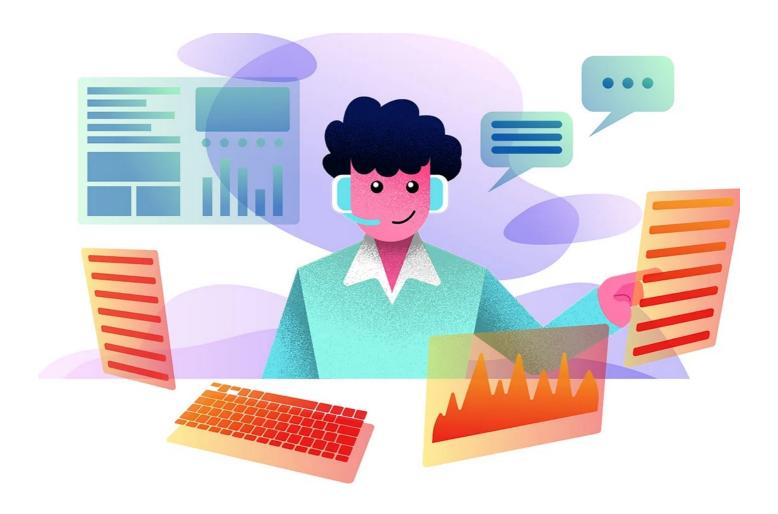
# PL/SQL & SQL Coding Guidelines



# **Tips for Development & Operation**

Document Version 4.0-SNAPSHOT © 2020 Trivadis AG



# **Table of Contents**

Table of Contents	2
About	6
Foreword	6
License	7
Trademarks	7
Disclaimer	7
Revision History	7
Introduction	8
Scope	8
Document Conventions	8
SQALE characteristics and subcharacteristics	8
Severity of the rule Keywords used	10
Why are standards important	
We have other standards	11
We do not agree with all your standards	12
Naming Conventions	13
General Guidelines	13
Naming Conventions for PL/SQL	13
Database Object Naming Conventions	14
Collection Type	14
Column Check Constraint	
DML / Instead of Trigger	15
Foreign Key Constraint	15
Function Index	16 16
Object Type	16
Package	16
Primary Key Constraint Procedure	
Sequence	17
Synonym	17
System Trigger Table	
Temporary Table (Global Temporary Table)	18
Unique Key Constraint View	
Coding Style	20
Formatting	20
RulesExample	20
Code Commenting	22
Conventions	22
Commenting Tags Example	22 22
example	ZZ
Language Usage	23
General	23
G-1010: Try to label your sub blocks.	23
G-1020: Always have a matching loop or block label. G-1030: Avoid defining variables that are not used.	24 26
G-1040: Avoid dead code.	27
G-1050: Avoid using literals in your code.	

G-1060: Avoid storing ROWIDs or UROWIDs in database tables. G-1070: Avoid nesting comment blocks.	31
Variables & Types	33
General	33
G-2110: Try to use anchored declarations for variables, constants and types. G-2120: Try to have a single location to define your types.	33 34
G-2130: Try to use subtypes for constructs used often in your code.	35
G-2135: Avoid assigning values to local variables that are not used by a subsequent statement. G-2140: Never initialize variables with NULL.	36 37
G-2150: Avoid comparisons with NULL value, consider using IS [NOT] NULL.	38
G-2160: Avoid initializing variables using functions in the declaration section.	39
G-2170: Never overload variables. G-2180: Never use quoted identifiers.	40
G-2185: Avoid using overly short names for explicitly or implicitly declared identifiers.	41
G-2190: Avoid using ROWID or UROWID.	43
Numeric Data Types	44
G-2210: Avoid declaring NUMBER variables, constants or subtypes with no precision.	44
G-2220: Try to use PLS_INTEGER instead of NUMBER for arithmetic operations with integer values.	45
G-2230: Try to use SIMPLE_INTEGER datatype when appropriate.	46
Character Data Types	47
G-2310: Avoid using CHAR data type. G-2320: Never use VARCHAR data type.	47
G-2330: Never use zero-length strings to substitute NULL.	49
G-2340: Always define your VARCHAR2 variables using CHAR SEMANTIC (if not defined anchored).	50
Boolean Data Types	51
G-2410: Try to use boolean data type for values with dual meaning.	51
Large Objects	52
G-2510: Avoid using the LONG and LONG RAW data types.	52
DML & SQL	53
General	53
G-3110: Always specify the target columns when coding an insert statement.	53
G-3120: Always use table aliases when your SQL statement involves more than one source.	54
G-3130: Try to use ANSI SQL-92 join syntax.	56
G-3140: Try to use anchored records as targets for your cursors.	57
G-3150: Try to use identity columns for surrogate keys.	58
G-3160: Avoid visible virtual columns.	59
G-3170: Always use DEFAULT ON NULL declarations to assign default values to table columns if you refuse to store NULL values.  G-3180: Always specify column names instead of positional references in ORDER BY clauses.	61
G-3190: Avoid using NATURAL JOIN.	63
G-3195: Always use wildcards in a LIKE clause.	64
Bulk Operations	65
G-3210: Always use BULK OPERATIONS (BULK COLLECT, FORALL) whenever you have to execute a DML statement for more than 4 times.	65
Transaction Control	66
G-3310: Never commit within a cursor loop.	66
G-3320: Try to move transactions within a non-cursor loop into procedures.	68
Control Structures	69
CURSOR	69
G-4110: Always use %NOTFOUND instead of NOT %FOUND to check whether a cursor returned data.	69
G-4120: Avoid using %NOTFOUND directly after the FETCH when working with BULK OPERATIONS and LIMIT clause.	70
G-4130: Always close locally opened cursors.	72
G-4140: Avoid executing any statements between a SQL operation and the usage of an implicit cursor attribute.  CASE / IF / DECODE / NVL / NVL2 / COALESCE	73 75
G-4210: Try to use CASE rather than an IF statement with multiple ELSIF paths.	75
G-4220: Try to use CASE rather than DECODE.	76
G-4230: Always use a COALESCE instead of a NVL command, if parameter 2 of the NVL function is a function call or a SELECT statement.	77
G-4240: Always use a CASE instead of a NVL2 command if parameter 2 or 3 of NVL2 is either a function call or a SELECT statement.	78
G-4250: Avoid using identical conditions in different branches of the same IF or CASE statement.	79
G-4260: Avoid inverting boolean conditions with NOT.	80
G-4270: Avoid comparing boolean values to boolean literals.	81
Flow Control	82
G-4310: Never use GOTO statements in your code.	82 84
G-4320: Always label your loops. G-4330: Always use a CURSOR FOR loop to process the complete cursor results unless you are using bulk operations.	86
G-4330. Always use a CORSOR FOR loop to process the complete cursor results unless you are using bulk operations.  G-4340: Always use a NUMERIC FOR loop to process a dense array.	87
G-4350: Always use 1 as lower and COUNT() as upper bound when looping through a dense array.	88
G-4360: Always use a WHILE loop to process a loose array.	89
G-4370: Avoid using EXIT to stop loop processing unless you are in a basic loop.	90
G-4375: Always use EXIT WHEN instead of an IF statement to exit from a loop.	92
G-4380 Try to label your EXIT WHEN statements.	93
G-4385: Never use a cursor for loop to check whether a cursor returns data.	95

G-4390: Avoid use of unreferenced FOR loop indexes.	96
G-4395: Avoid hard-coded upper or lower bound values with FOR loops.	97
Exception Handling	98
G-5010: Try to use a error/logging framework for your application. G-5020: Never handle unnamed exceptions using the error number.	98
G-5030: Never handle unifiathed exceptions using the error humber.  G-5030: Never assign predefined exception names to user defined exceptions.	100
G-5040: Avoid use of WHEN OTHERS clause in an exception section without any other specific handlers.	102
Example (exception to the rule)	102
G-5050: Avoid use of the RAISE_APPLICATION_ERROR built-in procedure with a hard-coded 20nnn error number or hard-coded message.	103
G-5060: Avoid unhandled exceptions.	104
G-5070: Avoid using Oracle predefined exceptions.	105
Dynamic SQL	106
G-6010: Always use a character variable to execute dynamic SQL.	106
G-6020: Try to use output bind arguments in the RETURNING INTO clause of dynamic DML statements rather than the USING clause.	107
Stored Objects	108
General	108
G-7110: Try to use named notation when calling program units.	108
G-7120 Always add the name of the program unit to its end keyword.	109
G-7130: Always use parameters or pull in definitions rather than referencing external variables in a local program unit.	110
G-7140: Always ensure that locally defined procedures or functions are referenced.	112
G-7150: Try to remove unused parameters.	113
G-7160: Always explicitly state parameter mode.	114
G-7170: Avoid using an IN OUT parameter as IN or OUT only.	115
Packages  G-7210: Try to keep your packages small. Include only few procedures and functions that are used in the same context.	117 117
G-7210: Try to keep your packages small. Include only few procedures and functions that are used in the same context.  G-7220: Always use forward declaration for private functions and procedures.	118
G-7230: Avoid declaring global variables public.	120
G-7240: Never use RETURN in package initialization block.	122
Procedures	123
G-7310: Avoid standalone procedures – put your procedures in packages.	123
G-7320: Avoid using RETURN statements in a PROCEDURE.	124
G-7330: Always assign values to OUT parameters.	125
Functions 0.7410. Available of matters and the street in a final fraction in a street in a final final fraction.	126
G-7410: Avoid standalone functions – put your functions in packages. G-7420: Always make the RETURN statement the last statement of your function.	126 127
G-7420: Always make the RETORN statement the last statement of your function.  G-7430: Try to use no more than one RETURN statement within a function.	127
G-7440: Never use OUT parameters to return values from a function.	129
G-7450: Never return a NULL value from a BOOLEAN function.	130
G-7460: Try to define your packaged/standalone function deterministic if appropriate.	131
Oracle Supplied Packages	132
G-7510: Always prefix ORACLE supplied packages with owner schema name.	132
Object Types	133
Triggers	134
G-7710: Avoid cascading triggers.	134
G-7720: Never use multiple UPDATE OF in trigger event clause. G-7730: Avoid multiple DML events per trigger if primary key is assigned in trigger.	136 137
Sequences	137
G-7810: Never use SQL inside PL/SQL to read sequence numbers (or SYSDATE).	139
Patterns	140
Checking the Number of Rows	140
G-8110: Never use SELECT COUNT(*) if you are only interested in the existence of a row.  G-8120: Never check existence of a row to decide whether to create it or not.	140
Access objects of foreign application schemas	141
G-8210: Always use synonyms when accessing objects of another application schema.	142
Validating input parameter size	143
G-8310: Always validate input parameter size by assigning the parameter to a size limited variable in the declaration section of program unit.	143
Ensure single execution at a time of a program unit	145
G-8410: Always use application locks to ensure a program unit is only running once at a given time.	145
Use dbms_application_info package to follow progress of a process	147
G-8510: Always use dbms_application_info to track program process transiently.	147
omplexity Analysis	148
Halstead Metrics	148
Calculation	148
McCabe's Cyclomatic Complexity	148
Description	148
Calculation	149

Code Reviews	151
Tool Support	152
Development	152
Setting the preferences	152
Activate PLSQL Cop using context menu	152
Software metrics	153
Appendix	155
A - PL/SQL & SQL Coding Guidelines as PDF	155
B - Mapping new guidelines to prior versions	155

# **About**

# Foreword



In the I.T. world of today, robust and secure applications are becoming more and more important. Many business processes no longer work without I.T. and the dependence of businesses on their I.T. has grown tremendously, meaning we need robust and maintainable applications. An important requirement is to have standards and guidelines, which make it possible to maintain source code created by a number of people quickly and easily. This forms the basis of well functioning off- or on-shoring strategy, as it allows quality assurance to be carried out efficiently at the source.

Good standards and guidelines are based on the wealth of experience and knowledge gained from past (and future?) problems, such as those, which can arise in a cloud environment, for example.



Urban Lankes Chairman of the Board of Directors Trivadis



The Oracle Database Developer community is made stronger by resources freely shared by experts around the world, such as the Trivadis Coding Guidelines. If you have not yet adopted standards for writing SQL and PL/SQL in your applications, this is a great place to start.

Steven Feuerstein

Steven Feuerstein Team Lead, Oracle Developer Advocates Oracle



Coding Guidelines are a crucial part of software development. It is a matter of fact, that code is more often read than written – therefore we should take efforts to ease the work of the reader, which is not necessarily the author.

I am convinced that this standard may be a good starting point for your own guidelines.

Roger Troller Senior Consultant finnova AG Bankware

# License

The Trivadis PL/SQL & SQL Coding Guidelines are licensed under the Apache License, Version 2.0. You may obtain a copy of the License at <a href="http://www.apache.org/licenses/LICENSE-2.0">http://www.apache.org/licenses/LICENSE-2.0</a>.

# **Trademarks**

All terms that are known trademarks or service marks have been capitalized. All trademarks are the property of their respective owners.

# Disclaimer

The authors and publisher shall have neither liability nor responsibility to any person or entity with respect to the loss or damages arising from the information contained in this work. This work may include inaccuracies or typographical errors and solely represent the opinions of the authors. Changes are periodically made to this document without notice. The authors reserve the right to revise this document at any time without notice.

# **Revision History**

The first version of these guidelines was compiled by Roger Troller on March 17, 2009. Jörn Kulessa, Daniela Reiner, Richard Bushnell, Andreas Flubacher and Thomas Mauch helped Roger complete version 1.2 until August 21, 2009. This was the first GA version. The handy printed version in A5 format was distributed free of charge at the DOAG Annual Conference and on other occasions. Since then Roger updated the guidelines regularily. Philipp Salvisberg was involved in the review process for version 3.0 which was a major update. Philipp took the lead, after Roger left Trivadis in 2016. In 2020 Kim Berg Hansen started handling guidelines maintenance, letting Philipp concentrate on the related Trivadis PL/SQL Cop tool.

Since July, 7 2018 these guidelines are hosted on GitHub. Ready to be enhanced by the community and forked to fit specific needs.

On https://github.com/Trivadis/plsql-and-sql-coding-guidelines/releases you find the release information for every version since 1.2.

# Introduction

This document describes rules and recommendations for developing applications using the PL/SQL & SQL Language.

# Scope

This document applies to the PL/SQL and SQL language as used within ORACLE databases and tools, which access ORACLE databases.

