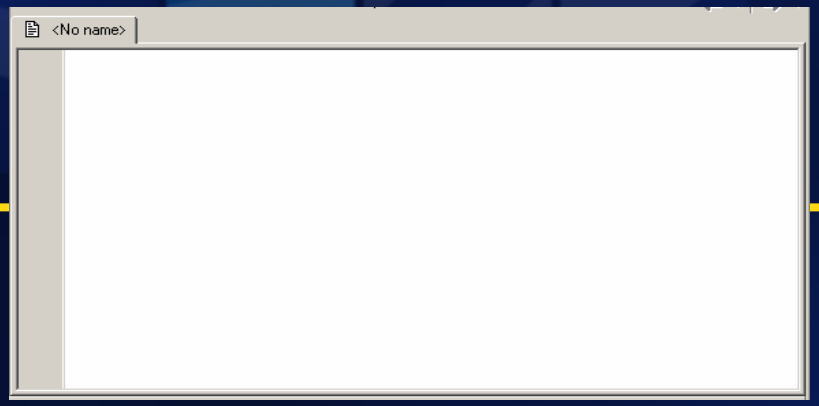
And if you write your tests *afterward*, you are likely to prejudice your tests to show "success."

Brainstorm the test cases

- Even a simple program will have many test cases!
  - You don't have to think of *every* one before you implement your program and start your testing.
  - You should aim at least for a "representative" sampling.
- But where do you store/define the test cases?
  - You can certainly put the information in and work from a document or spreadsheet.
  - Best of all, however, is to link the test case definitions as tightly as possible to the code.

Some of the test cases for BETWNSTR

- Start and end within the string ("normal" usage)
- Start of 0
- End past end of string
- Null string, string of single character, 32767 len character
- Null start and/or end
- Negative start and end
- Start larger than end (positive and negative)
- Variations of the above with different inclusive values



Test cases and Test Code

- The challenge (terror?) of the blank screen....
  - How do I define the test cases?
  - How do I set up those tests?
  - How do I verify the results?
- Let's see how Quest Code Tester helps me tackle these challenges.
  - Define and maintain your test cases through a graphical interface, then let it do all the work.

Write the program.

- Now that I know I can test the program, I can start implementing betwnstr... Finally!

First version of "between string"

```
CREATE OR REPLACE FUNCTION betwnstr (
   string_in       IN   VARCHAR2
 , start_in        IN   PLS_INTEGER
 , end_in          IN   PLS_INTEGER
 , inclusive_in    IN   BOOLEAN DEFAULT TRUE
)
   RETURN VARCHAR2 DETERMINISTIC
IS
BEGIN
   RETURN ( SUBSTR (
       string_in
     , start_in
     , end_in - start_in + 1 )
     );
END;
```

betwnstr1.sf

Test, debug, fix, test, debug, fix, test, debug...

- With a test script in place, I can quickly and easily move back and forth between running my program, identifying errors, debugging and fixing the code, running the program again.
- I also then have my test process and regression test in place so that as I make enhancements or fix bugs, I can fall back on this foundation.
  - It is *critical* that you maintain your test case definitions and test code as your program evolves.
  - And update those *first* -- before you change the program!

Change Your Testing Ways

- Qute (and even utPLSQL) can make a dramatic difference in your ability to test and your confidence in the resulting code.
- Build a comprehensive "library" of unit tests as you build your application
  - These tests and all their test cases can be passed on to other developers
  - Anyone can now enhance or maintain the code with confidence. Make your changes and run the tests. If you get a green light, you're OK!

Testing: Baby steps better than paralysis.

- Unit testing is an intimidating process.
  - You are never really done.
  - You have to maintain your test code along with your application code.
- But every incremental improvement in testing yields immediate and long-term benefits.
  - Don't worry about 100% test coverage.
  - Download Qute and give it a try!

www.ToadWorld.com Downloads link

> Optimize SQL in PL/SQL programs

- Take advantage of PL/SQL-specific enhancements for SQL.
  - BULK COLLECT and FORALL, cursor variables, table functions
- Hide your SQL statements behind a procedural interface so that you can easily change and upgrade.
  - Avoid repetition and dispersion.
- Assume change is going to happen; build that assumption into your code.

