Draft proposal for PL/SQL unit testing in PSI project

DRAFT

Summary

We need unit testing to develop reliable, maintainable software. We propose using the utPLSQL framework for PL/SQL unit testing and present an example addressing how we might use it in the PSI project. We could get started quickly with a cautious phased approach. The initial installation and early troubleshooting will require support from the DBA team unless SYSDBA privileges can be granted to the development team in the development environments.

Overview

Automated unit testing is key to developing reliable, maintainable software. Unit tests should be easy to run without side-effects and provide rapid feedback to the developer. Unit tests are not created post-development; they are part of the development process. They must be first-class citizens of the code base. Good unit tests do not take more time to develop; they save time by providing a constant safety net. They give the developer courage to make changes without the fear of breaking things. They are a first check that our change does what we expect, **and all the other code covered by tests still does what it should**.

Unit tests should be fast while providing quick feedback to the developer. They should not have side effects. Running the tests several times should not fill up files or tables.

> For each desired change, make the change easy (warning: this may be hard), then make the easy change. - *Kent Beck*

If the code is testable, if it is covered by tests that can be run quickly while providing easy to understand feedback, then we can make changes with confidence. We can refactor and make improvements that seemed too risky before.

Definition

With the term "unit tests" we mean automated tests focused on the smallest piece of the software that can be tested (the unit). Unit tests are part of the software development process. They are written and can be run by the developer. Unit tests can cover multiple modules but should never rely on external systems.

Unit tests do not rely on external systems or operator inputs. Integration testing and systems testing are important but distinctly different from unit testing.

Framework

Unit testing in most scenarios should be independent of a database. PL/SQL presents a unique challenge as the database is the engine running the code. utPLSQL is a framework for unit testing PL/SQL code. It is installed in the database and runs in the database. It is designed to give an experience similar to other modern unit testing frameworks.

How do we start?

Note: We can do the examples in real-time as a live demonstration.

Let's consider a fictional example requiring a change to the existing `kmprailscalenglo.setloadnoinrailcar` procedure. For the exercise, we need to make sure it only acts on "Pickup" transactions, not "Delivery" transactions.

Cover the existing behavior

Before changing an existing function or procedure, we want to create a test that covers its correct behavior. We are building our own safety net. With a test coverage, we can have some confidence that our changes haven't broken that behavior.

We must review the code to understand how it works. The comments help us out here.

```
-- PROCEDURE:   setLoadNoInRailcar
--
-- DESCRIPTION: Update LoadNo,FLAGISLOADED into the NUCOR_INOUTBOUNDTRANSACTION Table
```

We start with defining a test to cover the normal behavior. We want to make sure that the load number gets updated correctly and the flag indicating a loaded rail car is set.

```
create or replace package test_setloadnoinrailcar as

  -- %suite(kmprailscalenglo.setloadnoinrailcar tests)

  -- %test(Should set load number and flag as loaded)
  procedure normal_load;

end;
/
```

The comments here are annotations used by testing framework. They identify the `test_setloadnoinrailcar` package as a test suite and the `normal_load` procedure as a test.

With this specification in the database, let's try running our first test.

```
Running tests
kmprailscalenglo.setloadnoinrailcar tests
  Should set load number and flag as loaded [.087 sec] (FAILED - 1)

Failures:

  1) normal_load
      ORA-04067: not executed, package body "TEST.TEST_SETLOADNOINRAILCAR" does not exist
      ORA-06508: PL/SQL: could not find program unit being called: "TEST.TEST_SETLOADNOINRAI
      ORA-06512: at line 6
Finished in .108976 seconds
1 tests, 0 failed, 1 errored, 0 disabled, 0 warning(s)
```

We can see our annotations as part of the test summary output. The test has failed because we do not have a body, yet.

Let's get started with a simple stub.

```
create or replace package body test_setloadnoinrailcar as

  procedure normal_load
  is
  begin
    null;
  end;

end;
/
```

The does nothing right now, but it passes (Green). We have a no-op stub as a starting point. This might not be necessary, but as someone with little experience in PL/SQL it helps me confirm the syntax and structure.

Implementing the test can be hard. It may be difficult to arrange conditions for testing code that was not designed to be easy to test. It may require refactoring without the safety net of existing tests. This is very important to do and we will gain immediate benefit by capturing existing behavior. The next time we need to change this code, the test will already be there. We are setting up our future self for success.