# **Document Conventions**

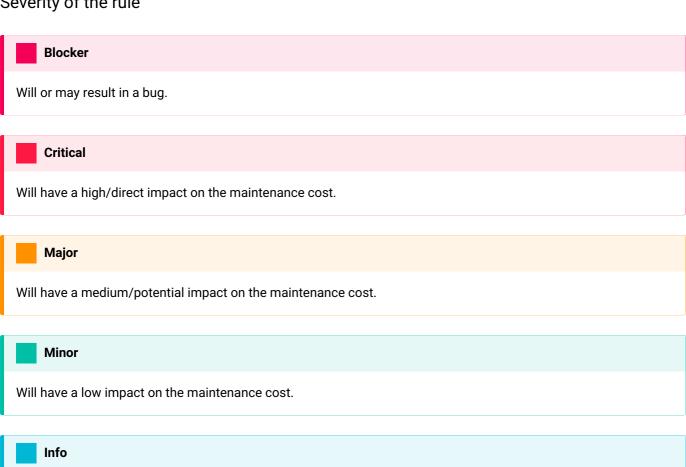
SQALE (Software Quality Assessment based on Lifecycle Expectations) is a method to support the evaluation of a software application source code. It is a generic method, independent of the language and source code analysis tools.

# SQALE characteristics and subcharacteristics

Changeability	The capability of the software product to enable a specified modification to be implemented.  • Architecture related changeability
	Logic related changeability
	Data related changeability
Efficiency	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.  • Memory use  • Processor use
	Network use
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.  • Understandability  • Readability
Portability	The capability of the software product to be transferred from one environment to another.
·	Compiler related portability
	Hardware related portability
	Language related portability
	OS related portability
	Software related portability
	Time zone related portability.

# Reliability The capability of the software product to maintain a specified level of performance when used under specified conditions. · Architecture related reliability Data related reliability **Exception handling** • Fault tolerance Instruction related reliability Logic related reliability Resource related reliability Synchronization related reliability Unit tests coverage. Reusability The capability of the software product to be reused within the development process. Modularity • Transportability. Security The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them. API abuse • Errors (e.g. leaving a system in a vulnerable state) Input validatation and representation Security features. Testability The capability of the software product to enable modified software to be validated. • Integration level testability · Unit level testability.

# Severity of the rule



Very low impact; it is just a remediation cost report.

# Keywords used

Keyword	Meaning
Always	Emphasizes this rule must be enforced.
Never	Emphasizes this action must not happen.
Avoid	Emphasizes that the action should be prevented, but some exceptions may exist.
Try	Emphasizes that the rule should be attempted whenever possible and appropriate.
Example	Precedes text used to illustrate a rule or a recommendation.
Reason	Explains the thoughts and purpose behind a rule or a recommendation.
Restriction	Describes the circumstances to be fulfilled to make use of a rule.

# Why are standards important

For a machine executing a program, code formatting is of no importance. However, for the human eye, well-formatted code is much easier to read. Modern tools can help to implement format and coding rules.

Implementing formatting and coding standards has the following advantages for PL/SQL development:

- Well-formatted code is easier to read, analyze and maintain (not only for the author but also for other developers).
- The developers do not have to define their own guidelines it is already defined.
- The code has a structure that makes it easier to avoid making errors.
- The code is more efficient concerning performance and organization of the whole application.
- The code is more modular and thus easier to use for other applications.

# We have other standards

This document only defines possible standards. These standards are not written in stone, but are meant as guidelines. If standards already exist, and they are different from those in this document, it makes no sense to change them.

# We do not agree with all your standards

There are basically two types of standards.

## 1. Non-controversial

These standards make sense. There is no reason not to follow them. An example of this category is G-2150: Avoid comparisons with NULL value, consider using IS [NOT] NULL.

## 2. Controversial

Almost every rule/guideline falls into this category. An example of this category is 3 space indention. - Why not 2 or 4 or even 8? Why not use tabs? You can argue in favor of all these options. In most cases it does not really matter which option you choose. Being consistent is more important. In this case it will make the code easier to read.

For very controversial rules, we have started to include the reasoning either as a footnote or directly in the text.

Usually it is not helpful to open an issue on GitHub to request to change a highly controversial rule such as the one mentioned. For example, use 2 spaces instead of 3 spaces for an indentation. This leads to a discussion where the people in favor of 4 spaces start to argument as well. There is no right or wrong here. You just have to agree on a standard.

More effective is to fork this repository and amend the standards to fit your needs/expectations.

# Naming Conventions

# General Guidelines

- 1. Never use names with a leading numeric character.
- 2. Always choose meaningful and specific names.
- 3. Avoid using abbreviations unless the full name is excessively long.
- 4. Avoid long abbreviations. Abbreviations should be shorter than 5 characters.
- 5. Any abbreviations must be widely known and accepted.
- 6. Create a glossary with all accepted abbreviations.
- 7. Never use ORACLE keywords as names. A list of ORACLEs keywords may be found in the dictionary view v\$reserved\_words.
- 8. Avoid adding redundant or meaningless prefixes and suffixes to identifiers. Example: create table emp\_table.
- 9. Always use one spoken language (e.g. English, German, French) for all objects in your application.
- 10. Always use the same names for elements with the same meaning.

# Naming Conventions for PL/SQL

In general, ORACLE is not case sensitive with names. A variable named personname is equal to one named PersonName, as well as to one named PERSONNAME. Some products (e.g. TMDA by Trivadis, APEX, OWB) put each name within double quotes (") so ORACLE will treat these names to be case sensitive. Using case sensitive variable names force developers to use double quotes for each reference to the variable. Our recommendation is to write all names in lowercase and to avoid double quoted identifiers.

A widely used convention is to follow a {prefix}variablecontent{suffix} pattern.

The following table shows a possible set of naming conventions.

Identifier	Prefix	Suffix	Example
Global Variable	g_		g_version
Local Variable	1_		l_version
Cursor	c_		c_employees
Record	r_		r_employee
Array / Table	t_		t_employees
Object	0_		o_employee
Cursor Parameter	p_		p_empno

In Parameter	in_		in_empno
Out Parameter	out_		out_ename
In/Out Parameter	io_		io_employee
Record Type Definitions	r_	_type	r_employee_type
Array/Table Type Definitions	t_	_type	t_employees_type
Exception	e_		e_employee_exists
Constants	co_		co_empno
Subtypes		_type	big_string_type

# **Database Object Naming Conventions**

Never enclose object names (table names, column names, etc.) in double quotes to enforce mixed case or lower case object names in the data dictionary.

# **Collection Type**

A collection type should include the name of the collected objects in their name. Furthermore, they should have the suffix \_ct to identify it as a collection.

Optionally prefixed by a project abbreviation.

- employees\_ct
- orders\_ct

# Column

Singular name of what is stored in the column (unless the column data type is a collection, in this case you use plural names)

Add a comment to the database dictionary for every column.

# **Check Constraint**

Table name or table abbreviation followed by the column and/or role of the check constraint, a \_ck and an optional number suffix.

# Examples:

- employees\_salary\_min\_ck
- orders\_mode\_ck

# DML / Instead of Trigger

Choose a naming convention that includes:

## either

- the name of the object the trigger is added to,
- any of the triggering events:
  - \_br\_iud for Before Row on Insert, Update and Delete
  - \_io\_id for Instead of Insert and Delete

or

- the name of the object the trigger is added to,
- the activity done by the trigger,
- the suffix \_trg

# Examples:

- employees\_br\_iud
- orders\_audit\_trg
- orders\_journal\_trg

# Foreign Key Constraint

Table abbreviation followed by referenced table abbreviation followed by a \_fk and an optional number suffix.

- empl\_dept\_fk
- sct\_icmd\_ic\_fk1

# **Function**

Name is built from a verb followed by a noun in general. Nevertheless, it is not sensible to call a function get\_... as a function always gets something.

The name of the function should answer the question "What is the outcome of the function?"

Optionally prefixed by a project abbreviation.

Example: employee\_by\_id

If more than one function provides the same outcome, you have to be more specific with the name.

# Index

Indexes serving a constraint (primary, unique or foreign key) are named accordingly.

Other indexes should have the name of the table and columns (or their purpose) in their name and should also have \_idx as a suffix.

# **Object Type**

The name of an object type is built by its content (singular) followed by a \_ot suffix.

Optionally prefixed by a project abbreviation.

Example: employee\_ot

# Package

Name is built from the content that is contained within the package.

Optionally prefixed by a project abbreviation.

# Examples:

- employees\_api API for the employee table
- logging\_up Utilities including logging support

# **Primary Key Constraint**

Table name or table abbreviation followed by the suffix \_pk .

- employees\_pk
- departments\_pk
- sct\_contracts\_pk

# Procedure

Name is built from a verb followed by a noun. The name of the procedure should answer the question "What is done?"

Procedures and functions are often named with underscores between words because some editors write all letters in uppercase in the object tree, so it is difficult to read them.

Optionally prefixed by a project abbreviation.

# Examples:

- calculate\_salary
- set\_hiredate
- check\_order\_state

# Sequence

Name is built from the table name (or its abbreviation) the sequence serves as primary key generator and the suffix \_seq or the purpose of the sequence followed by a \_seq.

Optionally prefixed by a project abbreviation.

# Examples:

- employees\_seq
- order\_number\_seq

# Synonym

Synonyms should be used to address an object in a foreign schema rather than to rename an object. Therefore, synonyms should share the name with the referenced object.

# System Trigger

Name of the event the trigger is based on.

- · Activity done by the trigger
- Suffix \_trg

- ddl\_audit\_trg
- logon\_trg

# **Table**

Plural<sup>1</sup> name of what is contained in the table (unless the table is designed to always hold one row only – then you should use a singular name).

Suffixed by \_eb when protected by an editioning view.

Add a comment to the database dictionary for every table and every column in the table.

Optionally prefixed by a project abbreviation.

# Examples:

- employees
- departments
- countries\_eb table interfaced by an editioning view named countries
- sct\_contracts
- sct\_contract\_lines
- sct\_incentive\_modules

# Temporary Table (Global Temporary Table)

Naming as described for tables.

Optionally suffixed by \_tmp

Optionally prefixed by a project abbreviation.

# Examples:

- employees\_tmp
- contracts\_tmp

# **Unique Key Constraint**

Table name or table abbreviation followed by the role of the unique key constraint, a \_uk and an optional number suffix.

- employees\_name\_uk
- departments\_deptno\_uk
- sct\_contracts\_uk
- sct\_coli\_uk
- sct\_icmd\_uk1

# View

Plural<sup>1</sup> name of what is contained in the view. Optionally suffixed by an indicator identifying the object as a view (mostly used, when a 1:1 view layer lies above the table layer)

Editioning views are named like the original underlying table to avoid changing the existing application code when introducing edition based redefinition (EBR).

Add a comment to the database dictionary for every view and every column.

Optionally prefixed by a project abbreviation.

- active\_orders
- orders\_v a view to the orders table
- countries an editioning view for table countries\_eb

# Coding Style

# Formatting

# Rules

Rule	Description
1	Keywords and names are written in lowercase <sup>2</sup> .
2	3 space indention <sup>3</sup> .
3	One command per line.
4	Keywords loop, else, elsif, end if, when on a new line.
5	Commas in front of separated elements.
6	Call parameters aligned, operators aligned, values aligned.
7	SQL keywords are right aligned within a SQL command.
8	Within a program unit only line comments are used.
9	Brackets are used when needed or when helpful to clarify a construct.

# Example

```
1 procedure set_salary(in_employee_id in employees.employee_id%type) is
 cursor c_employees(p_employee_id in employees.employee_id%type) is
 3
      select last_name
          ,first_name
 4
 5
              , salary
from employees
where employee_id = p_employee_id
order by last_name
first_name;
9
10
11 r_employee
                    c_employees%rowtype;
      1_new_salary employees.salary%type;
12
13 begin
open c_employees(p_employee_id => in_employee_id);
15
      fetch c_employees into r_employee;
16
     close c_employees;
17
18     new_salary (in_employee_id => in_employee_id
19
                 ,out_salary => l_new_salary);
20
21
     -- Check whether salary has changed
22
     if r_employee.salary <> l_new_salary then
23
         update employees
24
           set salary = l_new_salary
25
          where employee_id = in_employee_id;
26
     end if;
27 end set_salary;
```

# Code Commenting

# Conventions

Inside a program unit only use the line commenting technique -- unless you temporarly deactivate code sections for testing.

To comment the source code for later document generation, comments like /\*\* ... \*/ are used. Within these documentation comments, tags may be used to define the documentation structure.

Tools like ORACLE SQL Developer or PL/SQL Developer include documentation functionality based on a javadoc-like tagging.

# **Commenting Tags**

Tag	Meaning	Example
param	Description of a parameter.	@param in_string input string
return	Description of the return value of a function.	@return result of the calculation
throws	Describe errors that may be raised by the program unit.	@throws NO_DATA_FOUND

# Example

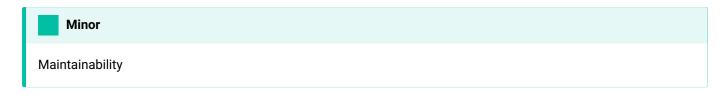
This is an example using the documentation capabilities of SQL Developer.

```
1  /**
2  Check whether we passed a valid sql name
3
4  @param in_name string to be checked
5  @return in_name if the string represents a valid sql name
6  @throws ORA-44003: invalid SQL name
7
8  <b>Call Example:</b>
9  
10      select TVDAssert.valid_sql_name('TEST') from dual;
11      select TVDAssert.valid_sql_name('123') from dual
12  
13 */
```

# Language Usage

# General

G-1010: Try to label your sub blocks.



## Reason

It's a good alternative for comments to indicate the start and end of a named processing.

# Example (bad)

```
1 begin
2 begin
3    null;
4 end;
5
6 begin
7    null;
8 end;
9 end;
```

```
begin

compare_data>>

data>>

data>>
data>>

data>>
data>>
data>>
data>>
data>>
data>>
data>>
data>>
data>>
data>>
data>
data>>
data>
data

data
data>
data>
data>
data>
data>
data>
data>
data>
data

data

data
```

G-1020: Always have a matching loop or block label.



## Reason

Use a label directly in front of loops and nested anonymous blocks:

- To give a name to that portion of code and thereby self-document what it is doing.
- So that you can repeat that name with the end statement of that block or loop.

# Example (bad)

```
1 declare
    i integer;
 2
     co_min_value constant integer := 1;
 3
      co_max_value constant integer := 10;
 4
     co_increment constant integer := 1;
 5
 6 begin
 7
     <<pre><<pre><<pre>c
     begin
8
     null;
9
     end;
10
11
12
     <<pre><<pre><<pre><<pre><<pre><<pre><<pre><<pre>
     begin
13
      null;
14
      end;
15
16
     i := co_min_value;
17
      <<while_loop>>
18
      while (i <= co_max_value)</pre>
19
20
       i := i + co_increment;
21
22
      end loop;
23
     <<basic_loop>>
24
     loop
25
      exit basic_loop;
26
     end loop;
27
28
      <<for_loop>>
29
     for i in co_min_value..co_max_value
30
31
32
       sys.dbms_output.put_line(i);
33
      end loop;
34 end;
35
```

```
1 declare
      2 i integer;
      3
                                co_min_value constant integer := 1;
      co_max_value constant integer := 10;
co_increment constant integer := 1;
       6 begin
      9 null;
10 end prepare_data;
 11
12 <<pre>composed control of the con
16
17     i := co_min_value;
18     <<while_loop>>
19  while (i <= co_max_value)
20  loop
21     i := i + co_increment;
22  end loop while_loop;</pre>
23
28
31
                                  loop
34 end;
35 /
```

G-1030: Avoid defining variables that are not used.



