# Advanced PL/SQL Instrumentation

## Implementing A Logging Interface Using Oracle Object Types

**Anthony Harper**

Senior Technical Architect

information architects
designing enduring quality

## Overview

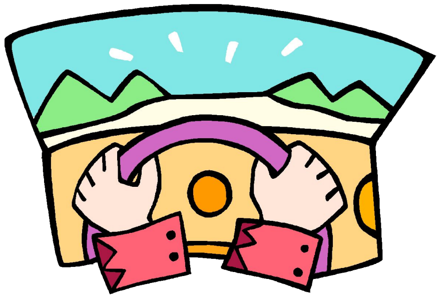
- Tracing and Control: Costs and Benefits.
- Object Types to the rescue.
- Details of the implementation.
- Stepping through the code (if time allows).
- Q&A.

## Context

- We are talking about tracing for PL/SQL packaged programs, not making trace calls from a client application.
- The PL/SQL Execution environment is in the database, so no issues from traveling to and from the database.
- I’ll be looking at how to maximize benefits of the “best practice” of tracing in PL/SQL.

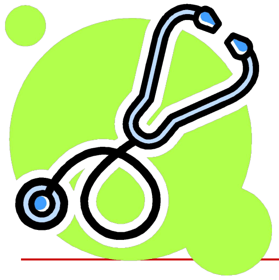
# What Is Instrumentation?

- A way to make your code musical.
- A new dashboard for your code.
- A way to give your code a checkup.
- The only way to fly.



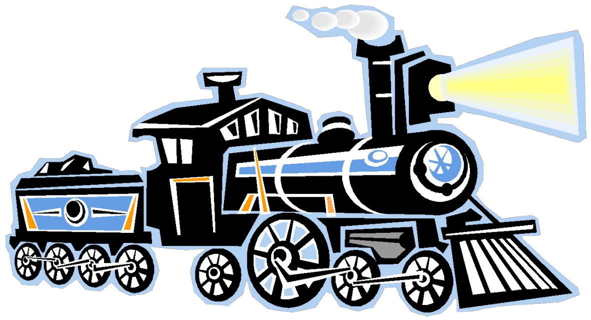
## All of the above, and more.

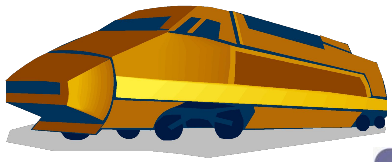
- With instrumentation code can sing.
- Build a dashboard to see how your code is moving on the road.
- Give your programs a checkup, without going to the doctor's office.
- Ask the pilot questions while your code is airborne.



## The Hidden Cost of Tracing

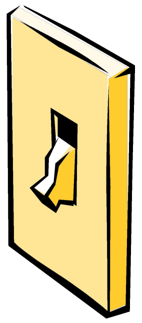
- "Instrumentation sounds just like what I need, what's the problem?"
- Wait, before we get on the tracing train…something isn't quite right…
- "Doesn't tracing add excessive overhead to PL/SQL program execution?"

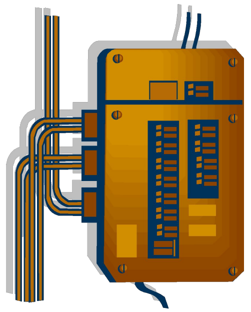


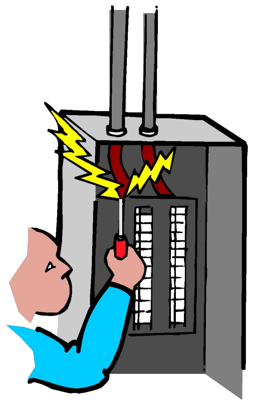


# Control Concepts

- Wouldn't it be nice to be able to 'control' when the code in your program wrote to the logs by switching the settings in a table?
- Wouldn't it be even better to control logging for every package?







## Control Issues

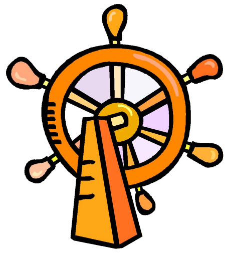
- We can make tracing manageable by looking up control settings in a table.
- These lookups add overhead, we don’t want to check the environment table with every call to the tracing utility.
- This just moves the problem area, now we are checking the environment too much, just to reduce tracing overhead.