After some trial and error we manage a test to verify the load is given the correct number and flagged as loaded.

```
create or replace package body test_setloadnoinrailcar as

  subtype ticket_no_type is main.nucor_inoutboundtransaction.ticketno%type;
  subtype transport_mode_type is main.nucor_inoutboundtransaction%rowtype;
  subtype transport_mode_id_type is main.tm.tmbez%type;
  subtype load_no_type is main.pg.pg_id%type;
```

```
  procedure create_onsite_load
    ( ticket_no in ticket_no_type
    , railcar_id in transport_mode_id_type
    , load_no in load_no_type )
  is
  begin

    insert into main.pg (pg_id) values (load_no);

    insert into main.nucor_inoutboundtransaction (ticketno, mottype, flagonsite, motid)
    values (ticket_no, 'test_mot', 'Y', railcar_id);

  end create_onsite_load;

  procedure normal_load is
    railcar_id  transport_mode_id_type;
    load_no     load_no_type;
    railcar     transport_mode_type;
    ticket_no   ticket_no_type;
  begin

    -- arrange
    railcar_id := 'rail_car_id';
    load_no := 'load_number';
    ticket_no := -1;
    create_onsite_load(ticket_no, railcar_id, load_no);

    -- act
    main.kmprailscalenglo.setloadnoinrailcar(railcar_id, load_no);

    -- assert
    select * into railcar
    from main.nucor_inoutboundtransaction
    where ticketno = ticket_no;

    ut.expect(railcar.loadno).to_(equal(load_no));
    ut.expect(railcar.flagisloaded).to_(equal('Y'));

  end normal_load;

end;
```

The test is rather verbose. This will likely be common when covering code which was not designed to be tested. At least we did not have to modify the original procedure.

Let's compile and run the tests.

```
Running tests
kmprailscalenglo.setloadnoinrailcar tests
Running tests finished.
  Should set load number and flag as loaded [1.755 sec]
  2022-11-22 16:16:53.597|8|kmpRailScaleNGLO.setLoadNoInRailcar     ||->
  2022-11-22 16:16:54.874|8|kmpRailScaleNGLO.setLoadNoInRailcar     ||<-

Finished in 1.787185 seconds
1 tests, 0 failed, 0 errored, 0 disabled, 0 warning(s)
```

We have our first passing test covering the intended behavior in one scenario. We might consider this the bare minimum to continue. It's a good start. Consider taking the time to cover corner-cases and possible error conditions. For instance, if this procedure is called for a rail car that is not on site, it will throw a `no data found` exception. We could write a test to document that behavior.

Notice the test output includes log messages. Ideally, we do not want tests to have side-effects.

New behavior

With a test in place, we can proceed with more confidence. We'll use the principles of test driven development to write a test covering some part of the requirement to be implemented. This test will most likely fail or not compile (Red). We'll write the least amount of code we can to make the test pass (Green). Then we'll refactor, knowing that our tests will let us know right away if the behavior has changed. We'll write another test to cover more of the desired behavior and repeat this cycle, Red -> Green -> Refactor, until the full required behavior is implemented and proven by tests.

An example In the specification for the `test_setloadnoinrailcar` test suite, let's add a new test.

```
  -- %test does not update "Delivery" transactions.
  procedure does_not_update_delivery;
```

Running the tests now causes the test suite to fail (Red). We need a test body. Let's write the minimum code we can to make it pass.

```
  procedure does_not_update_delivery
  is
  begin
    null;
  end;
```

Once again, this does nothing. We have a no-op stub to help confirm the syntax and structure. It's probably not necessary but shows how the tests let us take

small steps with constant feedback.

Now let's enhance the test. We want to use a `Delivery` transaction in the `mottransactionkind` field. Setting up the test could be similar to `normal_load`. In fact, let's just copy the whole thing. We'll just make a small changes to populate the field with a new `create_onsite_delivery` helper.

```
procedure create_onsite_delivery
  ( ticket_no in ticket_no_type
  , railcar_id in transport_mode_id_type
  , load_no in load_no_type )
is
begin

  insert into main.pg (pg_id) values (load_no);

  insert into main.nucor_inoutboundtransaction
    ( ticketno
    , mottype
    , flagonsite
    , motid
    , mottransactionkind)
  values (ticket_no, 'test_mot', 'Y', railcar_id, 'Delivery');

end create_onsite_delivery;
```

How should the system respond when we try to update a delivery transaction? Should it silently do nothing? Should it throw a standard exception? Or maybe a custom exception?