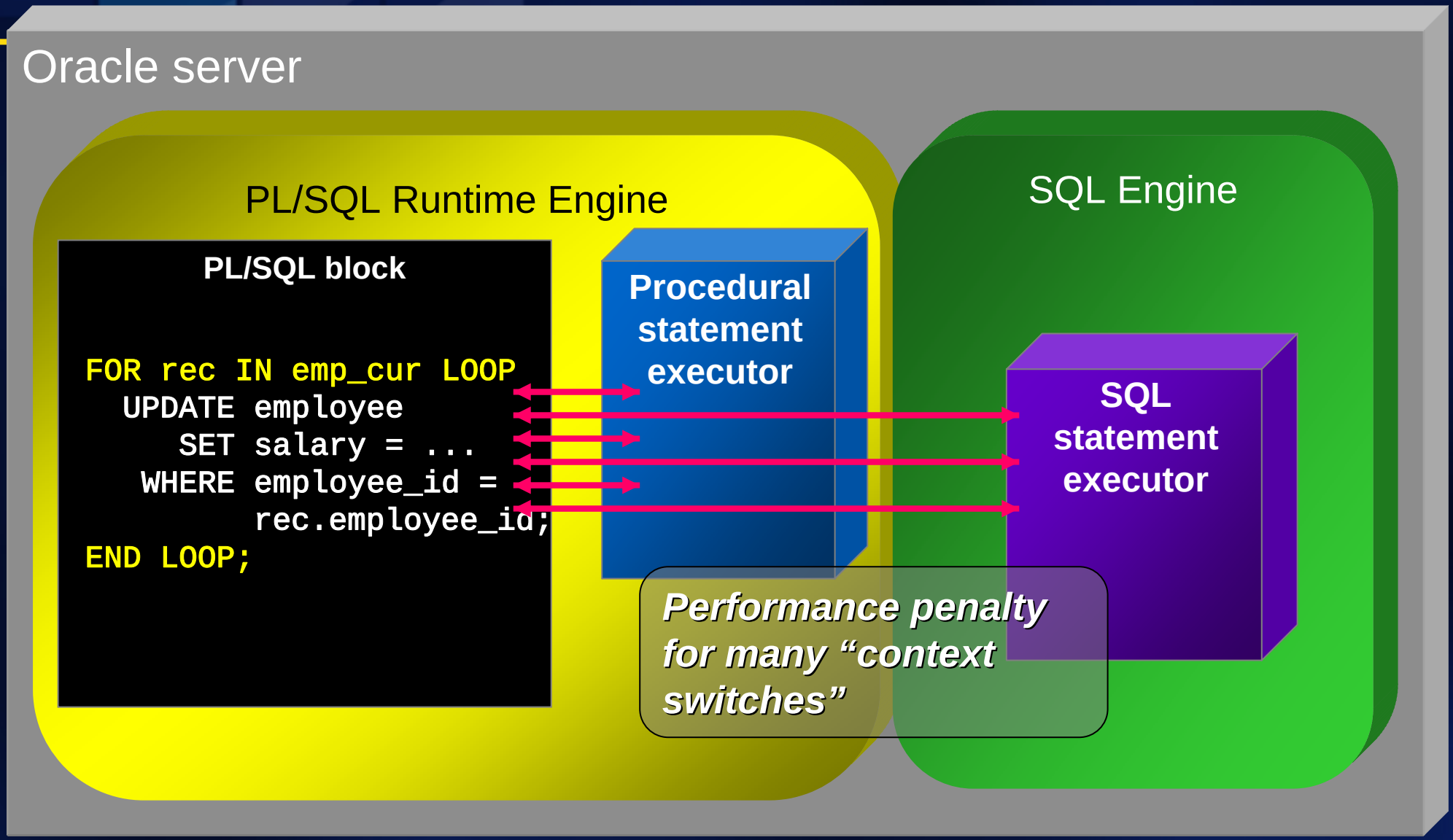
Turbo-charged SQL with BULK COLLECT and FORALL

- Improve the performance of multi-row SQL operations by an order of magnitude or more with bulk/array processing in PL/SQL!

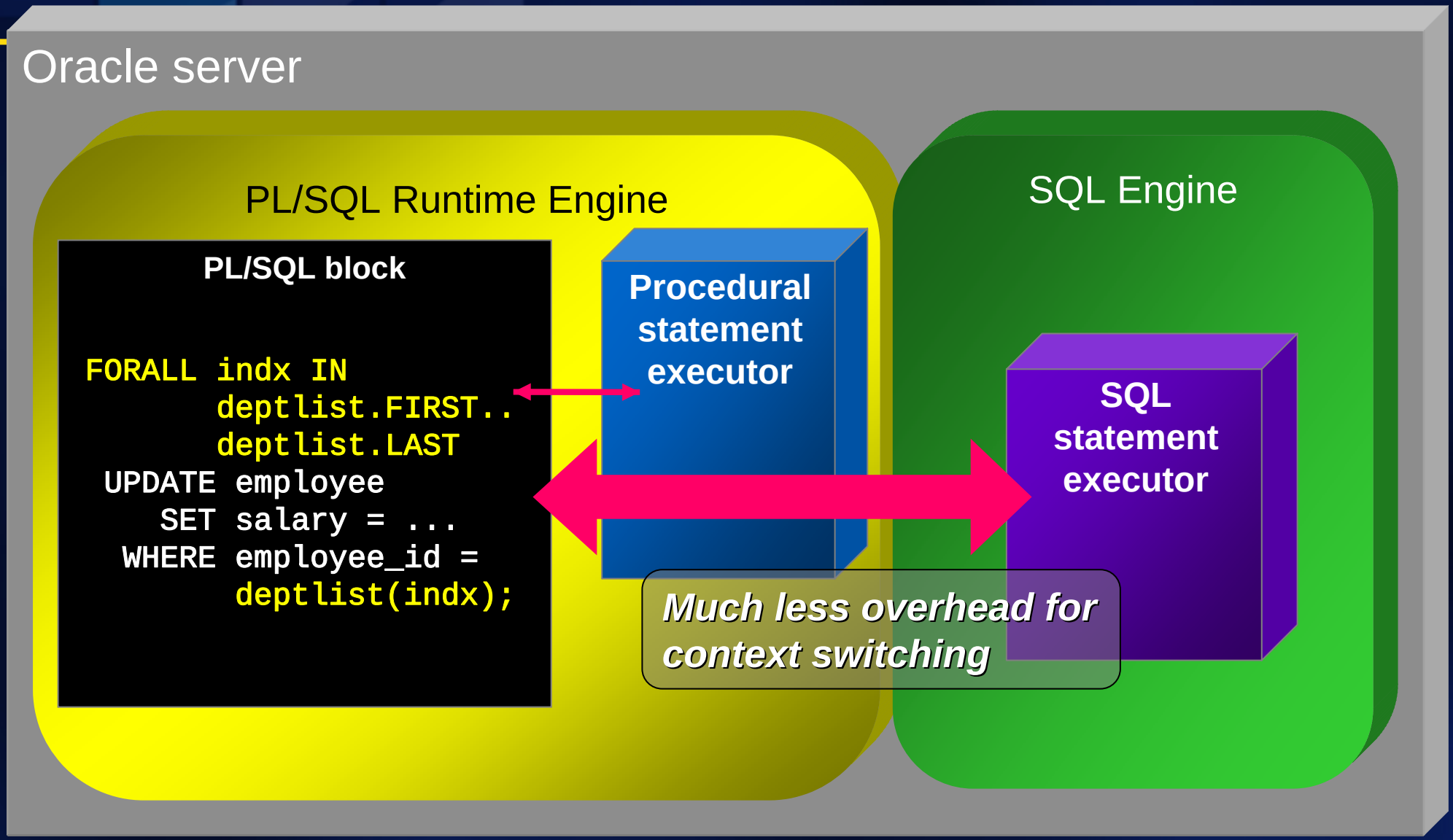
```
CREATE OR REPLACE PROCEDURE upd_for_dept (
   dept_in IN employee.department_id%TYPE
  ,newsal_in IN employee.salary%TYPE)
IS
   CURSOR emp_cur IS
      SELECT employee_id,salary,hire_date
        FROM employee WHERE department_id = dept_in;
BEGIN
   FOR rec IN emp_cur LOOP
      UPDATE employee SET salary = newsal_in
       WHERE employee_id = rec.employee_id;
   END LOOP;
END upd_for_dept;
```