Minor

Efficiency, Maintainability

## Reason

Unused variables decrease the maintainability and readability of your code.

# Example (bad)

```
1 create or replace package body my_package is
    procedure my_proc is
2
      1_last_name employees.last_name%type;
3
4
         1_first_name employees.first_name%type;
5
         co_department_id constant departments.department_id%type := 10;
6
         e_good exception;
     begin
7
     select e.last_name
8
9
          into l_last_name
          from employees e
10
      where e.department_id = co_department_id;
11
12
     exception
     when no_data_found then null; -- handle_no_data_found;
13
14
         when too_many_rows then null; -- handle_too_many_rows;
15 end my_proc;
16 end my_package;
17
```

```
1 create or replace package body my_package is
 procedure my_proc is
 3
        1_last_name employees.last_name%type;
        co_department_id constant departments.department_id%type := 10;
 4
 5
         e_good exception;
 6
    begin
     select e.last_name
7
8
         into l_last_name
9
          from employees e
10
         where e.department_id = co_department_id;
11
       raise e_good;
12
13
     exception
       when no_data_found then null; -- handle_no_data_found;
14
15
         when too_many_rows then null; -- handle_too_many_rows;
16
      end my_proc;
17 end my_package;
18
```



Minor

Maintainability

## Reason

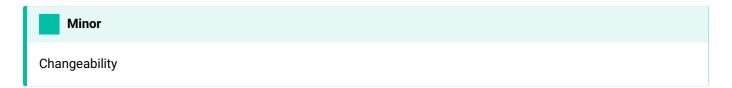
Any part of your code, which is no longer used or cannot be reached, should be eliminated from your programs to simplify the code.

# Example (bad)

```
1 declare
      co_dept_purchasing constant departments.department_id%type := 30;
 3 begin
     if 2=3 then
 4
 5
        null; -- some dead code here
 6
     end if;
 7
      null; -- some enabled code here
 8
9
10
      <<my_loop>>
11
     loop
12
        exit my_loop;
        null; -- some dead code here
13
14
     end loop my_loop;
15
16
      null; -- some other enabled code here
17
18
      case
        when 1 = 1 and x' = y' then
19
           null; -- some dead code here
20
21
        else
22
            null; -- some further enabled code here
23
      end case;
24
25
       <<my_loop2>>
       for r_emp in (select last_name
26
27
                     from employees
                     where department_id = co_dept_purchasing
28
29
                       or commission_pct is not null
                      and 5=6)
30
                   -- "or commission_pct is not null" is dead code
31
32
     loop
       sys.dbms_output.put_line(r_emp.last_name);
33
34
      end loop my_loop2;
35
36
      return;
       null; -- some dead code here
37
38 end;
39
```

```
1 declare
 co_dept_admin constant dept.deptno%type := 10;
 3 begin
 4 null; -- some enabled code here
5 null; -- some other enabled code here
6 null; -- some further enabled code here
 7
8      <<my_loop2>>
9      for r_emp in (select last_name
10
                          from employees
11
                         where department_id = co_dept_admin
12
                          or commission_pct is not null)
13 loop
sys.dbms_output.put_line(r_emp.last_name);
end loop my_loop2;
16 end;
17 /
```

G-1050: Avoid using literals in your code.



## Reason

Literals are often used more than once in your code. Having them defined as a constant reduces typos in your code and improves the maintainability.

All constants should be collated in just one package used as a library. If these constants should be used in SQL too it is good practice to write a deterministic package function for every constant.

In specific situations this rule could lead to an extreme plethora of constants, for example if you use Logger like logger.append\_param(p\_params =>l\_params, p\_name => 'p\_param1\_todo', p\_val => p\_param1\_todo);, where the value for p\_name always should be the name of the variable that is passed to p\_val. For such cases it would be overkill to add constants for every single variable name you are logging, so if you use Logger or similar, consider making that an exception to the rule, just document exactly which exceptions you will allow and stick to them.

## Example (bad)

```
1 declare
2
     l_job employees.job_id%type;
3 begin
    select e.job_id
4
       into l_job
5
       from employees e
6
7
      where e.manager_id is null;
8
     if l_job = 'AD_PRES' then
9
10
        null;
     end if;
11
12 exception
     when no_data_found then
13
        null; -- handle_no_data_found;
14
15
     when too_many_rows then
        null; -- handle_too_many_rows;
16
17 end;
18
```

```
1 create or replace package constants_up is
 co_president constant employees.job_id%type := 'AD_PRES';
 3 end constants_up;
 4 /
 5
 6 declare
 7 l_job employees.job_id%type;
8 begin
9 select e.job_id
into l_job
from employees e
where e.manager_id is null;
13
14    if l_job = constants_up.co_president then
15     null;
16     end if;
17 exception
18 when no_data_found then
       null; -- handle_no_data_found;
19
20     when too_many_rows then
21     null; -- handle_too_many_rows;
22 end;
23 /
```

G-1060: Avoid storing ROWIDs or UROWIDs in database tables.



## Reason

It is an extremely dangerous practice to store rowid's in a table, except for some very limited scenarios of runtime duration. Any manually explicit or system generated implicit table reorganization will reassign the row's rowid and break the data consistency.

Instead of using rowid for later reference to the original row one should use the primary key column(s).

# Example (bad)

```
1 begin
2
   insert into employees_log (employee_id
3
                             ,last_name
4
                             ,first_name
5
                             ,rid)
     select employee_id
6
7
         ,last_name
           ,first_name
8
9
           , rowid
    from employees;
10
11 end;
12 /
```

G-1070: Avoid nesting comment blocks.



Minor

Maintainability

# Reason

Having an end-of-comment within a block comment will end that block-comment. This does not only influence your code but is also very hard to read.

# Example (bad)

```
begin
/* comment one -- nested comment two */
null;
-- comment three /* nested comment four */
null;
end;
//
```

```
begin
/* comment one, comment two */
null;
-- comment three, comment four
null;
end;
// /
```

# Variables & Types

# General

G-2110: Try to use anchored declarations for variables, constants and types.



## REASON

Changing the size of the database column last\_name in the employees table from varchar2(20) to varchar2(30) will result in an error within your code whenever a value larger than the hard coded size is read from the table. This can be avoided using anchored declarations.

## **EXAMPLE (BAD)**

```
1 create or replace package body my_package is
    procedure my_proc is
        1_last_name varchar2(20 char);
      co_first_row constant integer := 1;
5 begin
      select e.last_name
7
          into l_last_name
          from employees e
9
         where rownum = co_first_row;
10
    exception
       when no_data_found then null; -- handle no_data_found
11
        when too_many_rows then null; -- handle too_many_rows (impossible)
12
    end my_proc;
14 end my_package;
15 /
```

```
1 create or replace package body my_package is
procedure my_proc is
3
        l_last_name employees.last_name%type;
4
        co_first_row constant integer := 1;
5 begin
     select e.last_name
6
7
         into l_last_name
8
          from employees e
9
         where rownum = co_first_row;
10
       when no_data_found then null; -- handle no_data_found
11
        when too_many_rows then null; -- handle too_many_rows (impossible)
12
13
     end my_proc;
14 end my_package;
15
```

# G-2120: Try to have a single location to define your types.



Minor

Changeability

## **REASON**

Single point of change when changing the data type. No need to argue where to define types or where to look for existing definitions.

A single location could be either a type specification package or the database (database-defined types).

## **EXAMPLE (BAD)**

```
create or replace package body my_package is
procedure my_proc is
subtype big_string_type is varchar2(1000 char);

l_note big_string_type;
begin
l_note := some_function();
end my_proc;
end my_package;
//
```

```
create or replace package types_up is
subtype big_string_type is varchar2(1000 char);
end types_up;

//

create or replace package body my_package is
procedure my_proc is
l_note types_up.big_string_type;
begin
l_note := some_function();
end my_proc;
end my_package;
//

//

create or replace package body my_package is
procedure my_proc is
l_note types_up.big_string_type;
end my_proc;
```

# G-2130: Try to use subtypes for constructs used often in your code.



## **REASON**

Single point of change when changing the data type.

Your code will be easier to read as the usage of a variable/constant may be derived from its definition.

## **EXAMPLES OF POSSIBLE SUBTYPE DEFINITIONS**

Туре	Usage
ora_name_type	Object corresponding to the ORACLE naming conventions (table, variable, column, package, etc.).
max_vc2_type	String variable with maximal VARCHAR2 size.
array_index_type	Best fitting data type for array navigation.
id_type	Data type used for all primary key (id) columns.

# **EXAMPLE (BAD)**

```
create or replace package body my_package is
procedure my_proc is
l_note varchar2(1000 char);
begin
l_note := some_function();
end my_proc;
end my_package;
///
// charter
```

```
1 create or replace package types_up is
     subtype big_string_type is varchar2(1000 char);
2
3 end types_up;
4 /
5
6 create or replace package body my_package is
7
    procedure my_proc is
8
        l_note types_up.big_string_type;
    begin
9
10
        l_note := some_function();
     end my_proc;
11
12 end my_package;
13
```

G-2135: Avoid assigning values to local variables that are not used by a subsequent statement.



Efficiency, Maintainability, Testability

## **REASON**

Expending resources calculating and assigning values to a local variable and never use the value subsequently is at best a waste, at worst indicative of a mistake that leads to a bug.

## **EXAMPLE (BAD)**

```
1 create or replace package body my_package is
 procedure my_proc is
       co_employee_id constant employees.employee_id%type := 1042;
3
        1_last_name employees.last_name%type;
 4
 5
        1_message varchar2(100 char);
     begin
 6
     select emp.last_name
 7
8
         into l_last_name
9
          from employees emp
10
        where emp.employee_id = co_employee_id;
11
        l_message := 'Hello, ' || l_last_name;
12
13 exception
     when no_data_found then null; -- handle_no_data_found;
14
15
        when too_many_rows then null; -- handle_too_many_rows;
16
     end my_proc;
17 end my_package;
18 /
```

```
1 create or replace package body my_package is
procedure my_proc is
       co_employee_id constant employees.employee_id%type := 1042;
3
4
        l_last_name employees.last_name%type;
5
         1_message varchar2(100 char);
6
     begin
7
     select emp.last_name
8
          into l_last_name
9
          from employees emp
10
         where emp.employee_id = co_employee_id;
11
        l_message := 'Hello, ' || l_last_name;
12
13
        message_api.send_message(l_message);
14
15
     exception
       when no_data_found then null; -- handle_no_data_found;
16
17
         when too_many_rows then null; -- handle_too_many_rows;
18
      end my_proc;
19 end my_package;
20 /
```

# G-2140: Never initialize variables with NULL.



Minor

Maintainability

#### **REASON**

Variables are initialized to null by default.

# **EXAMPLE (BAD)**

```
1 declare
2    l_note big_string_type := null;
3 begin
4    sys.dbms_output.put_line(l_note);
5 end;
6 /
```

```
1 declare
2    l_note big_string_type;
3 begin
4    sys.dbms_output.put_line(l_note);
5 end;
6 /
```

# G-2150: Avoid comparisons with NULL value, consider using IS [NOT] NULL.



Portability, Reliability

#### **REASON**

The null value can cause confusion both from the standpoint of code review and code execution. You must always use the is null or is not null syntax when you need to check if a value is or is not null.

# **EXAMPLE (BAD)**

```
1 declare
2  l_value integer;
3 begin
4  if l_value = null then
5   null;
6  end if;
7 end;
8 /
```

```
1 declare
2  l_value integer;
3 begin
4  if l_value is null then
5   null;
6  end if;
7 end;
8 /
```

G-2160: Avoid initializing variables using functions in the declaration section.



#### **REASON**

If your initialization fails, you will not be able to handle the error in your exceptions block.

#### **EXAMPLE (BAD)**

```
declare
co_department_id constant integer := 100;
l_department_name departments.department_name%type :=
department_api.name_by_id(in_id => co_department_id);
begin
sys.dbms_output.put_line(l_department_name);
end;
// end;
```

```
1 declare
    co_department_id constant integer := 100;
3
     co_unkown_name constant departments.department_name%type := 'unknown';
4
     l_department_name departments.department_name%type;
5 begin
6
     <<init>>
7
    begin
8
        l_department_name := department_api.name_by_id(in_id => co_department_id);
9
    exception
      when value_error then
10
11
           1_department_name := co_unkown_name;
12
    end init;
13
14
      sys.dbms_output.put_line(l_department_name);
15 end;
16
```

#### G-2170: Never overload variables.



#### **REASON**

The readability of your code will be higher when you do not overload variables.

#### **EXAMPLE (BAD)**

```
1 begin
     <<main>>
 3
     declare
        co_main constant user_objects.object_name%type := 'test_main';
 5
        co_sub constant user_objects.object_name%type := 'test_sub';
        co_sep constant user_objects.object_name%type := ' - ';
 6
 7
         l_variable user_objects.object_name%type := co_main;
8
     begin
9
         <<sub>>
10
        declare
11
           l_variable user_objects.object_name%type := co_sub;
12
13
           sys.dbms_output.put_line(l_variable || co_sep || main.l_variable);
14
         end sub:
15
       end main:
16 end:
17 /
```

```
1 begin
 2
     <<main>>
 3
     declare
        co_main constant user_objects.object_name%type := 'test_main';
 5
        co_sub constant user_objects.object_name%type := 'test_sub';
        co_sep constant user_objects.object_name%type := ' - ';
 6
 7
         l_main_variable user_objects.object_name%type := co_main;
8
     begin
9
         <<sub>>
10
        declare
           l_sub_variable user_objects.object_name%type := co_sub;
11
12
         begin
            sys.dbms_output.put_line(l_sub_variable || co_sep || l_main_variable);
13
         end sub;
14
15
       end main;
16 end;
17
```

# G-2180: Never use quoted identifiers.



Maintainability

#### **REASON**

Quoted identifiers make your code hard to read and maintain.

# **EXAMPLE (BAD)**

```
1 declare
2  l_sal_comm  integer;
3  co_my_constant constant integer := 1;
4  e_my_exception exception;
5 begin
6  l_sal_comm := co_my_constant;
7 exception
8  when e_my_exception then
9  null;
10 end;
11 /
```

G-2185: Avoid using overly short names for explicitly or implicitly declared identifiers.



Minor

Maintainability

#### **REASON**

You should ensure that the name you have chosen well defines its purpose and usage. While you can save a few keystrokes typing very short names, the resulting code is obscure and hard for anyone besides the author to understand.