A custom exception is probably ideal here, but for demonstration purposes we'll go with `no data found` exception.

In the specification

```
-- %test does not update "Delivery" transactions.
-- %throws(no_data_found)
procedure does_not_update_delivery;
```

And in the body

```
procedure does_not_update_delivery
is
  railcar_id  transport_mode_id_type;
  load_no     load_no_type;
  railcar     transport_mode_type;
  ticket_no   ticket_no_type;
begin
  -- arrange
```

```
    railcar_id := 'rail_car_id';
    load_no := 'load_number';
    ticket_no := -1;
    create_onsite_delivery(ticket_no, railcar_id, load_no);

    -- act
    main.kmprailscalenglo.setloadnoinrailcar(railcar_id, load_no);

    -- assert
    -- throws no data found

  end;
```

Let's compile and run the tests.

```
Failures:

  1) does_not_update_delivery
      Expected one of exceptions (-1403) but nothing was raised.

Finished in .312338 seconds
2 tests, 1 failed, 0 errored, 0 disabled, 0 warning(s)
```

We have a test which fails (Red). Let's try to make the test pass by changing the code. In the `setLoadNoInRailcar` procedure we'll add an additional condition when selecting railcar records.

Original

```
  --Select active record for the railcar
  SELECT * INTO vNINOUT FROM NUCOR_INOUTBOUNDTRANSACTION
      WHERE MOTID = pi_RailcarID
        AND FLAGONSITE = glo.FLAG_YES
        AND DTCHECKOUT IS NULL;
```

And we add criteria to filter out 'Delivery' transactions.

```
  --Select active non-Delivery record for the railcar
  SELECT * INTO vNINOUT FROM NUCOR_INOUTBOUNDTRANSACTION
      WHERE MOTID = pi_RailcarID
        AND MOTTRANSACTIONKIND <> 'Delivery'
        AND FLAGONSITE = glo.FLAG_YES
        AND DTCHECKOUT IS NULL;
```

We compile and test. In my limited understanding, I expected this pass. I'm wrong. We get a `no data found` error for the first test. I broke the existing behavior! But I also got immediate feedback on it and know I have to fix it.

The circumstances make it clear that `mottransactionkind` defaults to 'Delivery'. Our normal load test does not consider this field. We now understand that

normal behavior does not apply the 'Delivery' transactions. Let's specify that the normal case is for a 'Pickup' transaction type.

```
procedure create_onsite_pickup
  ( ticket_no in ticket_no_type
  , railcar_id in transport_mode_id_type
  , load_no in load_no_type )
is
begin

  insert into main.pg (pg_id) values (load_no);

  insert into main.nucor_inoutboundtransaction
    ( ticketno
    , mottype
    , flagonsite
    , motid
    , mottransactionkind)
  values (ticket_no, 'test_mot', 'Y', railcar_id, 'Pickup');

end create_onsite_pickup;

procedure normal_load is
  railcar_id  transport_mode_id_type;
  load_no     load_no_type;
  railcar     transport_mode_type;
  ticket_no   ticket_no_type;
begin

  -- arrange
  railcar_id := 'rail_car_id';
  load_no := 'load_number';
  ticket_no := -1;
  create_onsite_pickup(ticket_no, railcar_id, load_no);

  -- act
  main.kmprailscalenglo.setloadnoinrailcar(railcar_id, load_no);

  -- assert
  select * into railcar
  from main.nucor_inoutboundtransaction
  where ticketno = ticket_no;

  ut.expect(railcar.loadno).to_(equal(load_no));
  ut.expect(railcar.flagisloaded).to_(equal('Y'));
```

```
  end normal_load;
```

We compile and test.

```
Finished in .302474 seconds
2 tests, 0 failed, 0 errored, 0 disabled, 0 warning(s)

Running tests finished.
```

All tests pass (Green). But we are not done. The next step is to refactor. There is a good deal of duplicated code. Can we minimize that? Can we improve the variable and procedure names to make things easier to read? We can take small steps toward improvement and run the test to get immediate feedback.

