PL/SQL Practicum #2: Assertions, Exceptions and Module Stability

John Beresniewicz

Technology Manager

Precise Software Solutions

Agenda

- Design by Contract
- Assertions
- Exceptions
- Modular Code

Precise

DESIGN BY CONTRACT

A software engineering discipline for building reliable systems

Design by Contract

Design by Contract is a powerful metaphor that... makes it possible to design software systems of much higher reliability than ever before; the key is understanding that reliability problems (more commonly known as bugs) largely occur at module boundaries, and most often result from inconsistencies in both sides' expectations.

Bertrand Meyer, Object Success

Design by Contract

- Software modules have client-supplier relationships
  - *Client requests and supplier responds*
- These relationships can be expressed as contracts between client and supplier
- Formalizing and enforcing module contracts promotes software reliability

Contract elements

- PRECONDITIONS
  - *What will be true when module is entered?*
  - *Caller's obligation and module's benefit*
- POSTCONDITIONS
  - *What will be true when module completes?*
  - *Module's obligation and caller's benefit*
- INVARIANTS
  - *Anything that should not change as a result of module execution*

Design by Contract and code stability

- TRUST
  - *Preconditions allow modules to trust their input data*
  - *Postconditions allow clients to trust module output*
- CORRECTNESS
  - *Explicit contracts require careful consideration*
- Trusted data + correct algorithms = solid code
- SAFETY
  - *Invariant preservation minimizes risk to other modules*

PL/SQL Call Structure

```
PROCEDURE foo IS...
   IF pkg.fcn(p1var) THEN ...END IF;
```

p1var = 1234 (pre)
TRUE or FALSE (post)

The contract

```
PACKAGE pkg
   FUNCTION fcn (p1_IN IN INTEGER)
      RETURN BOOLEAN
```

PL/SQL and Design by Contract

- Design by Contract = formalizing interfaces
  - *Preconditions are obligations of calling module*
  - *Postconditions are obligations of called module*
  - *Invariants are preserved system states*
- Module IN parm values must obey preconditions
- Module OUT parm and function RETURN values must satisfy postconditions
  - *Implemented by module logic*
- Exception handling state = invariant violation

ASSERTIONS

Enforcing contracts programmatically

PL/SQL assertions

- Test a boolean condition and complain if not TRUE
  - *What does "complain" mean?*
- PL/SQL assertions implemented as a procedure
  - *Always executed, unlike some language environments*

```
PROCEDURE Assert (cond_IN IN BOOLEAN);

Assert(parm1 BETWEEN 0 AND 100);
Assert(plsqltbl.COUNT > 0);
Assert(vbl2 IS NOT NULL);
Assert(fcnX > constantY);
```

Simplest assert procedure

```
PROCEDURE assert (cond_IN BOOLEAN)
IS
BEGIN
   IF NOT NVL(cond_IN,FALSE)
      THEN
         RAISE ASSERTFAIL;
      END IF;
END assert;
```

- Complain = raise assertfail exception
  - *System state change: exception handling*
- NULL tests FALSE and raises the exception

Assert contract preconditions

- Module calls have contract obligations
- Module parameters implement the contract
  - *IN parameter values must obey preconditions*
  - *OUT and RETURN values must obey postconditions*
- Assert preconditions at module entry points
  - *Enforces one side of all contracts*
- *Increased probability all contracts obeyed equates to increased code stability*

Packaging assertions I

```
PACKAGE foo IS
  ASSERTFAIL  EXCEPTION;
  PROCEDURE proc1 (p1 integer);
END foo;
```

```
PACKAGE BODY foo IS
BEGIN
   PROCEDURE proc1 (p1 integer) IS
   BEGIN
      assert(p1 < 100); -- precondition
      /* proc1 code */
   END proc1;
```

- Standard local assertion module in each package reduces coupling

Callers can program defensively

```
BEGIN
   -- other code
   BEGIN
      callme(p1val);
   EXCEPTION WHEN ASSERTFAIL
      THEN apologize_for_p1val;
   END;
   -- more code
```