# **EXAMPLE (BAD)**

```
1 declare
2    i integer;
3    c constant integer := 1;
4    e exception;
5 begin
6    i := c;
7 exception
8    when e then
9         null;
10 end;
11 /
```

```
1 declare
2  l_sal_comm  integer;
3  co_my_constant constant integer := 1;
4  e_my_exception exception;
5 begin
6  l_sal_comm := co_my_constant;
7 exception
8  when e_my_exception then
9  null;
10 end;
11 /
```

#### G-2190: Avoid using ROWID or UROWID.



#### **REASON**

Be careful about your use of Oracle-specific data types like rowid and urowid. They might offer a slight improvement in performance over other means of identifying a single row (primary key or unique index value), but that is by no means guaranteed.

Use of rowid or urowid means that your SQL statement will not be portable to other SQL databases. Many developers are also not familiar with these data types, which can make the code harder to maintain.

#### **EXAMPLE (BAD)**

```
1  declare
2     l_department_name departments.department_name%type;
3     l_rowid rowid;
4  begin
5     update departments
6     set department_name = l_department_name
7     where rowid = l_rowid;
8  end;
9  /
```

# Numeric Data Types

G-2210: Avoid declaring NUMBER variables, constants or subtypes with no precision.



#### **REASON**

If you do not specify precision number is defaulted to 38 or the maximum supported by your system, whichever is less. You may well need all this precision, but if you know you do not, you should specify whatever matches your needs.

# **EXAMPLE (BAD)**

```
create or replace package body constants_up is
co_small_increase constant number := 0.1;

function small_increase return number is
begin
return co_small_increase;
end small_increase;
end constants_up;

///
```

```
create or replace package body constants_up is
co_small_increase constant number(5,1) := 0.1;

function small_increase return number is
begin
return co_small_increase;
end small_increase;
end constants_up;

//
```

# G-2220: Try to use PLS\_INTEGER instead of NUMBER for arithmetic operations with integer values.



#### **REASON**

pls\_integer having a length of -2,147,483,648 to 2,147,483,647, on a 32bit system.

There are many reasons to use pls\_integer instead of number:

- pls\_integer uses less memory
- pls\_integer uses machine arithmetic, which is up to three times faster than library arithmetic, which is used by number .

## **EXAMPLE (BAD)**

```
create or replace package body constants_up is
co_big_increase constant number(5,0) := 1;

function big_increase return number is
begin
return co_big_increase;
end big_increase;
end constants_up;

///
```

```
create or replace package body constants_up is
co_big_increase constant pls_integer := 1;

function big_increase return pls_integer is
begin
return co_big_increase;
end big_increase;
end constants_up;

///
```

#### G-2230: Try to use SIMPLE\_INTEGER datatype when appropriate.



#### RESTRICTION

ORACLE 11g or later

#### **REASON**

simple\_integer does no checks on numeric overflow, which results in better performance compared to the other numeric datatypes.

With ORACLE 11g, the new data type <code>simple\_integer</code> has been introduced. It is a sub-type of <code>pls\_integer</code> and covers the same range. The basic difference is that <code>simple\_integer</code> is always not null. When the value of the declared variable is never going to be null then you can declare it as <code>simple\_integer</code>. Another major difference is that you will never face a numeric overflow using <code>simple\_integer</code> as this data type wraps around without giving any error. <code>simple\_integer</code> data type gives major performance boost over <code>pls\_integer</code> when code is compiled in <code>native</code> mode, because arithmetic operations on <code>simple\_integer</code> type are performed directly at the hardware level.

#### **EXAMPLE (BAD)**

```
create or replace package body constants_up is
co_big_increase constant number(5,0) := 1;

function big_increase return number is
begin
return co_big_increase;
end big_increase;
end constants_up;

///
```

```
create or replace package body constants_up is
co_big_increase constant simple_integer := 1;

function big_increase return simple_integer is
begin
return co_big_increase;
end big_increase;
end constants_up;

///
```

# **Character Data Types**

# G-2310: Avoid using CHAR data type.



#### **REASON**

char is a fixed length data type, which should only be used when appropriate. char columns/variables are always filled to its specified lengths; this may lead to unwanted side effects and undesired results.

# **EXAMPLE (BAD)**

```
1 create or replace package types_up
2 is
3 subtype description_type is char(200);
4 end types_up;
5 /
```

```
1 create or replace package types_up
2 is
3 subtype description_type is varchar2(200 char);
4 end types_up;
5 /
```

# G-2320: Never use VARCHAR data type.



#### **REASON**

Do not use the varchar data type. Use the varchar2 data type instead. Although the varchar data type is currently synonymous with varchar2, the varchar data type is scheduled to be redefined as a separate data type used for variable-length character strings compared with different comparison semantics.

# **EXAMPLE (BAD)**

```
1 create or replace package types_up is
2 subtype description_type is varchar(200);
3 end types_up;
4 /
```

```
1 create or replace package types_up is
2 subtype description_type is varchar2(200 char);
3 end types_up;
4 /
```

# G-2330: Never use zero-length strings to substitute NULL.



#### **REASON**

Today zero-length strings and null are currently handled identical by ORACLE. There is no guarantee that this will still be the case in future releases, therefore if you mean null use null.

# **EXAMPLE (BAD)**

```
create or replace package body constants_up is
co_null_string constant varchar2(1) := '';

function null_string return varchar2 is
begin
return co_null_string;
end null_string;
end constants_up;

///
```

```
create or replace package body constants_up is

function empty_string return varchar2 is
begin
return null;
end empty_string;
end constants_up;

///

// Package body constants_up is

// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
// Package body constants_up is
//
```

# G-2340: Always define your VARCHAR2 variables using CHAR SEMANTIC (if not defined anchored).



#### **REASON**

Changes to the nls\_length\_semantic will only be picked up by your code after a recompilation.

In a multibyte environment a varchar2(10) definition may not necessarily hold 10 characters when multibyte characters are part of the value that should be stored, unless the definition was done using the char semantic.

#### **EXAMPLE (BAD)**

```
create or replace package types_up is
subtype description_type is varchar2(200);
end types_up;
//
```

```
1 create or replace package types_up is
2 subtype description_type is varchar2(200 char);
3 end types_up;
4 /
```

# **Boolean Data Types**

G-2410: Try to use boolean data type for values with dual meaning.



Maintainability

#### **REASON**

The use of true and false clarifies that this is a boolean value and makes the code easier to read.

## **EXAMPLE (BAD)**

```
declare
co_newfile constant pls_integer := 1000;
co_oldfile constant pls_integer := 500;
l_bigger pls_integer;
begin
if co_newfile < co_oldfile then
l_bigger := constants_up.co_numeric_true;
else
l_bigger := constants_up.co_numeric_false;
end if;
end;
//</pre>
```

# **EXAMPLE (BETTER)**

```
declare
co_newfile constant pls_integer := 1000;
co_oldfile constant pls_integer := 500;
l_bigger boolean;
begin
if co_newfile < co_oldfile then
l_bigger := true;
else
l_bigger := false;
end if;
end;
//</pre>
```

```
declare
co_newfile constant pls_integer := 1000;
co_oldfile constant pls_integer := 500;
l_bigger boolean;
begin
l_bigger := nvl(co_newfile < co_oldfile, false);
end;
// end;</pre>
```

# Large Objects

# G-2510: Avoid using the LONG and LONG RAW data types.



#### **REASON**

long and long raw data types have been deprecated by ORACLE since version 8i - support might be discontinued in future ORACLE releases.

There are many constraints to long datatypes in comparison to the lob types.

#### **EXAMPLE (BAD)**

```
1 create or replace package example_package is
2
     g_long long;
     g_raw long raw;
3
4
     procedure do_something;
5
6 end example_package;
7 /
8
9 create or replace package body example_package is
10
     procedure do_something is
11
     begin
12
       null;
13
     end do_something;
14 end example_package;
15 /
```

```
1 create or replace package example_package is
    procedure do_something;
3 end example_package;
5
6 create or replace package body example_package is
7
    g_long clob;
8
     g_raw blob;
9
10
     procedure do_something is
11
     begin
12
        null;
13
     end do_something;
14 end example_package;
15
```

# DML & SQL

# General

G-3110: Always specify the target columns when coding an insert statement.



#### REASON

Data structures often change. Having the target columns in your insert statements will lead to change-resistant code.

# **EXAMPLE (BAD)**

```
insert into departments
values (departments_seq.nextval
,'Support'
,100
,10);
```

```
1
   insert into departments (department_id
                          , department_name
2
                          ,manager_id
3
                           , location\_id)
4
5
        values (departments_seq.nextval
6
             ,'Support'
              , 100
7
               ,10);
8
```

G-3120: Always use table aliases when your SQL statement involves more than one source.



#### **REASON**

It is more human readable to use aliases instead of writing columns with no table information.

Especially when using subqueries the omission of table aliases may end in unexpected behavior and result.

## **EXAMPLE (BAD)**

```
1  select last_name
2    ,first_name
3    ,department_name
4   from    employees
5     join departments using (department_id)
6   where extract(month from hire_date) = extract(month from sysdate);
```

#### **EXAMPLE (BETTER)**

#### **EXAMPLE (GOOD)**

Using meaningful aliases improves the readability of your code.

# **EXAMPLE SUBQUERY (BAD)**

If the jobs table has no employee\_id column and employees has one this query will not raise an error but return all rows of the employees table as a subquery is allowed to access columns of all its parent tables - this construct is known as correlated subquery.

## **EXAMPLE SUBQUERY (GOOD)**

If the jobs table has no employee\_id column this query will return an error due to the directive (given by adding the

table alias to the column) to read the <code>employee\_id</code> column from the <code>jobs</code> table.

# G-3130: Try to use ANSI SQL-92 join syntax.



Minor

Maintainability, Portability

#### **REASON**

ANSI SQL-92 join syntax supports the full outer join. A further advantage of the ANSI SQL-92 join syntax is the separation of the join condition from the query filters.

# **EXAMPLE (BAD)**

G-3140: Try to use anchored records as targets for your cursors.



Maintainability, Reliability

#### **REASON**

Using cursor-anchored records as targets for your cursors results enables the possibility of changing the structure of the cursor without regard to the target structure.

## **EXAMPLE (BAD)**

```
1 declare
 2 cursor c_employees is
3
      select employee_id, first_name, last_name
          from employees;
 4
     l_employee_id employees.employee_id%type;
 5
     l_first_name employees.first_name%type;
 6
 7
     1_last_name employees.last_name%type;
8 begin
9
     open c_employees;
     fetch c_employees into l_employee_id, l_first_name, l_last_name;
10
     <<pre><<pre><<pre><<pre><<pre><>>
11
12 while c_employees%found
13 loop
      -- do something with the data
14
        fetch c_employees into l_employee_id, l_first_name, l_last_name;
15
16
     end loop process_employees;
    close c_employees;
17
18 end;
19 /
```

```
1 declare
 cursor c_employees is
 3
       select employee_id, first_name, last_name
 4
          from employees;
 5
     r_employee c_employees%rowtype;
 6 begin
 7
   open c_employees;
 8
     fetch c_employees into r_employee;
9
      <<pre><<pre><<pre><<pre><<pre><>>
10
     while c_employees%found
11
     loop
      -- do something with the data
12
13
        fetch c_employees into r_employee;
14
     end loop process_employees;
15
     close c_employees;
16 end;
17 /
```

#### G-3150: Try to use identity columns for surrogate keys.



Minor

Maintainability, Reliability

#### RESTRICTION

**ORACLE 12c** 

#### **REASON**

An identity column is a surrogate key by design – there is no reason why we should not take advantage of this natural implementation when the keys are generated on database level. Using identity column (and therefore assigning sequences as default values on columns) has a huge performance advantage over a trigger solution.

#### **EXAMPLE (BAD)**

```
1 create table locations (
 2 location_id number(10)
                                           not null
 3 ,location_name varchar2(60 char) not null 
4 ,city varchar2(30 char) not null
3
 5    ,constraint locations_pk primary key (location_id)
6    )
 7 /
8
9 create sequence location_seq start with 1 cache 20
10 /
11
12 create or replace trigger location_br_i
13 before insert on locations
14
       for each row
15 begin
16 :new.location_id := location_seq.nextval;
17 end;
18 /
```

#### **EXAMPLE (GOOD)**

```
1  create table locations (
2  location_id    number(10)  generated always as identity
3  ,location_name    varchar2(60 char) not null
4  ,city    varchar2(30 char) not null
5  ,constraint locations_pk primary key (location_id))
6  /
```

generated always as identity ensures that the location\_id is populated by a sequence. It is not possible to override the behavior in the application.

However, if you use a framework that produces an insert statement including the surrogate key column, and you cannot change this behavior, then you have to use the generated by default on null as identity option. This has the downside that the application may pass a value, which might lead to an immediate or delayed ORA-00001: unique constraint violated error.

#### G-3160: Avoid visible virtual columns.



Major

Maintainability, Reliability

#### RESTRICTION

**ORACLE 12c** 

#### **REASON**

In contrast to visible columns, invisible columns are not part of a record defined using <code>%rowtype</code> construct. This is helpful as a virtual column may not be programmatically populated. If your virtual column is visible you have to manually define the record types used in API packages to be able to exclude them from being part of the record definition.

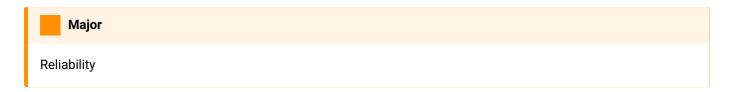
Invisible columns may be accessed by explicitly adding them to the column list in a select statement.

#### **EXAMPLE (BAD)**

```
1 alter table employees
2 add total_salary generated always as (salary + nvl(commission_pct, ♥) * salary)
3 /
4
5 declare
     r_employee employees%rowtype;
7
     l_id employees.employee_id%type := 107;
8 begin
9
     r_employee := employee_api.employee_by_id(l_id);
10
     r_employee.salary := r_employee.salary * constants_up.small_increase();
11
12
     update employees
13
       set row = r_employee
14
      where employee_id = l_id;
15 end;
16 /
17
18 Error report -
19 ORA-54017: update operation disallowed on virtual columns
20 ORA-06512: at line 9
```

```
1 alter table employees
 2 add total_salary invisible generated always as
 3
        (salary + nvl(commission_pct,0) * salary)
 4 /
 5
 6 declare
 7 r_employee employees%rowtype;
8
      co_id constant employees.employee_id%type := 107;
9 begin
r_employee := employee_api.employee_by_id(co_id);
r_employee.salary := r_employee.salary * constants_up.small_increase();
12
13 update employees
14 set row = r_employee
15 where employee_id = co_id;
16 end;
17 /
```

G-3170: Always use DEFAULT ON NULL declarations to assign default values to table columns if you refuse to store NULL values.



#### RESTRICTION

**ORACLE 12c** 

#### **REASON**

Default values have been nullifiable until ORACLE 12c. Meaning any tool sending null as a value for a column having a default value bypassed the default value. Starting with ORACLE 12c default definitions may have an on null definition in addition, which will assign the default value in case of a null value too.