"Conventional binds" (and lots of them!)

Conventional Bind



Enter the “Bulk Bind”



Use the FORALL Bulk Bind Statement

- Instead of executing repetitive, individual DML statements, you can write your code like this:

```
PROCEDURE remove_emps_by_dept (deptlist dlist_t)
IS
BEGIN
   FORALL aDept IN deptlist.FIRST..deptlist.LAST
      DELETE FROM emp WHERE deptno = deptlist(aDept);
END;
```

- Things to be aware of:
  - You MUST know how to use collections to use this feature!
  - Only a single DML statement is allowed per FORALL.
  - SQL%BULK_ROWCOUNT returns the number of rows affected by each row in the binding array.
  - Prior to Oracle10g, the *binding array* must be sequentially filled.
  - Use SAVE EXCEPTIONS to continue past errors.

bulktiming.sql
bulk_rowcount.sql
bulkexc.sql

Use BULK COLLECT INTO for Queries

Declare a collection of records to hold the queried data.

Use BULK COLLECT to retrieve all rows.

Iterate through the collection contents with a loop.

```
DECLARE
   TYPE employees_aat IS TABLE OF employees%ROWTYPE
      INDEX BY BINARY_INTEGER;

   l_employees employees_aat;
BEGIN
   SELECT *
   BULK COLLECT INTO l_employees
     FROM employees;

   FOR indx IN 1 .. l_employees.COUNT
   LOOP
      process_employee (l_employees(indx));
   END LOOP;
END;
```

bulkcoll.sql

Limit the number of rows returned by BULK COLLECT

```
CREATE OR REPLACE PROCEDURE bulk_with_limit
   (deptno_in IN dept.deptno%TYPE)
IS
   CURSOR emps_in_dept_cur IS
      SELECT  *
        FROM emp
       WHERE deptno = deptno_in;

   TYPE emp_tt IS TABLE OF emp%ROWTYPE;
   emps emp_tt;
BEGIN
   OPEN three_cols_cur;
   LOOP
      FETCH emps_in_dept_cur
         BULK COLLECT INTO emps
         LIMIT 100;

      EXIT WHEN emps.COUNT = 0;

      process_emps (emps);
   END LOOP;
END bulk_with_limit;
```

Use the LIMIT clause with the INTO to manage the amount of memory used with the BULK COLLECT operation.

WARNING!

BULK COLLECT will *not* raise NO_DATA_FOUND if no rows are found.

Best to check contents of collection to confirm that something was retrieved.

bulklimit.sql

Tips and Fine Points

- Use bulk binds in these circumstances:
  - Recurring SQL statement in PL/SQL loop. Oracle recommended threshold: five rows!
- Bulk bind rules:
  - Can be used with any kind of collection; Collection subscripts cannot be expressions; The collections must be densely filled (pre-10g); If error occurs, prior successful DML statements are NOT ROLLED BACK.
- Bulk collects:
  - Can be used with implicit and explicit cursors
  - Collection is always filled sequentially, starting at row 1

emplu.pkg
cfl_to_bulk*.*

Don’t take SQL for granted: hide it!

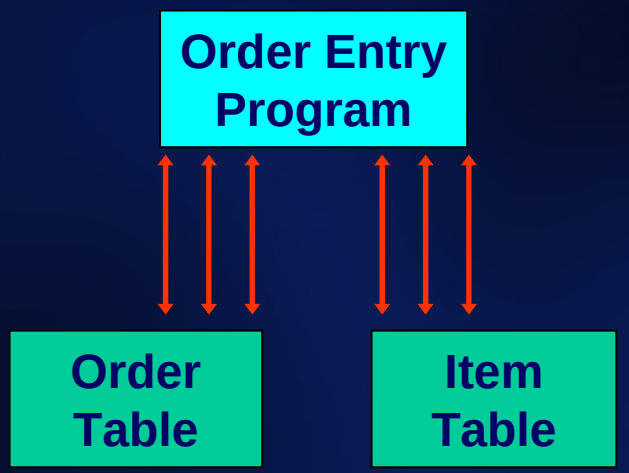
"Why does Steven make such a big deal about writing SQL inside PL/SQL? It's a **no-brainer** in PL/SQL, the *last* thing we have to worry about!"

- I moan and groan about SQL because it is the "Achilles Heel" of PL/SQL.
  - It's so easy to write SQL, it is *too* easy.
- We take SQL for granted, and pay a steep price.

Why We Write PL/SQL Code

- PL/SQL is an embedded language. Its purpose is to provide high-speed, easy access to the Oracle RDBMS.
- The layer of PL/SQL code should *support* the data model, not disrupt our ability to evolve it.

Bottom line: if everyone writes SQL whenever and wherever they want to, it is *very* difficult to maintain and optimize the code.



Don’t Repeat SQL Statements

- Our data structures are about the most volatile part of our application.
  - SQL statements "hard code" those structures and relationships.
  - Shouldn't we then at least avoid repeating the same logical statement?
- Otherwise we have to debug, optimize and maintain the same logic in multiple places.
- How can we avoid such repetition?