## Types of Instrumentation

- Exception Logging for errors.
- Trace Logging for troubleshooting.
- Audit Logging for history.
- Timing statistics for ETL processing.
- All these types of instrumentation have different control needs.

## Object Types To The Rescue

- Objects have properties, so we can control tracing at the package level.
- Objects have methods, making it a breeze to use controlled logging without duplicating code.
- Object Types let you implement a logging interface for your package.



## Object State

- We can create the properties once in the object type to hold tracing state.
- These properties are local to the declared instances of the object type.
- With an initialize method called by package initialization, control overhead is reduced to the first call to each package in each session.

## Object Methods

- Using an object type to wrap logging calls connects tracing control and tracing activity.
- Control state is enforced locally, before calling the central logging utility.
- Wrapping calls in objects cleans up our code, making it a snap to use.

## Setup For Instrumentation

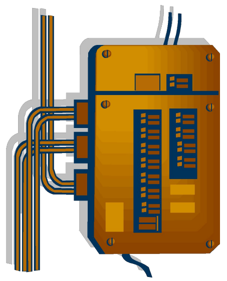
- Instrumentation Mechanism
- Control Mechanism
- Creating An Object Type
- Integration With Program Code

## Instrumentation Mechanism

- Best practices discussions recommend using a centralized mechanism for instrumentation.
- Create a table for storing log entries.
- Create a utility package to write to the log.
- Add calls to the log utility to your programs to ‘instrument’ them.

## Managing Environment Settings

- Control Instrumentation at runtime by using environment values that are stored in a table.
- Use a central package to read/write control values.
- Read control state from environment table for each package on initialization.
- Now you have a central switchbox for your program instrumentation.



## Creating A Logging Object

- Create a type specification with properties for maintaining control state and methods for writing logs.
- Create a type body to implement the methods, checking the state variables to see if each call should be executed.
- Add an initialize method to call from the package initialization.

# The Key To Controlling Tracing

Forward trace calls to the log package only if the state is enabled.  Its that simple.

```
member procedure trace
  (
     p_message in varchar2,
     p_method  in varchar2 default '#USE_CURRENT'
  ) is
  begin
     if self.enable_traces = 1 then
        set_object_method(p_method);
        app#log_util.write_trace(
      self.object_name, self.method_name, p_message);
     end if;
  end trace;
```

## Instrument Your Programs

It is an easy process to implement advanced instrumentation at this point.

1. Declaring The Logging Object.
2. Initializing Logging State.
3. Using The Logging Type.

## 1) Declaring The Logging Object

- Declare a logging instance in the package specification of each package.
- Use the package name in the call to the constructor.
- You just implemented a logging interface!

```
create or replace package my_package
as
--declare the logging object
logger app#log_type := app#log_type('my_package');
…
```

## 2) Initializing Logging State

- From the package initialization section, call the logger.init method.
- This looks up the current control state in the environment and sets the type properties appropriately.

```
create or replace package body my_package
as
…(package implementations)

begin
   logger.init;
end my_package;
```

## 3) Using The Logging Type

- Using the logging type is simple:
- Just call the trace method to write traces.

```
procedure my_traced_call(
p_input in varchar2,
p_user in varchar2)
is
begin
logger.traces('Input was: ' || p_input, 'my_traced_call');
logger.traces('User was: ' || p_user);
…
```

## Production Scenarios

- For normal production operation, don’t enable tracing for all packages.
- If a problem happens, use the set environment method on the package to turn on the tracing calls.
- Have the user reproduce the problem.  All of the tracing calls will be written to the log table.

## Adding Value With Instrumentation

- Having controlled instrumentation is a real advantage.
- You can’t leverage this advantage if you don’t instrument your programs.
- Since the tracing calls are normally off, the cost is low for putting detailed tracing in each program unit.

## Practical Benefits

- Trace only when you need it.
- Eliminate calls from the dbas about the log tablespace getting filled up.
- Real time, **on-demand** ETL process monitoring.

## Practical Development

- Instrumentation can be a great aid even during the development cycle…showing that the code is working as designed.
- Now you can keep these benefits without commenting out traces before rolling out the code.