- Assert does not externalize error, catching scope decides what to do

Performance considerations

- Each call to assert is additional overhead
  - *BUT...assert is package local and code minimal*
- Assertion mechanism cannot be turned on/off
  - *Differences of opinion exist on turning off assertion checks*
- Modules called *very* frequently may need attention
  - *Invariant within large loops*

Turning off assertions

- Simply comment them out but leave in code
  - *They are part of module's specification*
- Only suppress for production performance issue

```
FUNCTION calledoften
  (p1 varchar2, p2 integer) RETURN BOOLEAN
IS
BEGIN
 -- assert(LENGTH(p1) BETWEEN 10 AND 100);
 -- assert(BITAND(p2,3) = 3);
   /* code for module... */
END calledoften;
```

EXCEPTIONS

Dealing with problems systematically

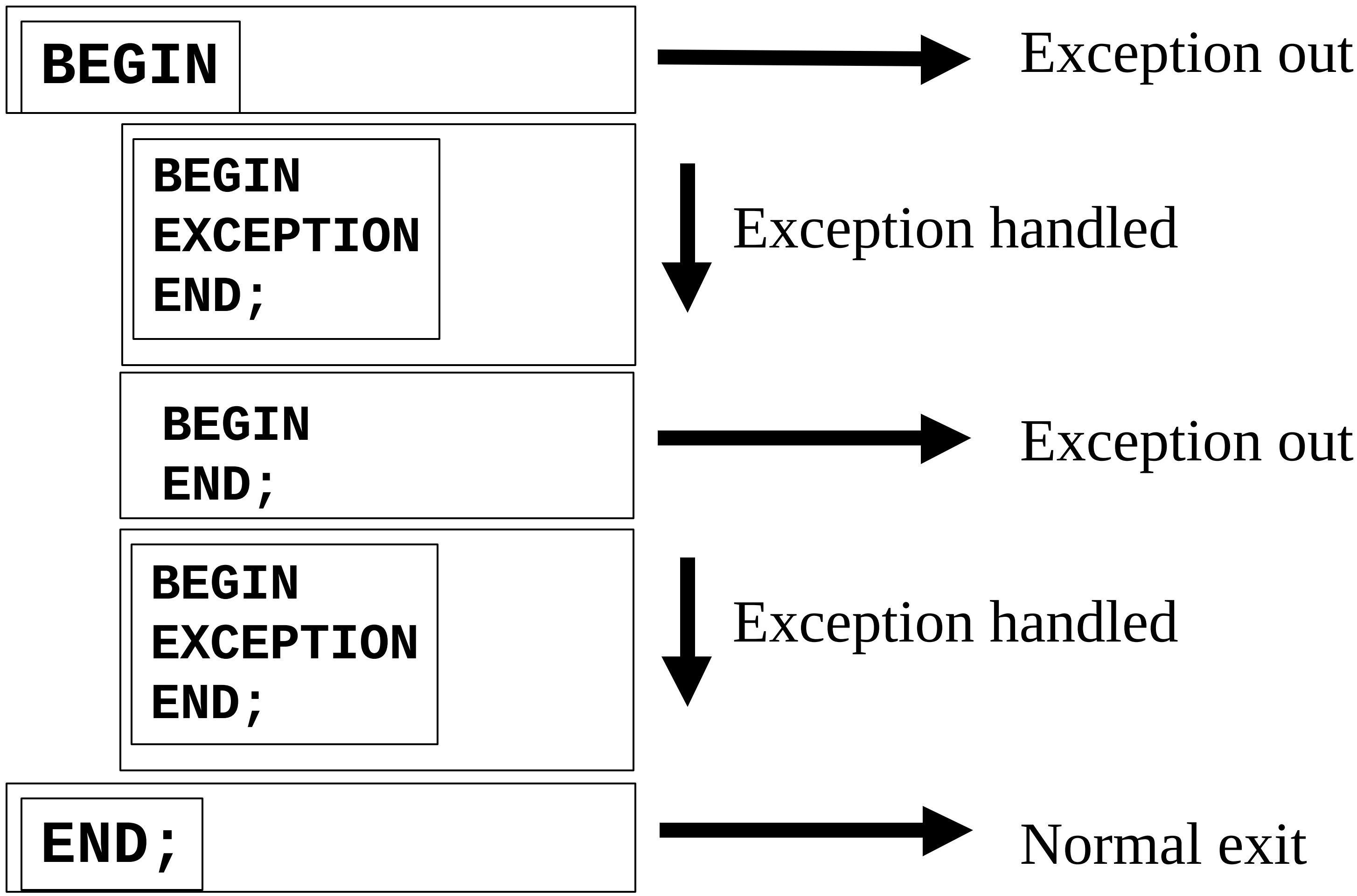
Exception fundamentals

- "Something" undesirable or unexpected happens
  - *We call that something an EXCEPTION*
  - *Either Oracle or application may signal exception*
- Processing jumps from execution block to exception block
  - *If no exception block, exit to caller's exception block...*
- Declaration exceptions exit to caller
  - *Not good, a local problem that cannot be dealt with locally*

Exception classes and treatments

- Anticipated, recoverable and false alarms
  - *Preserve normal program flow using sub-blocks*
- Anticipated, unrecoverable
  - *Contract violations (Assertfail exceptions)*
  - *Fix modules to obey contracts*
- Unanticipated, uncatchable
  - *Declaration exceptions*
- Unanticipated, catchable
  - *Clean up, log error and fail out for analysis*
  - *DO NOT catch and continue (unless mandatory)*

Nesting, program flow, exceptions