How to Avoid SQL Repetition

- You should, as a rule, not even *write* SQL in your PL/SQL programs
  - You can't repeat it if you don't write it
- Instead, rely on pre-built, pre-tested, written-once, used-often PL/SQL programs.
  - "Hide" both individual SQL statements and entire transactions.

Best option: comprehensive table APIs

- Many (not all!) of the SQL statements we need to write against underlying tables and views are very common and predictable.
  - Get me all rows for a foreign key.
  - Get me one row for a primary key.
  - Insert a row; insert a collection of rows.
- Why write these over and over? Instead, rely on a standard, *preferably generated,* programmatic interface that takes care of this "basic plumbing."

SOA for PL/SQL Developers!
SQL is a *service*.
Error mgt is a *service*.

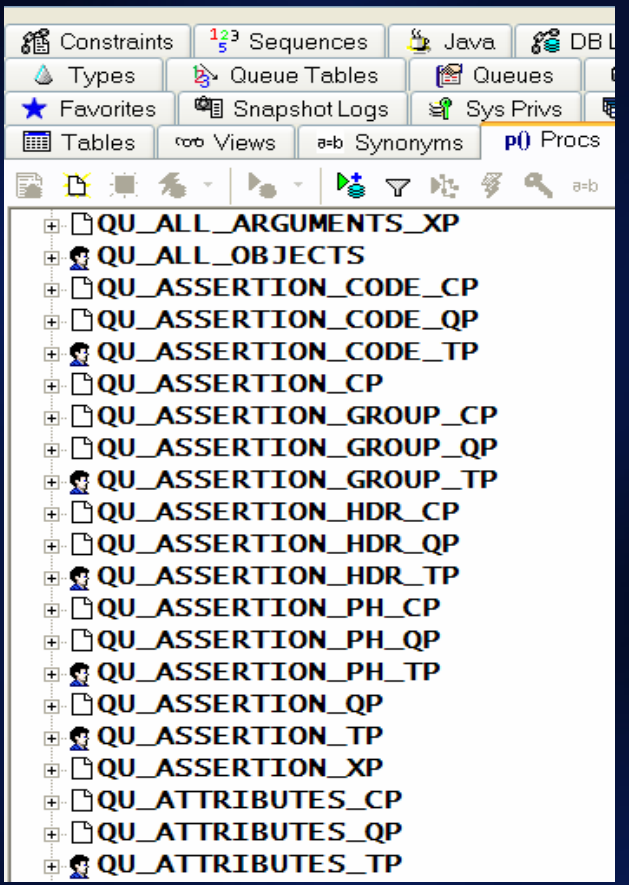
Qnxo
aka the Quest CodeGen Utility
www.qnxo.com

Clear benefits of encapsulated SQL

- Change/improve implementation without affecting application layer of code.
  - Switch between types of queries (implicit vs explicit)
  - Take advantage of data caching, bulk processing, SQL enhancements like MERGE.
- Consistent error handling
  - INSERT: dup_val_on_index?
  - SELECT: too_many_rows?
  - Much less likely to be ignored when the developer writes SQL directly in the application.

Example: Quest Code Tester backend

- For each table, we have three generated packages:
  - <table>_CP for DML
  - <table>_QP for queries
  - <table>_TP for types
- And for many an "extra stuff" package with custom SQL logic and related code:
  - <table>_XP



Hide single row queries

- Let's look at specific examples of encapsulations. First: single row queries.
  - Does a row exist? Get me the row for a unique value.
- Steps to follow:
  - Do *not* write your query directly in application code.
  - Establish clear rules: how are NO_DATA_FOUND and other common errors handled? How are single row queries implemented?
  - Build or generate a function to return the information, usually in the form of a record.

single_row_api.sql

Get me the name for an ID...

```
CREATE OR REPLACE PROCEDURE
process_employee (
   employee_id IN number)
IS
   l_name VARCHAR2(100);
BEGIN
   SELECT last_name || ',' ||
          first_name
     INTO l_name
     FROM employee
    WHERE employee_id =
            employee_id;
    ...
END;
```

Encapsulate SQL and rules...

```
CREATE OR REPLACE PACKAGE employee_rp
AS
   SUBTYPE fullname_t IS VARCHAR2 (200);

   -- The formula
   FUNCTION fullname (
      l employee.last_name%TYPE,
      f employee.first_name%TYPE
   )
      RETURN fullname_t;

   -- Retrieval function
   FUNCTION fullname (
      employee_id_in IN
         employee.employee_id%TYPE
   )
      RETURN fullname_t;
END;
/
```

And now call the function...

```
   l_name employee_rp.fullname_t;
BEGIN
   l_name :=
      employee_rp.fullname (
         employee_id_in);
   ...
END;
```

fullname.pkg
explimpl.pkg

Hide multi-row queries

- A trickier encapsulation challenge: how do you return multiple rows?
  - We will need a "container" or mechanism that is not just a single instance of a row.
- Options in PL/SQL from Oracle9i upwards:
  - Collection - use BULK COLLECT!
  - Cursor variable - especially handy when returning data to a non-PL/SQL host environment

Return multiple rows into a collection

- Collection type must be declared!
  - Can do so in package specification or even as a schema level object.

```
CREATE OR REPLACE PACKAGE BODY multirows
IS
   FUNCTION emps_in_dept (
      dept_in IN employee.department_id%TYPE )
      RETURN employees_aat
   IS
      l_employees employees_aat;
   BEGIN
      SELECT *
      BULK COLLECT INTO l_employees
        FROM employees
       WHERE department_id = dept_in;

      RETURN l_employees;
   END emps_in_dept;
END multirows;
```

multirows.sql

Return multiple rows w/ cursor variable

- A cursor variable is a variable that points to a result set.
  - You can pass CVs from one program unit to another, and even to non-PL/SQL programs!
  - Java, .Net, VB, etc. generally recognize and can work with cursor variables (fetch and even close).
- Uses the OPEN...FOR statement to associate the variable with a query.

return_refcur1.sql
return_refcur.tst
ref_cursor_example.sql

Hide complex data transformations

- Sometimes you need to return multiple rows of data that are the result of a complex transformation.
  - Can't fit it all (easily) into a SELECT statement.
- Table functions to the rescue!
  - A table function is a function that returns a collection and can be called in the FROM clause of a query.
  - Combine with cursor variables to return these datasets through a function interface.