## Practical Wizardry

- Controlled error logging with tracing allows proactive support of programs.
- If an error is being logged for a particular package, you can activate tracing and look at the problem before the user calls the helpdesk.
- This can definitely help with your reputation as a wizard when you call the user to discuss a problem they just ran into:)

## Stepping through the code

- (demo showing code in action)*

## Log Records: With Tracing Off

```
SQL> select * from app#logs_daily;

0 rows selected
```

# Log Records: With Tracing On

```
SQL> select * from app#logs_daily;
LOG_TYPE   OBJECT_NAME     OBJECT_METHOD             CREATED_BY   CREATED_DATE        MESSAGE
---------- --------------- ------------------------- ------------ -------------------
---------------------------------------------
TRACE      my_package      my_traced_procedure       OPP          25 08 2007 18:49:24 Entered Procedure
TRACE      my_package      my_other_procedure        web user     25 08 2007 18:49:24 Finished executable section
TRACE      my_package      my_traced_procedure       OPP          25 08 2007 18:49:24 User was: OPP
TRACE      my_package      my_traced_procedure       OPP          25 08 2007 18:49:24 Finished executable section
TRACE      my_package      my_other_procedure        OPP          25 08 2007 18:49:24 Entered Procedure
TRACE      my_package      my_other_procedure        OPP          25 08 2007 18:49:24 Input was: test input value xxx
TRACE      my_package      my_other_procedure        OPP          25 08 2007 18:49:24 User was: OPP
TRACE      my_package      my_other_procedure        OPP          25 08 2007 18:49:24 Calling my_traced_procedure to show
callstack
TRACE      my_package      my_traced_procedure       OPP          25 08 2007 18:49:24 Entered Procedure
TRACE      my_package      my_traced_procedure       OPP          25 08 2007 18:49:24 Input was: test input value xxx
TRACE      my_package      my_traced_procedure       OPP          25 08 2007 18:49:24 User was: OPP
TRACE      my_package      my_traced_procedure       OPP          25 08 2007 18:49:24 Finished executable section
TRACE      my_package      my_other_procedure        OPP          25 08 2007 18:49:24 Done with sub procedure call
TRACE      my_package      my_other_procedure        OPP          25 08 2007 18:49:24 Finished executable section
TRACE      my_package      my_traced_procedure       web user     25 08 2007 18:49:24 Entered Procedure
TRACE      my_package      my_traced_procedure       web user     25 08 2007 18:49:24 Input was: test input value xxx
TRACE      my_package      my_traced_procedure       web user     25 08 2007 18:49:24 User was: web user
TRACE      my_package      my_traced_procedure       web user     25 08 2007 18:49:24 Finished executable section
TRACE      my_package      my_other_procedure        web user     25 08 2007 18:49:24 Entered Procedure
TRACE      my_package      my_other_procedure        web user     25 08 2007 18:49:24 Input was: test input value xxx
TRACE      my_package      my_other_procedure        web user     25 08 2007 18:49:24 User was: web user
TRACE      my_package      my_other_procedure        web user     25 08 2007 18:49:24 Calling my_traced_procedure to show
callstack
TRACE      my_package      my_traced_procedure       web user     25 08 2007 18:49:24 Entered Procedure
TRACE      my_package      my_traced_procedure       web user     25 08 2007 18:49:24 Input was: test input value xxx
TRACE      my_package      my_traced_procedure       web user     25 08 2007 18:49:24 User was: web user
TRACE      my_package      my_traced_procedure       web user     25 08 2007 18:49:24 Finished executable section
TRACE      my_package      my_other_procedure        web user     25 08 2007 18:49:24 Done with sub procedure call
TRACE      my_package      my_traced_procedure       OPP          25 08 2007 18:49:24 Input was: test input value xxx
28 rows selected
```

## Other possibilities

- Create a timing type that inherits from the logging type to expose new methods for millisecond timing statistics. Implementing this type is approaching multiple inheritance.
- Update control mechanism for 11g to leverage PL/SQL caching. This solves the problem with applications that don't close sessions on a regular basis.
- Implement other standard interfaces for database email notification and PL/SQL web service consumers.

## Review and Questions

- Finally, a good use for Oracle objects.
- A simple way of adding fine grained control to your best practices implementation of logging.
- Add instrumentation to your code without slowing down healthy production code.
- Maximize the benefits of tracing and minimize the costs.
- Questions?????????????

- Thanks for attending.

## Resources

- The notes refer to the appropriate script.
- Class scripts can be downloaded at:

http://plsqlarchitects.com/logging

- Email:

anthony.harper@corporateinformationarchitects.com