After several rounds of refactoring we might have something like:

```
create or replace package test_setloadnoinrailcar as

  -- %suite(kmprailscalenglo.setloadnoinrailcar tests)

  -- %test(Sets load number and flag as loaded for Pickups.)
  procedure sets_loadno_flagisloaded_for_pickup;

  -- %test(Throws no data found for Deliveries.)
  -- %throws(no_data_found)
  procedure throws_nodatafound_for_delivery;

end;
/

create or replace package body test_setloadnoinrailcar as

  subtype transaction_type is main.nucor_inoutboundtransaction.mottransactionkind%type;

  delivery    constant transaction_type                                  := 'Delivery';
  pickup      constant transaction_type                                  := 'Pickup';
  yes         constant main.glo.flag_yes%type                            := main.glo.flag_yes;
  railcar_id  constant main.tm.tmbez%type                                := 'rail_car_id';
  load_no     constant main.pg.pg_id%type                                := 'load_number';
  mot         constant main.nucor_inoutboundtransaction.mottype%type     := 'test_mot';
  ticket_no   constant main.nucor_inoutboundtransaction.ticketno%type    := -1;
  railcar     main.nucor_inoutboundtransaction%rowtype;

  procedure setup_load (trnsaction in transaction_type) is
  begin

    insert into main.pg (pg_id) values (load_no);
```

```
    insert into main.nucor_inoutboundtransaction
      ( ticketno
      , mottype
      , flagonsite
      , motid
      , mottransactionkind)
    values (ticket_no, mot, yes, railcar_id, trnsaction);

  end setup_load;

  procedure sets_loadno_flagisloaded_for_pickup is
  begin

    -- arrange
    setup_load(pickup);

    -- act
    main.kmprailscalenglo.setloadnoinrailcar(railcar_id, load_no);

    -- assert
    select * into railcar
    from main.nucor_inoutboundtransaction
    where ticketno = ticket_no;

    ut.expect(railcar.loadno).to_(equal(load_no));
    ut.expect(railcar.flagisloaded).to_(equal(yes));

  end;

  procedure throws_nodatafound_for_delivery is
  begin
    setup_load(delivery);
    main.kmprailscalenglo.setloadnoinrailcar(railcar_id, load_no);
    -- throws no data found
  end;
end;
/
```

We've pulled common setup up into the package scope, renamed procedures, and made comments a bit more precise based on our better understanding. There may yet be more areas where this could be improved. For additional features, we can write a test (Red), make it pass with least code reasonably achievable (Green), and refactor. Repeating the Red/Green/Refactor cycle as our understanding and requirements change.

If all development follows a similar process going forward, test coverage will grow.

Phased approach

Consider a phased approach to implementation.

Initial phase

Start with an allowance that some procedures may not be reasonably covered by tests.

- All *new* functions and procedures should be covered by tests.
- A test driven development style is strongly encouraged.
- Writing tests for existing functions and procedures before making a change is strongly encouraged.
  - If a test cannot feasibly be implemented, the reason should be clearly explained and documented (how?)

Learning phase

Use the information we've learned in the initial phase to inform our decisions on how to improve. How often are we seeing untestable code? Can we refactor it to be testable? Shift from encouraging to requiring tests.

Maturing phase

As code coverage for new development grows, identify the gaps. What functionality is not covered? Consider allocating resource to widen test coverage for all Nucor defined behavior in PL/SQL code.

Installation and troubleshooting

Installation of the utPLSQL framework require SYSDBA privileges. SYSDBA will also be required to grant privileges to the test schema. Some privileges have been identified, but it is possible others will be not be identified until the early stages of actual use. Consider granting SYSDBA privileges to development team members to minimize demand on DBA team support, if permitted.

Open questions

- Is any existing code untestable? Can it feasibly be refactored to make it, or portions of it, testable?
- How would we document that a test is not feasible?
- Can we grant SYSDBA privileges to non-DBA team members in uncontrolled environments?
- Is it possible to disable trace logging for test scenarios?

Next steps - spend more time - slides - POC validate pull request CI/CD pipeline