# **EXAMPLE (BAD)**

```
1 create table null_test (
    test_case number(2) not null
 3
     , column_defaulted varchar2(10 char) default 'Default')
 4
 5 insert into null_test(test_case, column_defaulted) values (1,'Value');
 6 insert into null_test(test_case, column_defaulted) values (2,default);
 7
   insert into null_test(test_case, column_defaulted) values (3, null);
8
9 select * from null_test;
10
11 TEST_CASE COLUMN_DEF
12
    1 Value2 Default
13
14
15
```

```
1 create table null_test (
    test_case number(2) not null
 2
     , column_defaulted varchar2(10 char) default on null 'Default')
 3
 4 /
 5
   insert into null_test(test_case, column_defaulted) values (1,'Value');
   insert into null_test(test_case, column_defaulted) values (2, default);
   insert into null_test(test_case, column_defaulted) values (3, null);
 8
9 select * from null_test;
10
    TEST_CASE COLUMN_DEF
11
12
       1 Value
13
           2 Default
14
            3 Default
15
```

G-3180: Always specify column names instead of positional references in ORDER BY clauses.



#### **REASON**

If you change your select list afterwards the order by will still work but order your rows differently, when not changing the positional number. Furthermore, it is not comfortable to the readers of the code, if they have to count the columns in the select list to know the way the result is ordered.

# **EXAMPLE (BAD)**

```
1  select upper(first_name)
2    ,last_name
3    ,salary
4    ,hire_date
5   from employees
6   order by 4,1,3;
```

#### G-3190: Avoid using NATURAL JOIN.



Changeability, Reliability

#### **REASON**

A natural join joins tables on equally named columns. This may comfortably fit on first sight, but adding logging columns to a table (changed\_by, changed\_date) will result in inappropriate join conditions.

## **EXAMPLE (BAD)**

```
1 select department_name
2 ,last_name
6 ,last_name;
7
                       LAST_NAME
8 DEPARTMENT_NAME
                                         FIRST_NAME
9 -----
10 Accounting
                       Gietz
                                         William
11 Executive
                        De Haan
                                          Lex
12 ...
13
14 alter table departments add modified_at date default on null sysdate;
15 alter table employees add modified_at date default on null sysdate;
16
17    select department_name
18 ,last_name
19
      ,first_name
20 from employees natural join departments
21 order by department_name
, last_name;
23
24 No data found
```

```
1 select d.department_name
4 from employees e
5    join departments d on (e.department_id = d.department_id)
6 order by d.department_name
7
   ,e.last_name;
8
9 DEPARTMENT_NAME
                     LAST_NAME
                                      FIRST_NAME
10 -----
11 Accounting
                                    William
                     Gietz
12 Executive
                     De Haan
                                       Lex
13 ...
```

# G-3195: Always use wildcards in a LIKE clause.



Maintainability

#### **REASON**

Using like without at least one wildcard (% or \_ ) is unclear to a maintainer whether a wildcard is forgotten or it is meant as equality test. A common antipattern is also to forget that an underscore is a wildcard, so using like instead of equal can return unwanted rows. If the char datatype is involved, there is also the danger of like not using blank padded comparison where equal will. Depending on use case, you should either remember at least one wildcard or use normal equality operator.

# **EXAMPLE (BAD)**

# **EXAMPLE (GOOD)**

# **Bulk Operations**

G-3210: Always use BULK OPERATIONS (BULK COLLECT, FORALL) whenever you have to execute a DML statement for more than 4 times.



#### **REASON**

Context switches between PL/SQL and SQL are extremely costly. BULK Operations reduce the number of switches by passing an array to the SQL engine, which is used to execute the given statements repeatedly.

(Depending on the PLSQL\_OPTIMIZE\_LEVEL parameter a conversion to BULK COLLECT will be done by the PL/SQL compiler automatically.)

# **EXAMPLE (BAD)**

```
declare
2
     t_employee_ids employee_api.t_employee_ids_type;
3
       co_increase constant employees.salary%type := 0.1;
4
      co_department_id constant departments.department_id%type := 10;
5 begin
6
    t_employee_ids := employee_api.employee_ids_by_department(
7
                            id_in => co_department_id
8
9
      <<pre><<pre><<pre>cess_employees>>
     for i in 1..t_employee_ids.count()
10
11
12
          update employees
13
             set salary = salary + (salary * co_increase)
           where employee_id = t_employee_ids(i);
14
       end loop process_employees;
15
16 end;
17
```

```
1 declare
2
    t_employee_ids employee_api.t_employee_ids_type;
       co_increase constant employees.salary%type := 0.1;
3
4
      co_department_id constant departments.department_id%type := 10;
5
6
      t_employee_ids := employee_api.employee_ids_by_department(
7
                           id_in => co_department_id
8
9
       <<pre><<pre>cess_employees>>
10
       forall i in 1..t_employee_ids.count()
11
          update employees
12
             set salary = salary + (salary * co_increase)
          where employee_id = t_employee_ids(i);
13
14
   end;
15
```

# **Transaction Control**

# G-3310: Never commit within a cursor loop.



#### **REASON**

Doing frequent commits within a cursor loop (all types of loops over cursors, whether implicit cursor for loop or loop with explicit fetch from cursor or cursor variable) risks not being able to complete due to ORA-01555, gives bad performance, and risks that the work is left in an unknown half-finished state and cannot be restarted.

- If the work belongs together (an atomic transaction) the commit should be moved to after the loop. Or even better if the logic can be rewritten to a single DML statement on all relevant rows instead of a loop, committing after the single statement.
- If each loop iteration is a self-contained atomic transaction, consider instead to populate a collection of transactions to be done (taking restartability into account by collection population), loop over that collection (instead of looping over a cursor) and call a procedure (that contains the transaction logic and the commit) in the loop (see also G-3320).

#### **EXAMPLE (BAD)**

```
declare
1
   l_counter integer := 0;
2
3
      1_discount discount.percentage%type;
4 begin
   for r_order in (
5
6
       select o.order_id, o.customer_id
7
          from orders o
8
         where o.order_status = 'New'
     ) loop
9
10
        l_discount := sales_api.calculate_discount(p_customer_id => r_order.customer_id);
11
      update order_lines ol
12
           set ol.discount = l_discount
13
         where ol.order_id = r_order.order_id;
14
15
         1_counter := 1_counter + 1;
16
17
        if l_{counter} = 100 then
         commit;
18
19
           l_{counter} := 0;
      end if;
20
21
     end loop;
     if l_counter > 0 then
22
23
       commit;
24
     end if;
25 end;
26 /
```

```
1 declare
 2 l_discount discount.percentage%type;
3 begin
4 for r_order in (
5
      select o.order_id, o.customer_id
 6
         from orders o
7
8 ) loop
1_d:
         where o.order_status = 'New'
      l_discount := sales_api.calculate_discount(p_customer_id => r_order.customer_id);
10
11     update order_lines ol
         set ol.discount = l_discount
12
13
         where ol.order_id = r_order.order_id;
14 end loop;
15
16 commit;
17 end;
18 /
```

#### **EXAMPLE (BETTER)**

(Assuming suitable foreign key relationship exists to allow updating a join.)

```
1 begin
2 update (
      select o.customer_id, ol.discount
3
4
         from orders o
5
         join order_lines ol
             on ol.order_id = o.order_id
6
7
         where o.order_status = 'New'
     )
8
9
     set discount = sales_api.calculate_discount(p_customer_id => customer_id);
10
     commit;
11
12 end;
13 /
```

G-3320: Try to move transactions within a non-cursor loop into procedures.



Major

Maintainability, Reusability, Testability

#### **REASON**

Commit inside a non-cursor loop (other loop types than loops over cursors - see also G-3310) is either a self-contained atomic transaction, or it is a chunk (with suitable restartability handling) of very large data manipulations. In either case encapsulating the transaction in a procedure is good modularity, enabling reuse and testing of a single call.

# **EXAMPLE (BAD)**

```
begin
2
     for l_counter in 1..5 loop
         insert into headers (id, text) values (l_counter, 'Number '||l_counter);
3
4
     insert into lines (header_id, line_no, text)
select l_counter, rownum, 'Line '||rownum
5
6
7
           from dual
       connect by level <= 3;
8
9
10
        commit;
    end loop;
11
12 end;
13 /
```

```
1 declare
2
     procedure create_rows (
3
          p_header_id in headers.id%type
      ) is
4
5
     begin
      insert into headers (id, text) values (p_header_id, 'Number '||p_header_id);
6
7
     insert into lines (header_id, line_no, text)
select p_header_id, rownum, 'Line '||rownum
8
9
10
          from dual
       connect by level <= 3;</pre>
11
12
13
        commit;
14
       end;
15 begin
16
    for l_counter in 1..5 loop
17
        create_rows(l_counter);
18
       end loop;
19
   end;
20
```

# **Control Structures**

# **CURSOR**

G-4110: Always use %NOTFOUND instead of NOT %FOUND to check whether a cursor returned data.



#### REASON

The readability of your code will be higher when you avoid negative sentences.

# **EXAMPLE (BAD)**

```
1 declare
 cursor c_employees is
     select last_name
 3
 4
           ,first_name
 5
          from employees
 6
         where commission_pct is not null;
 7
8
     r_employee c_employees%rowtype;
9 begin
10
     open c_employees;
11
      <<read_employees>>
12
   <<rea
13
     fetch c_employees into r_employee;
exit read_employees when not c_employees%found;
14
15
     end loop read_employees;
16
17
18
     close c_employees;
19 end;
20 /
```

```
1 declare
 2
    cursor c_employees is
     select last_name
 3
            ,first_name
 4
 5
           from employees
 6
          where commission_pct is not null;
 7
 8
      r_employee c_employees%rowtype;
9 begin
10
     open c_employees;
11
13 loop
14
       <<read_employees>>
      fetch c_employees into r_employee;
exit read_employees when c_employees%notfound;
15
16
     end loop read_employees;
17
18
      close c_employees;
19 end;
20 /
```

G-4120: Avoid using %NOTFOUND directly after the FETCH when working with BULK OPERATIONS and LIMIT clause.



#### **REASON**

%notfound is set to true as soon as less than the number of rows defined by the limit clause has been read.

#### **EXAMPLE (BAD)**

The employees table holds 107 rows. The example below will only show 100 rows as the cursor attribute not found is set to true as soon as the number of rows to be fetched defined by the limit clause is not fulfilled anymore.

```
1 declare
 2
     cursor c_employees is
 3
         select *
 4
           from employees
 5
          order by employee_id;
 6
 7
      type t_employees_type is table of c_employees%rowtype;
       t_employees t_employees_type;
 8
 9
       co_bulk_size constant simple_integer := 10;
10 begin
      open c_employees;
11
12
13
      <<pre><<pre><<pre><<pre><<pre><>>
     loop
14
      fetch c_employees bulk collect into t_employees limit co_bulk_size;
15
16
        exit process_employees when c_employees%notfound;
17
         <<display_employees>>
18
         for i in 1..t_employees.count()
19
         loop
20
21
             sys.dbms\_output.put\_line(t\_employees(i).last\_name);\\
22
         end loop display_employees;
23
      end loop process_employees;
24
25
       close c_employees;
26 end;
27
```

# **EXAMPLE (BETTER)**

This example will show all 107 rows but execute one fetch too much (12 instead of 11).

```
1 declare
 cursor c_employees is
 3
       select *
 4
          from employees
 5
         order by employee_id;
 6
 7
      type t_employees_type is table of c_employees%rowtype;
8
      t_employees t_employees_type;
9
       co_bulk_size constant simple_integer := 10;
10 begin
     open c_employees;
11
12
13
      <<pre><<pre><<pre><<pre><<pre><</pre>
14
      loop
15
       fetch c_employees bulk collect into t_employees limit co_bulk_size;
16
        exit process_employees when t_employees.count() = 0;
17
        <<display_employees>>
18
         for i in 1..t_employees.count()
19
         loop
20
            sys.dbms_output.put_line(t_employees(i).last_name);
21
          end loop display_employees;
22
       end loop process_employees;
23
24
      close c_employees;
25 end;
26 /
```

# **EXAMPLE (GOOD)**

This example does the trick (11 fetches only to process all rows)

```
1 declare
     cursor c_employees is
 2
        select *
 3
          from employees
 4
 5
          order by employee_id;
 6
7
      type t_employees_type is table of c_employees%rowtype;
8
       t_employees t_employees_type;
9
      co_bulk_size constant simple_integer := 10;
10 begin
11
      open c_employees;
12
13
       <<pre><<pre><<pre>cess_employees>>
      loop
14
         fetch c_employees bulk collect into t_employees limit co_bulk_size;
15
16
         <<display_employees>>
17
         for i in 1..t_employees.count()
18
         loop
            sys.dbms_output.put_line(t_employees(i).last_name);
19
20
         end loop display_employees;
         exit process_employees when t_employees.count() <> co_bulk_size;
21
22
       end loop process_employees;
23
24
       close c_employees;
25 end;
26
```



Efficiency, Reliability

#### **REASON**

Any cursors left open can consume additional memory space (i.e. SGA) within the database instance, potentially in both the shared and private SQL pools. Furthermore, failure to explicitly close cursors may also cause the owning session to exceed its maximum limit of open cursors (as specified by the open\_cursors database initialization parameter), potentially resulting in the Oracle error of "ORA-01000: maximum open cursors exceeded".

#### **EXAMPLE (BAD)**

```
1 create or replace package body employee_api as
     function department_salary (in_dept_id in departments.department_id%type)
 3
         return number is
        cursor c_department_salary(p_dept_id in departments.department_id%type) is
 4
 5
           select sum(salary) as sum_salary
 6
              from employees
 7
             where department_id = p_dept_id;
 8
         r_department_salary c_department_salary%rowtype;
 9
     begin
10
        open c_department_salary(p_dept_id => in_dept_id);
11
         fetch c_department_salary into r_department_salary;
12
13
         return r_department_salary.sum_salary;
14
      end department_salary;
15 end employee_api;
16 /
```

```
1 create or replace package body employee_api as
     function department_salary (in_dept_id in departments.department_id%type)
3
        return number is
        cursor c_department_salary(p_dept_id in departments.department_id%type) is
           select sum(salary) as sum_salary
6
              from employees
7
             where department_id = p_dept_id;
8
         r_department_salary c_department_salary%rowtype;
9
     begin
        open c_department_salary(p_dept_id => in_dept_id);
10
11
        fetch c_department_salary into r_department_salary;
12
        close c_department_salary;
13
         return r_department_salary.sum_salary;
14
      end department_salary;
15 end employee_api;
16
```

G-4140: Avoid executing any statements between a SQL operation and the usage of an implicit cursor attribute.



#### **REASON**

Oracle provides a variety of cursor attributes (like %found and %rowcount) that can be used to obtain information about the status of a cursor, either implicit or explicit.

You should avoid inserting any statements between the cursor operation and the use of an attribute against that cursor. Interposing such a statement can affect the value returned by the attribute, thereby potentially corrupting the logic of your program.