Use nesting to continue normal program flow

Catching an exception on purpose

```
FUNCTION IsNumber (txt_IN IN varchar)
 RETURN BOOLEAN
IS
   test  NUMBER;
BEGIN
   BEGIN
      test := TO_NUMBER(txt_IN);
   EXCEPTION
      WHEN VALUE_ERROR THEN null;
   END;
 RETURN (test IS NOT NULL);
END IsNumber;
```

- The exception (or not) provides the essential information

Let's clean that function up some...

```
FUNCTION IsNumber (txt_IN IN varchar)
  RETURN BOOLEAN
IS
   test  NUMBER;
   myBoolReturn  BOOLEAN := FALSE;
BEGIN
   BEGIN
      test := TO_NUMBER(txt_IN);
      myBoolReturn := TRUE;
   EXCEPTION
      WHEN VALUE_ERROR
         THEN myBoolReturn := FALSE;
   END;
   RETURN myBoolReturn;
END IsNumber;
```

Best Practice: smart scoping

- WHEN an Oracle exception can be anticipated in a section of code,

- AND that exception can be safely handled,

- THEN enclose the code in a sub-block and handle the exception (and only that exception)

Declaration exception

```
PROCEDURE notsogood IS
   codevar CHAR(1) := ‘TOO LONG’;
BEGIN
   RAISE VALUE_ERROR;
EXCEPTION
   WHEN OTHERS THEN null;
END notsogood;
```

ORA-06502: PL/SQL: numeric or value error

Easily preventable by code inspection.

Declaration time bomb

```
PROCEDURE notmuchbetter IS
   mycode codes.code%TYPE := ‘SIZE?’;
BEGIN
   RAISE VALUE_ERROR;
EXCEPTION
   WHEN OTHERS THEN null;
END notmuchbetter;
```

ORA-06502: PL/SQL: numeric or value error

Code may break due to change in codes.code datatype.

Declaration mystery error

```
PROCEDURE reallybad IS
   localvar integer := somefcn;
BEGIN
   RAISE VALUE_ERROR;
EXCEPTION
   WHEN OTHERS THEN null;
END reallybad;
```

ORA-06502: PL/SQL: numeric or value error

Where is the exception generated?