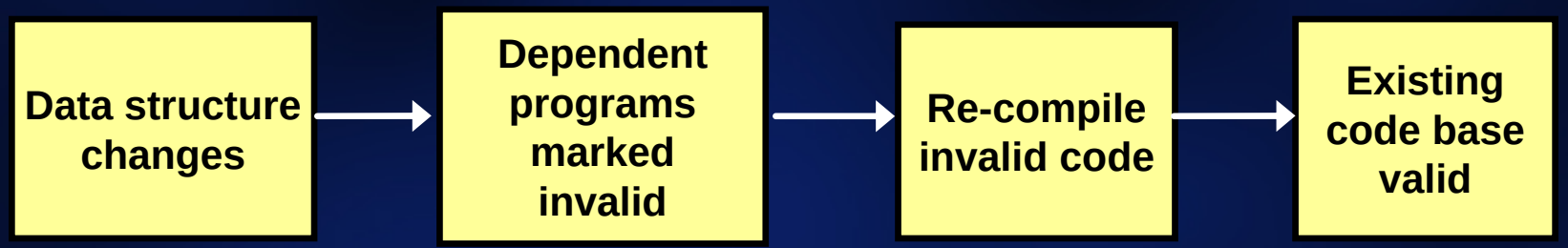
tabfunc_scalar.sql
tabfunc_streaming.sql
tabfunc_pipelined.sql

Hide single and multi-row DML operations

- As crucial as it is to hide queries, it is even more important to encapsulate DML.
  - Error management is more complex and critical.
  - Performance impact is greater.
- A generalized UPDATE is usually the most complicated.
  - Probably will need to hand-code specific update column combinations yourself.

employees_cp.pkb

Write Code Assuming Change



- Use anchoring to tightly link code to underlying data structures
- Fetch into cursor records
- Qualify all references to PL/SQL variables inside SQL statements

DBMS_UTILTY.COMPLE_SCHEMA
UTL_RECOMP(10g)
recompile.sql

Anchor Declarations of Variables

- You have two choices when you declare a variable:
  - Hard-coding the datatype
  - Anchoring the datatype to another structure

Whenever possible, use anchored declarations rather than explicit datatype references

- %TYPE for scalar structures
- %ROWTYPE for composite structures

Hard-Coded Declarations

```
ename VARCHAR2(30);
totsales NUMBER (10,2);
```

Anchored Declarations

```
v_ename emp.ename%TYPE;
totsales pkg.sales_amt%TYPE;

emp_rec emp%ROWTYPE;
tot_rec tot_cur%ROWTYPE;

-- Qnxo approach
emp_rec emp_tp.emp_rt;
l_ename emp_tp.ename_t;
```

Examples of Anchoring

```
DECLARE
   v_ename emp.ename%TYPE;

   v_totsal
      config.dollar_amt%TYPE;

   v_note config.big_string_t;

   v_oneemp
      config.emp_rowtype;
BEGIN
```

```
PACKAGE config
IS
   dollar_amt NUMBER (10, 2);

   SUBTYPE big_string_t IS
      VARCHAR2(32767);

   SUBTYPE emp_allrows_rt IS
      emp%ROWTYPE;
END config;
```

- Use %TYPE and %ROWTYPE when anchoring to database elements
- Use SUBTYPEs for programmatically-defined types
- SUBTYPEs can also be used to mask dependencies that are revealed by %TYPE and %ROWTYPE.

plsql_limits.pks
aq.pkg
Qnxo TP packages

Always Fetch into Cursor Records

wrong

```
   name VARCHAR2 (30);
   minbal NUMBER(10,2);
BEGIN
  OPEN company_pkg.allrows;
  FETCH company_pkg.allrows
     INTO name, minbal;

  IF name = 'ACME' THEN ...

  CLOSE company_pkg.allrows;
```

Fetching into individual variables hard-codes number of items in select list.

right

```
   rec company_pkg.allrows%ROWTYPE;
BEGIN
  OPEN company_pkg.allrows;
  FETCH company_pkg.allrows INTO rec;

  IF rec.name = 'ACME' THEN ...

  CLOSE company_pkg.allrows;
```

Fetching into a record means writing less code.

If the cursor select list changes, it doesn't necessarily affect your code.

Avoid SQL-PL/SQL Naming Conflicts

- One rule: make sure that you never define variables with same name as database elements
  - OK, you can be sure today, but what about tomorrow?
  - Naming conventions simply cannot offer any guarantee
- Better approach: always qualify references to PL/SQL variables inside SQL statements
  - Remember: you can use labels to give names to anonymous blocks

```
PROCEDURE del_scenario
IS
   reg_cd VARCHAR2(100) := :GLOBAL.reg_cd;
BEGIN
   DELETE FROM scenarios
    WHERE reg_cd = del_scenario.reg_cd
      AND scenario_id = :scenario.scenario_id;
END;
```

No problem!

delscen.sql
delscen1.sql
delscen2.sql

> 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

Define your requirements clearly

- When will errors be raised, when handled?
  - Do you let errors go unhandled to the host, trap locally, or trap at the top-most level?
- How should errors be raised and handled?
  - Will users do whatever they want or will there be standard approaches that everyone will follow?
- Useful to conceptualize errors into three categories:
  - Deliberate, unfortunate, unexpected