In the following example, a procedure call is inserted between the delete statement and a check for the value of sql%rowcount, which returns the number of rows modified by that last SQL statement executed in the session. If this procedure includes a commit / rollback or another implicit cursor the value of sql%rowcount is affected.

#### **EXAMPLE (BAD)**

```
1 create or replace package body employee_api as
2
      co_one constant simple_integer := 1;
3
      procedure process_dept(in_dept_id in departments.department_id%type) is
4
5
     begin
         null;
6
7
      end process_dept;
8
9
       procedure remove_employee (in_employee_id in employees.employee_id%type) is
10
                        employees.department_id%type;
         l_dept_id
11
       begin
         delete from employees
12
          where employee_id = in_employee_id
13
          returning department_id into l_dept_id;
14
15
         process_dept(in_dept_id => l_dept_id);
16
17
         if sql%rowcount > co_one then
18
19
            -- too many rows deleted.
20
            rollback;
         end if;
21
22
       end remove_employee;
23 end employee_api;
24
```

```
1 create or replace package body employee_api as
 co_one constant simple_integer := 1;
 3
     procedure process_dept(in_dept_id in departments.department_id%type) is
4
 5
     begin
6     null;
7     end process_dept;
8
9
     procedure remove_employee (in_employee_id in employees.employee_id%type) is
     l_dept_id employees.department_id%type;
10
11
         l_deleted_emps simple_integer;
12 begin
13
     delete from employees
14
        where employee_id = in_employee_id
15
         returning department_id into l_dept_id;
16
17
        1_deleted_emps := sql%rowcount;
18
19
        process_dept(in_dept_id => l_dept_id);
20
21
        if l_deleted_emps > co_one then
         -- too many rows deleted.
22
23
           rollback;
      end if;
24
25 end remove_employee;
26 end employee_api;
27 /
```

# CASE / IF / DECODE / NVL / NVL2 / COALESCE

G-4210: Try to use CASE rather than an IF statement with multiple ELSIF paths.



#### **REASON**

if statements containing multiple elsif tend to become complex quickly.

#### **EXAMPLE (BAD)**

```
1 declare
2 l_color varchar2(7 char);
3 begin
    if l_color = constants_up.co_red then
     my_package.do_red();
5
6 elsif l_color = constants_up.co_blue then
    my_package.do_blue();
7
    elsif l_color = constants_up.co_black then
8
9
      my_package.do_black();
10 end if;
11 end;
12 /
```

```
1  declare
2    l_color types_up.color_code_type;
3  begin
4    case l_color
5    when constants_up.co_red then
6        my_package.do_red();
7    when constants_up.co_blue then
8        my_package.do_blue();
9    when constants_up.co_black then
10        my_package.do_black();
11    else null;
12    end case;
13  end;
14  /
```

## G-4220: Try to use CASE rather than DECODE.



Minor

Maintainability, Portability

#### **REASON**

decode is an ORACLE specific function hard to understand and restricted to SQL only. The "newer" case function is much more common, has a better readability and may be used within PL/SQL too. Be careful that decode can handle null values, which the simple case cannot - for such cases you must use the searched case and is null instead.

## **EXAMPLE (BAD)**

## **EXAMPLE (GOOD)**

## **EXAMPLE (BAD)**

```
1 select decode(dummy, 'X', 1
2 , 'Y', 2
3 , null, -1
4 , 0)
5 from dual;
```

G-4230: Always use a COALESCE instead of a NVL command, if parameter 2 of the NVL function is a function call or a SELECT statement.



**Critical** 

Efficiency, Reliability

#### REASON

The nv1 function always evaluates both parameters before deciding which one to use. This can be harmful if parameter 2 is either a function call or a select statement, as it will be executed regardless of whether parameter 1 contains a null value or not.

The coalesce function does not have this drawback.

## **EXAMPLE (BAD)**

```
1 select nvl(dummy, my_package.expensive_null(value_in => dummy))
2 from dual;
```

```
1 select coalesce(dummy, my_package.expensive_null(value_in => dummy))
2 from dual;
```

G-4240: Always use a CASE instead of a NVL2 command if parameter 2 or 3 of NVL2 is either a function call or a SELECT statement.



Critical

Efficiency, Reliability

#### REASON

The nv12 function always evaluates all parameters before deciding which one to use. This can be harmful, if parameter 2 or 3 is either a function call or a select statement, as they will be executed regardless of whether parameter 1 contains a null value or not.

## **EXAMPLE (BAD)**

G-4250: Avoid using identical conditions in different branches of the same IF or CASE statement.



Major

Maintainability, Reliability, Testability

#### **REASON**

Conditions are evaluated top to bottom in branches of a case statement or chain of if/elsif statements. The first condition to evaluate as true leads to that branch being executed, the rest will never execute. Having an identical duplicated condition in another branch will never be reached and will be dead code.

## **EXAMPLE (BAD)**

```
1 declare
 2
    l_color types_up.color_code_type;
 3 begin
    case l_color
 4
       when constants_up.co_red
 5
 6
           my_package.do_red();
    my_package.do_red();
when constants_up.co_blue then
  my_package.do_blue();
when constants_up.co_red then -- never reached
 7
8
9
10
           else null;
11
   end case;
12
13 end;
14 /
```

```
1 declare
 2
    1_color types_up.color_code_type;
3 begin
    case l_color
 4
 5
         when constants_up.co_red
 6
             my_package.do_red();
    my_package.do_red();
when constants_up.co_blue then
my_package.do_blue();
when constants_up.co_black then
 7
8
9
10
             my_package.do_black();
        else null;
11
12
        end case;
13 end;
14
```

# G-4260: Avoid inverting boolean conditions with NOT.



Minor

Maintainability, Testability

#### **REASON**

It is more readable to use the opposite comparison operator instead of inverting the comparison with not.

## **EXAMPLE (BAD)**

```
1 declare
2    l_color varchar2(7 char);
3 begin
4    if not l_color != constants_up.co_red then
5        my_package.do_red();
6    end if;
7 end;
8 /
```

```
1 declare
2    l_color types_up.color_code_type;
3 begin
4    if l_color = constants_up.co_red then
5        my_package.do_red();
6    end if;
7 end;
8 /
```

## G-4270: Avoid comparing boolean values to boolean literals.



Minor

Maintainability, Testability

#### **REASON**

It is more readable to simply use the boolean value as a condition itself, rather than use a comparison condition comparing the boolean value to the literals true or false.

## **EXAMPLE (BAD)**

```
1 declare
2    l_string    varchar2(10 char) := '42';
3    l_is_valid boolean;
4 begin
5    l_is_valid := my_package.is_valid_number(l_string);
6    if l_is_valid = true then
7         my_package.convert_number(l_string);
8    end if;
9    end;
10 /
```

```
1 declare
2    l_string    varchar2(10 char) := '42';
3    l_is_valid boolean;
4 begin
5    l_is_valid := my_package.is_valid_number(l_string);
6    if l_is_valid then
7        my_package.convert_number(l_string);
8    end if;
9 end;
10 /
```

## Flow Control

## G-4310: Never use GOTO statements in your code.



Major

Maintainability, Testability

#### **REASON**

Code containing gotos is hard to format. Indentation should be used to show logical structure, and gotos have an effect on logical structure. Using indentation to show the logical structure of a goto and its target, however, is difficult or impossible. (...)

Use of gotos is a matter of religion. My dogma is that in modern languages, you can easily replace nine out of ten gotos with equivalent sequential constructs. In these simple cases, you should replace gotos out of habit. In the hard cases, you can still exorcise the goto in nine out of ten cases: You can break the code into smaller routines, use try-finally, use nested ifs, test and retest a status variable, or restructure a conditional. Eliminating the goto is harder in these cases, but it's good mental exercise (...).

-- McConnell, Steve C. (2004). Code Complete. Second Edition. Microsoft Press.

#### **EXAMPLE (BAD)**

```
1 create or replace package body my_package is
2
       procedure password_check (in_password in varchar2) is
3
        co_digitarray constant string(10 char) := '0123456789';
4
        co_lower_bound constant simple_integer := 1;
5
        co_errno constant simple_integer := -20501;
6
                      constant string(100 char) := 'Password must contain a digit.';
        co errmsa
        l_isdigit
l_len_pw
7
                      boolean := false;
8
                      pls_integer;
9
         l_len_array pls_integer;
10
     begin
11
        1_len_pw := length(in_password);
12
         l_len_array := length(co_digitarray);
13
14
         <<check_digit>>
15
         for i in co_lower_bound .. l_len_array
16
17
            <<check_pw_char>>
18
           for j in co_lower_bound .. l_len_pw
19
20
               if substr(in_password, j, 1) = substr(co_digitarray, i, 1) then
21
                  l_isdigit := true;
22
                  goto check_other_things;
23
               end if;
24
            end loop check_pw_char;
25
         end loop check_digit;
26
27
         <<check_other_things>>
28
         null;
29
30
          if not l_isdigit then
31
            raise_application_error(co_errno, co_errmsg);
32
         end if;
33
       end password_check;
34
  end my_package;
35
```

#### **EXAMPLE (BETTER)**

```
1 create or replace package body my_package is
      procedure password_check (in_password in varchar2) is
        co_digitarray constant string(10 char) := '0123456789';
 3
        co_lower_bound constant simple_integer := 1;
 4
        co_errno constant simple_integer := -20501;
 5
                     constant string(100 char) := 'Password must contain a digit.';
 6
        co_errmsg
7
                     boolean := false;
        l_isdigit
                    pls_integer;
8
        l_len_pw
        l_len_array pls_integer;
9
10
     begin
11
        1_len_pw := length(in_password);
12
        1_len_array := length(co_digitarray);
13
       <<check_digit>>
14
15
        for i in co_lower_bound .. l_len_array
16
         loop
17
           <<check_pw_char>>
18
           for j in co_lower_bound .. l_len_pw
19
           loop
20
              if substr(in_password, j, 1) = substr(co_digitarray, i, 1) then
21
                 l_isdigit := true;
22
                 exit check_digit; -- early exit condition
23
              end if:
24
           end loop check_pw_char;
25
      end loop check_digit;
26
        <<check_other_things>>
27
28
        null;
29
30
        if not l_isdigit then
           raise_application_error(co_errno, co_errmsg);
31
32
         end if;
33
     end password_check;
34 end my_package;
35 /
```

```
1 create or replace package body my_package is
2
     procedure password_check (in_password in varchar2) is
        co_digitpattern constant string(10 char) := '\d';
3
        co_errno constant simple_integer := -20501;
4
5
                      constant string(100 char) := 'Password must contain a digit.';
        co_errmsg
6
     begin
7
        if not regexp_like(in_password, co_digitpattern)
8
9
            raise_application_error(co_errno, co_errmsg);
10
         end if:
11
     end password_check;
12 end my_package;
13
```

## G-4320: Always label your loops.



Minor

Maintainability

#### **REASON**

It's a good alternative for comments to indicate the start and end of a named loop processing.

#### **EXAMPLE (BAD)**

```
1 declare
     i integer;
 3
     co_min_value constant simple_integer := 1;
 4 co_max_value constant simple_integer := 10;
 5
     co_increment constant simple_integer := 1;
 6 begin
7
     i := co_min_value;
8
     while (i <= co_max_value)</pre>
9
     loop
10
       i := i + co_increment;
     end loop;
11
12
13
     loop
14
        exit;
15
     end loop;
16
17
      for i in co_min_value..co_max_value
18
19
       sys.dbms_output.put_line(i);
     end loop;
20
21
22
      for r_employee in (select last_name from employees)
23
24
         sys.dbms_output.put_line(r_employee.last_name);
25
      end loop;
26 end;
27
```

```
1 declare
 2 i integer;
 3
     co_min_value constant simple_integer := 1;
 co_max_value constant simple_integer := 10;
co_increment constant simple_integer := 1;
 6 begin
 7 i := co_min_value;
 8
      <<while_loop>>
9  while (i <= co_max_value)
10  loop
11     i := i + co_increment;
12  end loop while_loop;</pre>
13
14 <<basic_loop>>
15 loop
16 exit basic_loop;
17 end loop basic_loop;
18
24
27
                           from employees)
28 loop
29
      sys.dbms_output.put_line(r_employee.last_name);
30
      end loop process_employees;
31 end;
32 /
```

G-4330: Always use a CURSOR FOR loop to process the complete cursor results unless you are using bulk operations.



#### REASON

It is easier for the reader to see, that the complete data set is processed. Using SQL to define the data to be processed is easier to maintain and typically faster than using conditional processing within the loop.

Since an exit statement is similar to a goto statement, it should be avoided, whenever possible.

#### **EXAMPLE (BAD)**

```
1 declare
 cursor c_employees is
3
      select employee_id, last_name
4
          from employees;
5 r_employee c_employees%rowtype;
 6 begin
 7     open c_employees;
8
9
     <<read_employees>>
10 loop
     fetch c_employees into r_employee;
11
      exit read_employees when c_employees%notfound;
12
13
       sys.dbms_output.put_line(r_employee.last_name);
14
     end loop read_employees;
15
16
      close c_employees;
17 end;
18 /
```

```
1 declare
cursor c_employees is
3
       select employee_id, last_name
         from employees;
4
5 begin
6 <<read_employees>>
     for r_employee in c_employees
7
8
9
         sys.dbms_output.put_line(r_employee.last_name);
10
      end loop read_employees;
11 end;
12 /
```

#### G-4340: Always use a NUMERIC FOR loop to process a dense array.



Minor

Maintainability

#### **REASON**

It is easier for the reader to see, that the complete array is processed.

Since an exit statement is similar to a goto statement, it should be avoided, whenever possible.

#### **EXAMPLE (BAD)**

```
1 declare
2
     type t_employee_type is varray(10) of employees.employee_id%type;
3
      t_employees t_employee_type;
4
      co_himuro
                   constant integer := 118;
5
      co_livingston constant integer := 177;
6
      co_min_value constant simple_integer := 1;
7
      co_increment constant simple_integer := 1;
8
      i pls_integer;
9 begin
10
    t_employees := t_employee_type(co_himuro, co_livingston);
11
                  := co_min_value;
12
13
      <<pre><<pre><<pre>cess_employees>>
14
     loop
        exit process_employees when i > t_employees.count();
15
        sys.dbms_output.put_line(t_employees(i));
16
          i := i + co_increment;
18
      end loop process_employees;
19
   end;
20
```

```
1 declare
2
      type t_employee_type is varray(10) of employees.employee_id%type;
3
      t_employees t_employee_type;
4
      co_himuro
                    constant integer := 118;
5
      co_livingston constant integer := 177;
6 begin
7
      t_employees := t_employee_type(co_himuro, co_livingston);
8
9
       <<pre><<pre><<pre>cess_employees>>
10
       for i in 1..t_employees.count()
      loop
11
        sys.dbms_output.put_line(t_employees(i));
12
      end loop process_employees;
13
14 end;
15
```

G-4350: Always use 1 as lower and COUNT() as upper bound when looping through a dense array.