Best Practice: declare safely

- Initialize declarations with safe assignments only
  - *Remembering that safe today may not be safe tomorrow*
- DO NOT use functions to initialize declarations
  - *Unless the functions are absolutely trusted*

```
PROCEDURE willnotfail IS
   localvar INTEGER;
BEGIN
   localvar := initfunction;
EXCEPTION
   WHEN OTHERS THEN null;
END willnotfail;
```

Worst practice: catch and ignore

```
FUNCTION badfcn(p1_IN integer)
  RETURN BOOLEAN IS
BEGIN
   /* some code */
EXCEPTION
   WHEN OTHERS THEN RETURN null;
END badfcn;
```

- Masks out ALL errors: callers will think all is fine when something really bad may have happened
- Returns NULL for BOOLEAN, losing opportunity to escape problematic three-valued logic of SQL

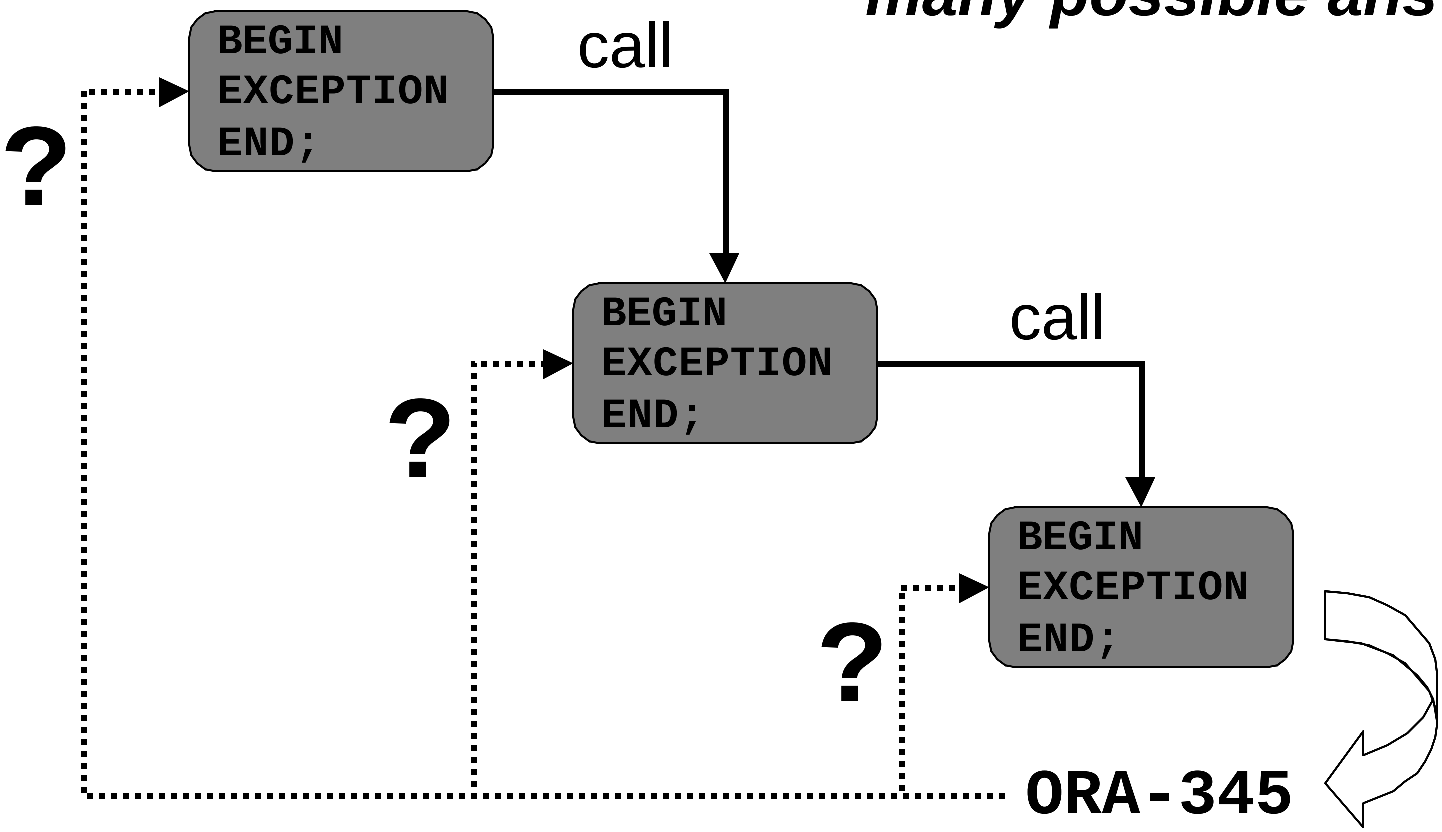
Catch, cleanup and RAISE

```
EXCEPTION
   WHEN OTHERS
   THEN
      log_error(SQLCODE);
      /* local clean up
            (e.g.close cursors) */
      RAISE;
```