Different types of exceptions

- Deliberate
  - The code architecture itself deliberately relies on an exception. Example: UTL_FILE.GET_LINE

exec_ddl_from_file.sql
get_nextline.sf

- Unfortunate
  - It is an error, but one that is to be expected and may not even indicate a problem. Example: SELECT INTO -> NO_DATA_FOUND

fullname.pkb

- Unexpected
  - A "hard" error that indicates a problem within the application. Example: Primary key lookup raises TOO_MANY ROWS

fullname.pkb

PL/SQL error management features

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

Quiz! Test your exception handling know-how

- What do you see after running this block?

```
DECLARE
   aname VARCHAR2(5);
BEGIN
   BEGIN
      aname := 'Justice';
      DBMS_OUTPUT.PUT_LINE (aname);
   EXCEPTION
      WHEN VALUE_ERROR
      THEN
         DBMS_OUTPUT.PUT_LINE ('Inner block');
   END;
   DBMS_OUTPUT.PUT_LINE ('What error?');
EXCEPTION
   WHEN VALUE_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
```

excquiz1.sql

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;
```

Quiz: An Exceptional Package

```
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
   p.l ('Before I show you v...');
EXCEPTION
  WHEN OTHERS THEN
    p.l ('Trapped the error!');
END valerr;
```

- So I create the valerr package and then execute the following command. What is displayed on the screen?

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

valerr.pkg
valerr2.pkg

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 functions

- 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, this function returns the full stack of errors with line number information.
  - Formerly, this stack was available only if you let the error go unhandled.

backtrace.sql

DBMS_ERRLOG (Oracle10gR2)

- 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.
- Note: FORALL with SAVE EXCEPTIONS offers similar capabilities.

dbms_errlog.*

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.

aftersererror.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.
- Some general guidelines:
  - Avoid hard-coding of error numbers and messages.
  - Build and use reusable components for raising, handling and logging errors.
- 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.

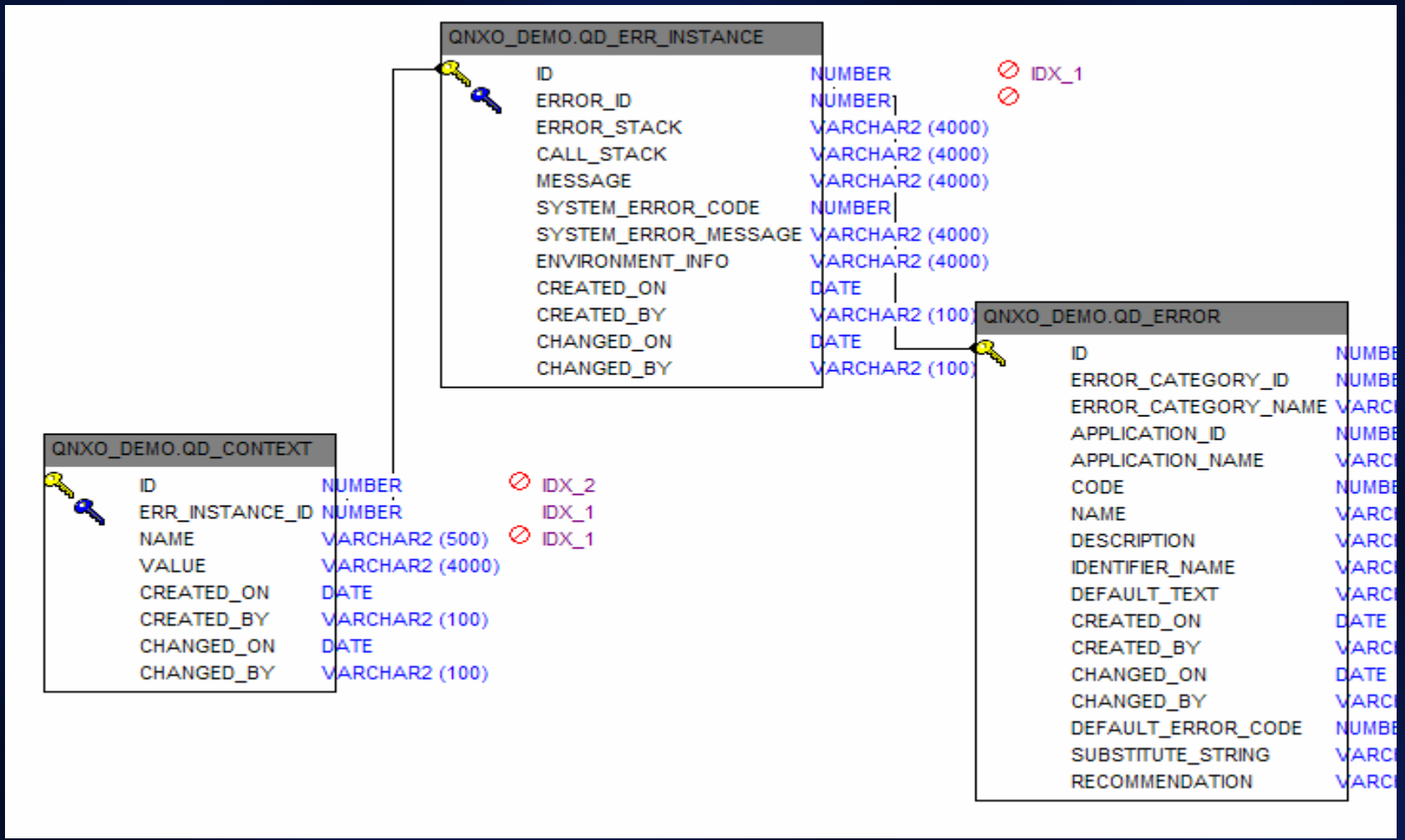
Compensate for PL/SQL weaknesses

- 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....

Object-like representation of an exception

- 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.

ERD for error definition tables



qd_error.erd
qd_runtime.pkb

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.

Prototype exception manager package

Generic Raises

Record and Stop

Record and Continue

```
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;
```

errpkg.pkg

Invoking standard handlers

- The rule: developers should *only* call a pre-defined handler inside an exception section
  - Make it easy for developers 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.

Specifying the error

How should I specify the application-specific error I need to raise?

* Just use -20000 all the time?
* Pick one of those 1000 numbers from -20999 to -20000?
* Use any positive error number besides 1 and 100?
* Use error *names* instead of numbers?

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 trigger logic using the infrastructure 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.

Raise/handle errors by number...or name?