#### **REASON**

Doing so will not raise a value\_error if the array you are looping through is empty. If you want to use first()..last() you need to check the array for emptiness beforehand to avoid the raise of value\_error.

#### **EXAMPLE (BAD)**

```
1 declare
     type t_employee_type is table of employees.employee_id%type;
3
      t_employees t_employee_type := t_employee_type();
4 begin
5
      <<pre><<pre><<pre><<pre><<pre><</pre>
6
      for i in t_employees.first()..t_employees.last()
7
8
         sys.dbms_output.put_line(t_employees(i)); -- some processing
9
     end loop process_employees;
10 end;
11 /
```

#### **EXAMPLE (BETTER)**

Raise an unitialized collection error if t\_employees is not initialized.

```
1 declare
     type t_employee_type is table of employees.employee_id%type;
2
3
     t_employees t_employee_type := t_employee_type();
5
     <<pre><<pre><<pre><<pre><<pre><</pre>
     for i in 1..t_employees.count()
6
7
8
         sys.dbms_output.put_line(t_employees(i)); -- some processing
9
      end loop process_employees;
10 end;
11 /
```

#### **EXAMPLE (GOOD)**

Raises neither an error nor checking whether the array is empty. t\_employees.count() always returns a number (unless the array is not initialized). If the array is empty count() returns 0 and therefore the loop will not be entered.

```
1 declare
2
      type t_employee_type is table of employees.employee_id%type;
      t_employees t_employee_type := t_employee_type();
3
4 begin
5
     if t_employees is not null then
6
          <<pre><<pre><<pre>cess_employees>>
7
         for i in 1..t_employees.count()
8
9
             sys.dbms_output.put_line(t_employees(i)); -- some processing
10
         end loop process_employees;
11
      end if;
12 end;
13
```



#### **REASON**

When a loose (also called sparse) array is processed using a numeric for loop we have to check with all iterations whether the element exist to avoid a no\_data\_found exception. In addition, the number of iterations is not driven by the number of elements in the array but by the number of the lowest/highest element. The more gaps we have, the more superfluous iterations will be done.

#### **EXAMPLE (BAD)**

```
1 declare -- raises no_data_found when processing 2nd record
      type t_employee_type is table of employees.employee_id%type;
 3
      t_employees t_employee_type;
     co_rogers constant integer := 134;
co_matos constant integer := 143;
co_mcewen constant integer := 158;
 4
 5
 6
 7
      co_index_matos constant integer := 2;
 8 begin
 9
      t_employees := t_employee_type(co_rogers, co_matos, co_mcewen);
10
      t_employees.delete(co_index_matos);
11
12
      if t_employees is not null then
13
          <<pre><<pre><<pre>cess_employees>>
14
          for i in 1..t_employees.count()
15
16
             sys.dbms_output.put_line(t_employees(i));
17
          end loop process_employees;
18
       end if;
19 end:
20
```

```
type t_employee_type is table of employees.employee_id%type;
 3
      t_employees t_employee_type;
     co_rogers constant integer := 134;
co_matos constant integer := 143;
co_mcewen constant integer := 158;
 5
 6
 7
      co_index_matos constant integer := 2;
 8
      l_index
                        pls_integer;
 9 begin
       t_employees := t_employee_type(co_rogers, co_matos, co_mcewen);
10
11
       t_employees.delete(co_index_matos);
12
13
       l_index := t_employees.first();
14
15
       <<pre><<pre><<pre>cess_employees>>
16
        while l_index is not null
17
18
           sys.dbms_output.put_line(t_employees(l_index));
19
          l_index := t_employees.next(l_index);
20
        end loop process_employees;
21
   end;
22
```

## G-4370: Avoid using EXIT to stop loop processing unless you are in a basic loop.



#### **REASON**

A numeric for loop as well as a while loop and a cursor for loop have defined loop boundaries. If you are not able to exit your loop using those loop boundaries, then a basic loop is the right loop to choose.

#### **EXAMPLE (BAD)**

```
1 declare
i integer;
     co_min_value constant simple_integer := 1;
3
     co_max_value constant simple_integer := 10;
4
      co_increment constant simple_integer := 1;
5
6 begin
7
     i := co_min_value;
8
      <<while_loop>>
9
     while (i <= co_max_value)</pre>
10
     loop
     i := i + co_increment;
11
       exit while_loop when i > co_max_value;
12
13
    end loop while_loop;
14
      <<basic_loop>>
15
    loop
16
      exit basic_loop;
17
     end loop basic_loop;
18
19
20
      <<for_loop>>
     for i in co_min_value..co_max_value
21
22
     loop
      null;
23
24
       exit for_loop when i = co_max_value;
25
     end loop for_loop;
26
      <<pre><<pre><<pre><<pre><<pre><>>
27
28
     for r_employee in (select last_name
29
                          from employees)
30
     loop
       sys.dbms_output.put_line(r_employee.last_name);
31
32
       null; -- some processing
33
        exit process_employees;
     end loop process_employees;
34
35 end;
36 /
```

```
1 declare
 2 i integer;
 3
     co_min_value constant simple_integer := 1;
 co_max_value constant simple_integer := 10;
co_increment constant simple_integer := 1;
 6 begin
 7 i := co_min_value;
 8
      <<while_loop>>
9  while (i <= co_max_value)
10  loop
11          i := i + co_increment;
12  end loop while_loop;</pre>
13
14 <<basic_loop>>
15 loop
16 exit basic_loop;
17 end loop basic_loop;
18
24
27
                            from employees)
28 loop
29
      sys.dbms_output.put_line(r_employee.last_name); -- some processing
30
      end loop process_employees;
31 end;
32 /
```

## G-4375: Always use EXIT WHEN instead of an IF statement to exit from a loop.



Minor

Maintainability

#### **REASON**

If you need to use an exit statement use its full semantic to make the code easier to understand and maintain. There is simply no need for an additional if statement.

## **EXAMPLE (BAD)**

```
1 declare
 2
      co_first_year constant pls_integer := 1900;
 3 begin
 4 <<pre><<pre>cess_employees>>
     loop
 5
        my_package.some_processing();
 6
 7
8     if extract(year from sysdate) > co_first_year then
9     exit process_employees;
9
           exit process_employees;
10
        end if;
11
12
          my_package.some_further_processing();
13
       end loop process_employees;
14 end;
15 /
```

```
1 declare
 2
     co_first_year constant pls_integer := 1900;
 3 begin
 4
     <<pre><<pre><<pre><<pre><<pre><>>
 5
     loop
 6
      my_package.some_processing();
 7
8
        exit process_employees when extract(year from sysdate) > co_first_year;
9
10
         my_package.some_further_processing();
11
       end loop process_employees;
12 end;
13 /
```

## G-4380 Try to label your EXIT WHEN statements.



Maintainability

#### **REASON**

It's a good alternative for comments, especially for nested loops to name the loop to exit.

#### **EXAMPLE (BAD)**

```
1 declare
     co_init_loop constant simple_integer
                                                  := 0;
3
     co_increment constant simple_integer
                                                  := 1;
     co_exit_value constant simple_integer := 3;
5
     co_outer_text constant types_up.short_text_type := 'Outer Loop counter is ';
     co_inner_text constant types_up.short_text_type := ' Inner Loop counter is ';
6
7
     l_outerlp pls_integer;
     l_innerlp pls_integer;
8
9 begin
10
     l_outerlp := co_init_loop;
11
     <<outerloop>>
12
     loop
        l_innerlp := co_init_loop;
13
        l_outerlp := nvl(l_outerlp,co_init_loop) + co_increment;
14
15
        <<innerloop>>
16
        loop
17
           l_innerlp := nvl(l_innerlp, co_init_loop) + co_increment;
           sys.dbms_output.put_line(co_outer_text || l_outerlp ||
18
19
                                   co_inner_text || l_innerlp);
20
21
           exit when l_innerlp = co_exit_value;
22
        end loop innerloop;
23
24
        exit when l_innerlp = co_exit_value;
25
      end loop outerloop;
26 end;
27
```

```
1 declare
 co_init_loop constant simple_integer := 0;
co_increment constant simple_integer := 1;
co_exit_value constant simple_integer := 3;
co_outer_text constant types_up.short_text_type := 'Outer Loop counter is ';
co_inner_text constant types_up.short_text_type := 'Inner Loop counter is ';
 7 l_outerlp pls_integer;
8 l_innerlp pls_integer;
 9 begin
10 l_outerlp := co_init_loop;
11 <<outerloop>>
12
        loop
l_innerlp := nvl(l_innerlp, co_init_loop) + co_increment;
sys.dbms_output_line(co_outer_text || l_outerlp ||
17
18
19
                                                     co_inner_text || l_innerlp);
20
21
                 exit outerloop when l_innerlp = co_exit_value;
22
            end loop innerloop;
23 end loop outerloop;
24 end;
25 /
```

G-4385: Never use a cursor for loop to check whether a cursor returns data.



#### **REASON**

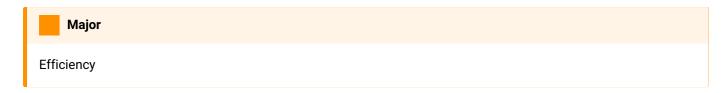
You might process more data than required, which leads to bad performance.

#### **EXAMPLE (BAD)**

```
1 declare
    l_employee_found boolean := false;
3
    cursor c_employees is
    select employee_id, last_name
4
5
         from employees;
6 begin
7
    <<check_employees>>
8
     for r_employee in c_employees
9
    loop
10
     l_employee_found := true;
    end loop check_employees;
11
12 end;
13 /
```

```
1 declare
2 l_employee_found boolean := false;
3
    cursor c_employees is
     select employee_id, last_name
4
5
        from employees;
fremployee c_employees%rowtype;
7 begin
9
    fetch c_employees into r_employee;
10
    1_employee_found := c_employees%found;
11
    close c_employees;
12 end;
13 /
```

#### G-4390: Avoid use of unreferenced FOR loop indexes.



#### **REASON**

If the loop index is used for anything but traffic control inside the loop, this is one of the indicators that a numeric for loop is being used incorrectly. The actual body of executable statements completely ignores the loop index. When that is the case, there is a good chance that you do not need the loop at all.

#### **EXAMPLE (BAD)**

```
1 declare
     1_row
 2
                  pls_integer;
       l_value pls_integer;
 3
      co_lower_bound constant simple_integer
co_upper_bound constant simple_integer
co_row_incr constant simple_integer
 4
 5
 6
    co_value_incr constant simple_integer := 1;

co_value_incr constant simple_integer := 10;

co_delimiter constant types_up.short_text_type := ' ';
 7
8
9
       co_first_value constant simple_integer
10 begin
     1_row := co_lower_bound;
11
       l_value := co_first_value;
12
13
        <<for_loop>>
       for i in co_lower_bound .. co_upper_bound
14
15
       sys.dbms_output.put_line(l_row || co_delimiter || l_value);
16
17
            1_row := 1_row + co_row_incr;
         l_value := l_value + co_value_incr;
18
19
       end loop for_loop;
20
    end;
21 /
```

```
1 declare
2
     co_lower_bound constant simple_integer
3
       co_upper_bound constant simple_integer
                                                            := 5;
       co_value_incr constant simple_integer := 10;
co_delimiter constant types_up.short_text_type := ' ';
4
5
6
       co_first_value constant simple_integer
7 begin
8
     <<for_loop>>
      for i in co_lower_bound .. co_upper_bound
9
10
11
           sys.dbms_output.put_line(i || co_delimiter ||
12
                                     to_char(co_first_value + i * co_value_incr));
13
       end loop for_loop;
14 end;
15
```

## G-4395: Avoid hard-coded upper or lower bound values with FOR loops.



Minor

Changeability, Maintainability

#### **REASON**

Your loop statement uses a hard-coded value for either its upper or lower bounds. This creates a "weak link" in your program because it assumes that this value will never change. A better practice is to create a named constant (or function) and reference this named element instead of the hard-coded value.

## **EXAMPLE (BAD)**

```
1 declare
2
      co_lower_bound constant simple_integer := 1;
3
      co_upper_bound constant simple_integer := 5;
4 begin
5
     <<for_loop>>
     for i in co_lower_bound..co_upper_bound
6
7
         sys.dbms_output.put_line(i);
8
9
     end loop for_loop;
10 end;
11 /
```

# **Exception Handling**

G-5010: Try to use a error/logging framework for your application.



Critical

Reliability, Reusability, Testability

#### Reason

Having a framework to raise/handle/log your errors allows you to easily avoid duplicate application error numbers and having different error messages for the same type of error.

This kind of framework should include

- Logging (different channels like table, mail, file, etc. if needed)
- Error Raising
- Multilanguage support if needed
- Translate ORACLE error messages to a user friendly error text
- Error repository

## Example (bad)

G-5020: Never handle unnamed exceptions using the error number.



#### Reason

When literals are used for error numbers the reader needs the error message manual to unterstand what is going on. Commenting the code or using constants is an option, but it is better to use named exceptions instead, because it ensures a certain level of consistency which makes maintenance easier.

## Example (bad)

```
1 declare
 co_no_data_found constant integer := -1;
 3 begin
4
    my_package.some_processing(); -- some code which raises an exception
 5 exception
 6 when too_many_rows then
7
      my_package.some_further_processing();
    when others then
8
9
     if sqlcode = co_no_data_found then
10
         null;
11 end if;
12 end;
13 /
```

G-5030: Never assign predefined exception names to user defined exceptions.



#### Reason

This is error-prone because your local declaration overrides the global declaration. While it is technically possible to use the same names, it causes confusion for others needing to read and maintain this code. Additionally, you will need to be very careful to use the prefix standard in front of any reference that needs to use Oracle's default exception behavior.

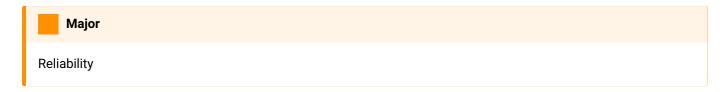
## Example (bad)

Using the code below, we are not able to handle the no\_data\_found exception raised by the select statement as we have overwritten that exception handler. In addition, our exception handler doesn't have an exception number assigned, which should be raised when the select statement does not find any rows.

```
1 declare
   1_dummy dual.dummy%type;
2
3
    no_data_found exception;
     co_rownum constant simple_integer := 0;
4
5
    co_no_data_found constant types_up.short_text_type := 'no_data_found';
6 begin
7
    select dummy
8
      into l_dummy
      from dual
9
     where rownum = co_rownum;
10
11
    if l_dummy is null then
12
13
      raise no_data_found;
14
    end if;
15 exception
    when no_data_found then
16
17
     sys.dbms_output.put_line(co_no_data_found);
18 end;
19 /
20
21 Error report -
22 ORA-01403: no data found
23 ORA-06512: at line 5
25 *Cause: No data was found from the objects.
26 *Action: There was no data from the objects which may be due to end of fetch.
```

```
1 declare
co_empty_value constant types_up.short_text_type := 'empty_value';
co_no_data_found constant types_up.short_text_type := 'no_data_found';
7 begin
8 select dummy
     into l_dummy
from dual
9
10 from dual
11 where rownum = co_rownum;
12
if l_dummy is null then
16 exception
17 when empty_value then
18
      sys.dbms_output.put_line(co_empty_value);
when no_data_found then
sys.dbms_output.put_line(co_no_data_found);
21 end;
22 /
```

G-5040: Avoid use of WHEN OTHERS clause in an exception section without any other specific handlers.