- Serious errors should be logged for analysis
- Clean up any resources that persist beyond call
- Re-raise exception to pass on to caller
  - *"Dead programs tell no lies"*

Who should catch exceptions?

A nontrivial question with many possible answers.



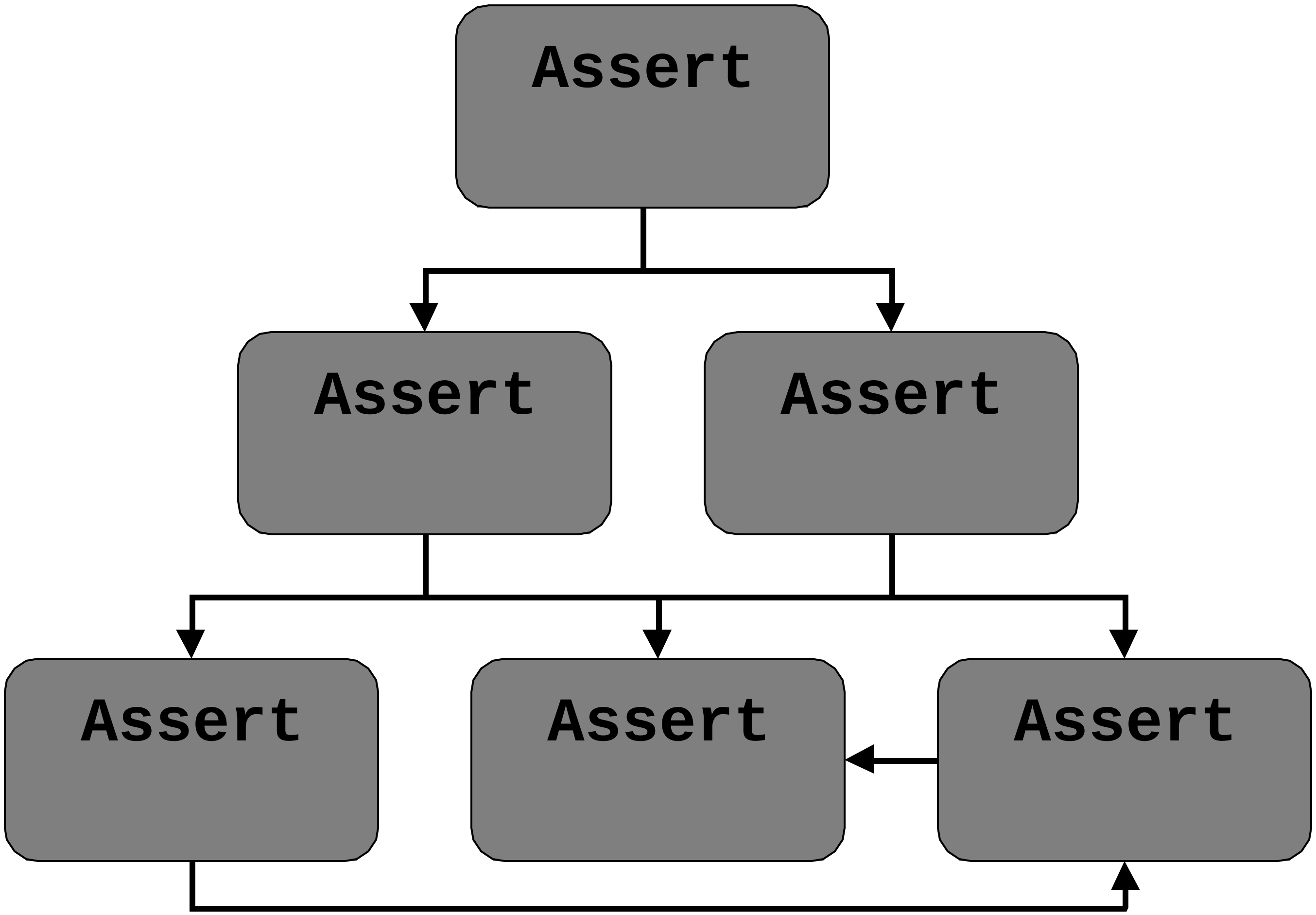
MODULAR CODE

Assembling systems from stable components

Why should we modularize?

- Increased contract enforcement
  - *More interfaces, more asserts, more problems caught*
- Code normalization and reuse
  - *Do things correctly in one place (implement once, call many)*
- Smaller, tighter source code units promote correctness
  - *Better algorithm inspection (especially by others)*

Increased contract enforcement



Where should we modularize?

- At the system level:
  - *Divide functionality into logical components*
  - *Organize components hierarchically*
- Around data:
  - *Encapsulate (table) data access and transactions*
  - *Shared abstract data types*
- Within modules:
  - *Private package modules to encapsulate shared functions*
  - *Private modules within procedures and functions*

Basically everywhere and as much as possible!

Stable, compact module

```
FUNCTION isWeekend(loc_IN IN varchar2
                  ,date_IN IN date)
 RETURN BOOLEAN IS
   tmp_dy  integer;
BEGIN
   assert(loc_IN IN (‘US’,’IL’));
   assert(date_IN IS NOT NULL);
   tmp_dy := TO_CHAR(date_IN,‘D’); -- problem?
   CASE loc_IN
      WHEN ‘US’ THEN RETURN (tmp_dy IN (7,1));
      WHEN ‘IL’ THEN RETURN (tmp_dy IN (6,7));