```
BEGIN
   IF employee_rp.is_to_young (:new.hire_date)
   THEN
      RAISE_APPLICATION_ERROR (
         -20175, 'You must be at least 18 years old!');
   END IF;
```

- The above trigger fragment illustrates a common problem: Hard-coding of error numbers and messages.
- Certainly, it is better to use named constants, as in:

```
BEGIN
   IF employee_rp.is_to_young (:new.hire_date)
   THEN
      RAISE_APPLICATION_ERROR (
         employee_rp.en_too_young
       , employee_rp.em_too_young);
   END IF;
```

But now we have a centralized dependency.

Raising errors by name

```
BEGIN
   IF employee_rp.is_to_young (:new.hire_date)
   THEN
      qd_runtime.raise_error (
         'EMPLOYEE-TOO-YOUNG'
       , name1_in => 'LAST_NAME'
       , value1_in => :new.last_name);
   END IF;
```

- Use an error name (literal value).
  - The code compiles *now*.
  - Later, I define that error in the repository.
  - No central point of failure.
- Downsides: risk of typos, runtime notification of "undefined error."

Qnxo
qd_runtime.*

Summary: an Exception Handling Architecture

- 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
- Don't accept the limitations of Oracle's current implementation.
  - You can do lots to improve the situation.

> Write readable, maintainable code

- PL/SQL allows you to write very readable, self-documenting and easily-maintained code.
  - This should be a primary objective for any program.
- Let's look at...
  - Readability features you should use
  - Modular construction in PL/SQL

Readability features you should use

- END labels
  - For program units, loops, nested blocks
- SUBTYPEs
  - Create application-specific datatypes!
- Named notation
  - Sometimes the extra typing is worth it!
- Local or nested modules
  - Key technique, to be covered under "Modular construction..."

end_labels.sql

plsql_limits.pks
explimpl.pkg

namednot.sql

Modular construction in PL/SQL

- Packages: some quick reminders...
  - Logical containers for related elements
  - Overloading
  - Package-level data and caching
  - Initialization section
- Local or nested modules
  - Avoid spaghetti code!
  - Keep your executable sections small/tiny.

Packages: key PL/SQL building block

- Employ object-oriented design principles
  - Build at higher levels of abstraction
  - Enforce information hiding - you can control what people can see and do
  - Call packaged code from object types and triggers
- Encourages top-down design and bottom-up construction
  - TD: Design the interfaces required by the different components of your application without addressing implementation details
  - BU: Existing packages contain building blocks for new code
- Organize your stored code more effectively
- Implements session-persistent data

Overloading

- Overloading, aka, "static polymorphism", occurs when 2 or more programs in the same scope have the same name.
  - Can overload in any declarations section.
- Benefits of overloading include...
  - Improved usability of package: users have to remember fewer names, overloadings anticipate different kinds of usages.
- Beware! Ambiguous overloadings are possible.

ambig_overload.sql

Package Data: Useful, but Sticky

- The scope of a package is your session, and any data defined at the "package level" also has session scope.
  - If defined in the package specification, any program can directly read/write the data.
  - Ideal for program-specific caching.
- General best practice: hide your package data in the body so that you can control access to it.
- Use the SERIALLY_REUSABLE pragma to move data to SGA and have memory released after each usage.

thisuser.pkg
thisuser.tst
emplu.pkg
serial.sql

Package Initialization

- The initialization section:
  - Is defined after and outside of any programs in the package.
  - Is not required. In fact, most packages you build won't have one.
  - Can have exception handling section.
- Useful for:
  - Performing complex setting of default or initial values.
  - Setting up package data which does not change for the duration of a session.
  - Confirming that package is properly instantiated.

```
PACKAGE BODY pkg
IS
   PROCEDURE proc IS
   BEGIN
   END;

   FUNCTION func RETURN
   BEGIN
   END;
BEGIN
   ...initialize...
END pkg;
```

BEGIN after/outside of any program defined in the pkg.

init.pkg
init.tst
datemgr.pkg
valerr.pkg
assoc_array5.sql

Write tiny chunks of code.

Limit executable sections to no more than 50 lines!

?!?!

- It is virtually impossible to understand and therefore debug or maintain code that has long, meandering executable sections.
- How do you follow this guideline?
  - Don't skimp on the packages.
  - Top-down design / step-wise refinement
  - Use lots of local or nested modules.

Let's read some code!

- Move blocks of complex code into the declaration section
- Replace them with descriptive names
- The code is now easier to read and maintain
- You can more easily identify areas for improvement

```
PROCEDURE assign_workload (department_in IN NUMBER)
IS
   CURSOR emps_in_dept_cur
   IS
      SELECT * FROM emp WHERE deptno = department_in;

   PROCEDURE assign_next_open_case
      (emp_id_in IN NUMBER, case_out OUT NUMBER) IS BEGIN … END;

   FUNCTION next_appointment (case_id_in IN NUMBER) IS BEGIN … END;

   PROCEDURE schedule_case
       (case_in IN NUMBER, date_in IN DATE) IS BEGIN … END;

BEGIN /* main */
   FOR emp_rec IN emps_in_dept_cur
   LOOP
      IF analysis.caseload (emp_rec.emp_id) <
         analysis.avg_cases (department_in)
      THEN
         assign_next_open_case (emp_rec.emp_id, case#);
         schedule_case
            (case#, next_appointment (case#));
      END IF;
   END LOOP
END assign_workload;
```

locmod.sp
10g_indices_of.sql

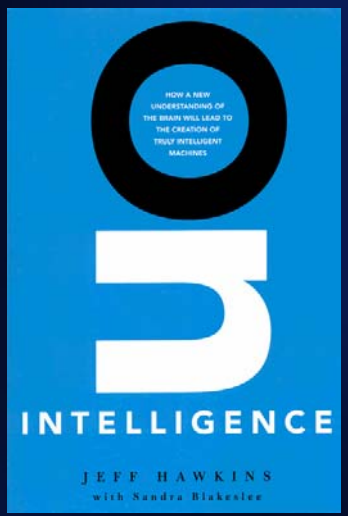
Check out my series on the OverloadCheck utility on OTN

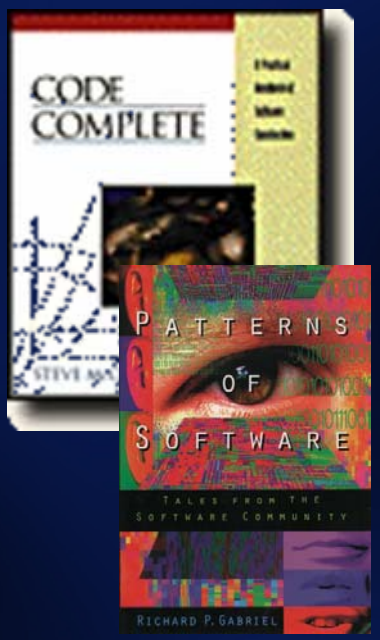
Challenges of local modules

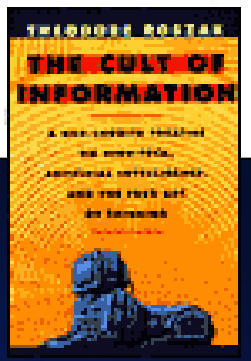
- Requires discipline.
  - Always be on the lookout for opportunities to refactor.
- Need to read from the bottom, up.
  - Takes some getting used to.
- Your IDE needs to reveal the internal structure of the program.
- Sometimes can feel like a "wild goose chase".
  - Where is the darned thing actually *implemented*?

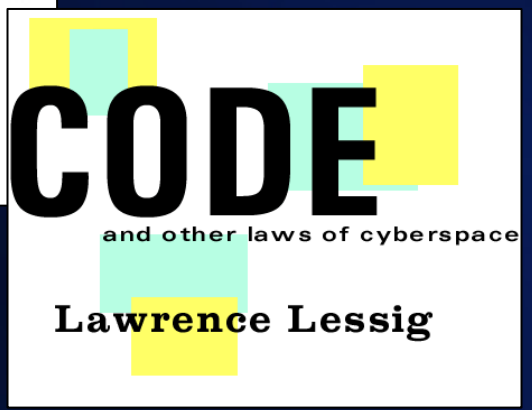
Acknowledgements and Resources

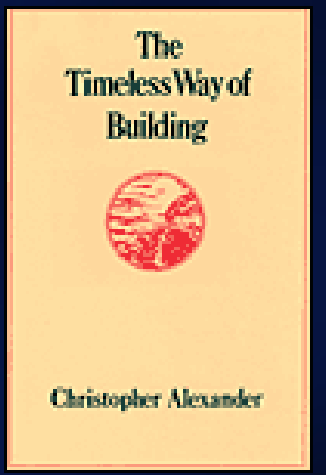
- Very few of my ideas are truly original. I have learned from every one of these books and authors – and you can, too!

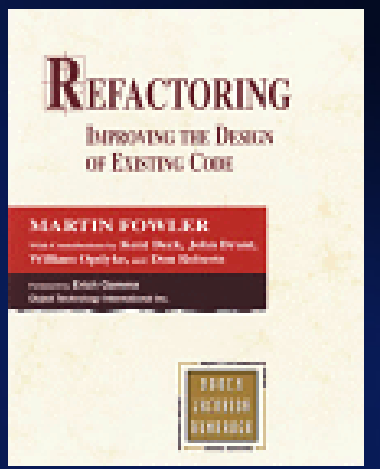












A guide to my mentors/resources

- **A Timeless Way of Building** – a beautiful and deeply spiritual book on architecture that changed the way many developers approach writing software.
- **On Intelligence** – a truly astonishing book that lays out very concisely a new paradigm for understanding how our brains work.
- **Peopleware** – a classic text on the human element behind writing software.
- **Refactoring** – formalized techniques for improving the internals of one's code without affect its behavior.
- **Code Complete** – another classic programming book covering many aspects of code construction.
- **The Cult of Information** – thought-provoking analysis of some of the down-sides of our information age.
- **Patterns of Software** – a book that wrestles with the realities and problems with code reuse and design patterns.
- **Extreme Programming Explained** – excellent introduction to XP.
- **Code and Other Laws of Cyberspace** – a groundbreaking book that recasts the role of software developers as law-writers, and questions the direction that software is today taking us.

(Mostly) Free PL/SQL Resources

- Oracle Technology Network PL/SQL page
  - **http://www.oracle.com/technology/tech/pl_sql/index.html**
- OTN Best Practice PL/SQL
  - **http://www.oracle.com/technology/pub/columns/plsql/index.html**
- Oracle documentation
  - **http://tahiti.oracle.com/**
- OraclePLSQLProgramming.com
  - **http://oracleplsqlprogramming.com/**
- Quest Pipelines
  - **http://quest-pipelines.com/**
- Quest Code Tester for Oracle
  - **http://www.ToadWorld.com** **http://unittest.inside.quest.com/index.jspa**
- PL/Vision
  - **http://quest-pipelines.com/pipelines/dba/PLVision/plvision.htm**

So much to learn, so many ways to improve...

- DRINK LOTS OF WATER.
- WRITE TINY CHUNKS OF CODE.
- STOP WRITING SO MUCH SQL.
- STOP GUESSING, START TESTING.

Never repeat anything!