   END CASE;
END isWeekend;
```

- Boolean function determines if date is weekend
  - *“Weekend” depends on location*

isWeekend contract elements

PRECONDITIONS

- Date_IN not null
- Loc_IN not null
- Loc_IN either ‘US’ or ‘IL’

POSTCONDITIONS

- RETURN TRUE if date_IN is weekend for loc_IN, FALSE otherwise

POTENTIAL PROBLEM?

- Do we REALLY know how TO_CHAR works (given NLS options)?
- We could introduce a new precondition:

```
-- September 2,2001 is Sunday
assert(1 = TO_CHAR(TO_DATE('09:02:2001','MM:DD:YYYY')
                  ,'D') );
```

Potential problem confirmed

The date format element D returns the number of the day of the week (1-7). The day of the week that is numbered 1 is specified implicitly by the initialization parameter NLS_TERRITORY.

Oracle8i SQL Reference

Summary Points

- Use standardized assertions
  - *Enforce preconditions in all modules*
- Code clearly and carefully
  - *Postconditions depend on proper algorithms*
- Use code inspection
  - *Clear, documented logic promotes accuracy*
- Modularize
  - *More modules = more contracts*
  - *Small execution sections promote better inspections*
- Eliminate exceptions
  - *Assert, anticipate, avoid invariant violations*

Resources

- *Object-oriented Software Construction, 2nd Edition* by Bertrand Meyer (Prentice-Hall, 2001)
- *Object Success* by Bertrand Meyer (out of print)
- *The Pragmatic Programmer* by Andrew Hunt, et al (Addison-Wesley, 1999)
- *PL/SQL Best Practices* by Steven Feuerstein (O'Reilly & Associates, 2001)