#### Reason

There is not necessarily anything wrong with using when others, but it can cause you to "lose" error information unless your handler code is relatively sophisticated. Generally, you should use when others to grab any and every error only after you have thought about your executable section and decided that you are not able to trap any specific exceptions. If you know, on the other hand, that a certain exception might be raised, include a handler for that error. By declaring two different exception handlers, the code more clearly states what we expect to have happen and how we want to handle the errors. That makes it easier to maintain and enhance. We also avoid hard-coding error numbers in checks against sqlcode.

When using a logging framework like Logger, consider making an exception to this rule and allow a when others even without other specific handlers, but *only* if the when others exception handler calls a logging procedure that saves the error stack (that otherwise is lost) and the last statement of the handler is raise.

## Example (bad)

```
begin
my_package.some_processing();
exception
when others then
my_package.some_further_processing();
end;
//
```

# Example (good)

```
begin
my_package.some_processing();
exception
when dup_val_on_index then
my_package.some_further_processing();
end;
//
```

## Example (exception to the rule)

```
begin
my_package.some_processing();
exception
when others then
logger.log_error('Unhandled Exception');
raise;
end;
// end;
```

G-5050: Avoid use of the RAISE\_APPLICATION\_ERROR built-in procedure with a hard-coded 20nnn error number or hard-coded message.



#### Reason

If you are not very organized in the way you allocate, define and use the error numbers between 20999 and 20000 (those reserved by Oracle for its user community), it is very easy to end up with conflicting usages. You should assign these error numbers to named constants and consolidate all definitions within a single package. When you call raise\_application\_error, you should reference these named elements and error message text stored in a table. Use your own raise procedure in place of explicit calls to raise\_application\_error. If you are raising a "system" exception like no\_data\_found, you must use raise. However, when you want to raise an application-specific error, you use raise\_application\_error. If you use the latter, you then have to provide an error number and message. This leads to unnecessary and damaging hard-coded values. A more fail-safe approach is to provide a predefined raise procedure that automatically checks the error number and determines the correct way to raise the error.

## Example (bad)

```
begin
raise_application_error(-20501,'Invalid employee_id');
end;
//
```

G-5060: Avoid unhandled exceptions.



#### Reason

This may be your intention, but you should review the code to confirm this behavior.

If you are raising an error in a program, then you are clearly predicting a situation in which that error will occur. You should consider including a handler in your code for predictable errors, allowing for a graceful and informative failure. After all, it is much more difficult for an enclosing block to be aware of the various errors you might raise and more importantly, what should be done in response to the error.

The form that this failure takes does not necessarily need to be an exception. When writing functions, you may well decide that in the case of certain exceptions, you will want to return a value such as null, rather than allow an exception to propagate out of the function.

## Example (bad)

```
create or replace package body department_api is
   function name_by_id (in_id in departments.department_id%type)
 2
 3
         return departments.department_name%type is
 4
         1_department_name departments.department_name%type;
 5
     begin
     select department_name
 6
          into l_department_name
 7
 8
          from departments
9
          where department_id = in_id;
10
11
       return l_department_name;
12
     end name_by_id;
13 end department_api;
14 /
```

```
1 create or replace package body department_api is
2
     function name_by_id (in_id in departments.department_id%type)
3
        return departments.department_name%type is
4
        l_department_name departments.department_name%type;
5
    begin
6
      select department_name
7
          into l_department_name
8
          from departments
9
         where department_id = in_id;
10
11
        return l_department_name;
12
     exception
13
       when no_data_found then return null;
14
        when too_many_rows then raise;
15
     end name_by_id;
16 end department_api;
17
```

G-5070: Avoid using Oracle predefined exceptions.



#### Reason

You have raised an exception whose name was defined by Oracle. While it is possible that you have a good reason for "using" one of Oracle's predefined exceptions, you should make sure that you would not be better off declaring your own exception and raising that instead.

If you decide to change the exception you are using, you should apply the same consideration to your own exceptions. Specifically, do not "re-use" exceptions. You should define a separate exception for each error condition, rather than use the same exception for different circumstances.

Being as specific as possible with the errors raised will allow developers to check for, and handle, the different kinds of errors the code might produce.

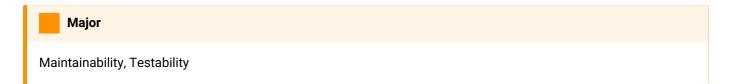
## Example (bad)

```
begin
raise no_data_found;
end;
// /
```

```
1 declare
2  my_exception exception;
3 begin
4  raise my_exception;
5 end;
6 /
```

# Dynamic SQL

G-6010: Always use a character variable to execute dynamic SQL.



#### Reason

Having the executed statement in a variable makes it easier to debug your code (e.g. by logging the statement that failed).

# Example (bad)

```
declare
l_next_val employees.employee_id%type;
co_sql constant types_up.big_string_type :=
'select employees_seq.nextval from dual';
begin
execute immediate co_sql into l_next_val;
end;
// end;
```

G-6020: Try to use output bind arguments in the RETURNING INTO clause of dynamic DML statements rather than the USING clause.



#### Reason

When a dynamic insert, update, or delete statement has a returning clause, output bind arguments can go in the returning into clause or in the using clause.

You should use the returning into clause for values returned from a DML operation. Reserve out and in out bind variables for dynamic PL/SQL blocks that return values in PL/SQL variables.

## Example (bad)

```
create or replace package body employee_api is
     2
3
4
                       , out_new_salary out employees.salary%type)
5
6
       co_sql_stmt constant types_up.big_string_type := '
            update employees set salary = salary + (salary / 100 * :1)
7
8
             where employee_id = :2
9
         returning salary into :3';
10
    begin
      execute immediate co_sql_stmt
11
12
            using in_increase_pct, in_employee_id, out out_new_salary;
13
    end upd_salary;
14 end employee_api;
15
```

```
1 create or replace package body employee_api is
     2
3
4
                       , out_new_salary out employees.salary%type)
5
     is
6
      co_sql_stmt constant types_up.big_string_type :=
7
           'update employees set salary = salary + (salary / 100 * :1)
8
            where employee_id = :2
9
         returning salary into :3';
10
11
      execute immediate co_sql_stmt
12
            using in_increase_pct, in_employee_id
13
            returning into out_new_salary;
14
      end upd_salary;
15 end employee_api;
16 /
```

# Stored Objects

## General

G-7110: Try to use named notation when calling program units.



Changeability, Maintainability

#### REASON

Named notation makes sure that changes to the signature of the called program unit do not affect your call.

This is not needed for standard functions like (to\_char, to\_date, nv1, round, etc.) but should be followed for any other stored object having more than one parameter.

## **EXAMPLE (BAD)**

```
1 declare
2    r_employee employees%rowtype;
3    co_id constant employees.employee_id%type := 107;
4    begin
5    employee_api.employee_by_id(r_employee, co_id);
6    end;
7    /
```

```
1 declare
2    r_employee employees%rowtype;
3    co_id constant employees.employee_id%type := 107;
4    begin
5    employee_api.employee_by_id(out_row => r_employee, in_employee_id => co_id);
6    end;
7    /
```

G-7120 Always add the name of the program unit to its end keyword.



Minor

Maintainability

#### **REASON**

It's a good alternative for comments to indicate the end of program units, especially if they are lengthy or nested.

#### **EXAMPLE (BAD)**

```
1 create or replace package body employee_api is
     function employee_by_id (in_employee_id in employees.employee_id%type)
3
        return employees%rowtype is
        r_employee employees%rowtype;
5
    begin
        select *
6
7
          into r_employee
8
          from employees
9
         where employee_id = in_employee_id;
10
11
        return r_employee;
12
    exception
      when no_data_found then
13
14
          null:
15
       when too_many_rows then
16
           raise;
17
      end:
18 end;
19 /
```

```
1 create or replace package body employee_api is
     function employee_by_id (in_employee_id in employees.employee_id%type)
        return employees%rowtype is
        r_employee employees%rowtype;
5 begin
     select *
6
7
          into r_employee
8
          from employees
9
         where employee_id = in_employee_id;
10
11
        return r_employee;
12
    exception
13
      when no_data_found then
14
          null;
       when too_many_rows then
15
16
           raise;
17
     end employee_by_id;
18 end employee_api;
19
```

G-7130: Always use parameters or pull in definitions rather than referencing external variables in a local program unit.



Major

Maintainability, Reliability, Testability

#### REASON

Local procedures and functions offer an excellent way to avoid code redundancy and make your code more readable (and thus more maintainable). Your local program refers, however, an external data structure, i.e., a variable that is declared outside of the local program. Thus, it is acting as a global variable inside the program.

This external dependency is hidden, and may cause problems in the future. You should instead add a parameter to the parameter list of this program and pass the value through the list. This technique makes your program more reusable and avoids scoping problems, i.e. the program unit is less tied to particular variables in the program. In addition, unit encapsulation makes maintenance a lot easier and cheaper.

#### **EXAMPLE (BAD)**

```
1 create or replace package body employee_api is
    procedure calc_salary (in_employee_id in employees.employee_id%type) is
 3
         r_emp employees%rowtype;
 4
 5
        function commission return number is
 6
           l_commission employees.salary%type := 0;
 7
        begin
 8
           if r_emp.commission_pct is not null
 9
10
               l_commission := r_emp.salary * r_emp.commission_pct;
11
           end if;
12
13
           return l_commission;
14
        end commission;
15
     begin
16
        select *
          into r_emp
17
18
           from employees
19
         where employee_id = in_employee_id;
20
21
         sys.dbms_output.put_line(r_emp.salary + commission());
22
     exception
       when no_data_found then
23
24
           null;
25
        when too_many_rows then
26
           null;
27
      end calc_salary;
28 end employee_api;
29 /
```

```
1 create or replace package body employee_api is
 procedure calc_salary (in_employee_id in employees.employee_id%type) is
 3
       r_emp employees%rowtype;
 4
 5 function commission (in_salary in employees.salary%type
 6
                        ,in_comm_pct in employees.commission_pct%type)
         return number is
7
    l_c
begin
8
          l_commission employees.salary%type := 0;
9
       if in_comm_pct is not null then
10
11
            l_commission := in_salary * in_comm_pct;
12
         end if;
13
14
         return l_commission;
    end commission;
15
16 begin
    select *
17
       into r_emp
from employees
18
19
20
        where employee_id = in_employee_id;
21
23
         r_emp.salary + commission(in_salary => r_emp.salary
24
                             ,in_comm_pct => r_emp.commission_pct)
    );
25
26 exception
27
     when no_data_found then
28
         null;
      when too_many_rows then
29
30
        null;
31 end calc_salary;
32 end employee_api;
33 /
```

#### G-7140: Always ensure that locally defined procedures or functions are referenced.



Maintainability, Reliability

#### **REASON**

This can occur as the result of changes to code over time, but you should make sure that this situation does not reflect a problem. And you should remove the declaration to avoid maintenance errors in the future.

You should go through your programs and remove any part of your code that is no longer used. This is a relatively straightforward process for variables and named constants. Simply execute searches for a variable's name in that variable's scope. If you find that the only place it appears is in its declaration, delete the declaration.

There is never a better time to review all the steps you took, and to understand the reasons you took them, then immediately upon completion of your program. If you wait, you will find it particularly difficult to remember those parts of the program that were needed at one point, but were rendered unnecessary in the end.

#### **EXAMPLE (BAD)**

```
1 create or replace package body my_package is
2
    procedure my_procedure is
     function my_func return number is
3
          co_true constant integer := 1;
4
       begin
5
6
          return co_true;
        end my_func;
7
     begin
8
9
         null;
     end my_procedure;
10
11 end my_package;
12 /
```

```
1 create or replace package body my_package is
    procedure my_procedure is
2
3
       function my_func return number is
4
          co_true constant integer := 1;
       begin
5
6
          return co_true;
        end my_func;
7
     begin
8
9
          sys.dbms_output.put_line(my_func());
     end my_procedure;
10
11 end my_package;
12 /
```



Minor

Efficiency, Maintainability

#### **REASON**

You should go through your programs and remove any parameter that is no longer used.

#### **EXAMPLE (BAD)**

```
1 create or replace package body department_api is
       function name_by_id (in_department_id in departments.department_id%type
 3
                          ,in_manager_id in departments.manager_id%type)
          return departments.department_name%type is
 5
         1_department_name departments.department_name%type;
 6
     begin
 7
         <<find_department>>
 8
        begin
 9
            select department_name
10
              into l_department_name
11
              from departments
12
             where department_id = in_department_id;
13
        exception
14
           when no_data_found or too_many_rows then
15
               1_department_name := null;
16
        end find_department;
17
18
         return l_department_name;
19
      end name_by_id;
20 end department_api;
21
```

```
create or replace package body department_api is
       function name_by_id (in_department_id in departments.department_id%type)
 3
          return departments.department_name%type is
          1_department_name departments.department_name%type;
 5
     begin
         <<find_department>>
 6
 7
        begin
 8
            select department_name
 9
              into l_department_name
10
              from departments
11
             where department_id = in_department_id;
12
        exception
13
            when no_data_found or too_many_rows then
14
               1_department_name := null;
15
         end find_department;
16
17
          return l_department_name;
18
       end name_by_id;
19
   end department_api;
20
```

# G-7160: Always explicitly state parameter mode.



Maintainability

#### **REASON**

By showing the mode of parameters, you help the reader. If you do not specify a parameter mode, the default mode is in . Explicitly showing the mode indication of all parameters is a more assertive action than simply taking the default mode. Anyone reviewing the code later will be more confident that you intended the parameter mode to be in, out or in out .

#### **EXAMPLE (BAD)**

```
1 create or replace package employee_api is
                    2
    procedure upsert (io_id
                                    employees.first_name%type
3
                    ,in_last_name
4
                                         employees.last_name%type
                    ,in_email
5
                                         employees.email%type
                    ,in_email employees.email%type
,in_department_id employees.department_id%type
6
                    , out_success out pls_integer);
7
8 end employee_up;
9
```

#### G-7170: Avoid using an IN OUT parameter as IN or OUT only.



Efficiency, Maintainability

# Unsupported in PL/SQL Cop Validators

Rule G-7170 is not expected to be implemented in the static code analysis validators.

#### **REASON**

Avoid using parameter mode in out unless you actually use the parameter both as input and output. If the code body only reads from the parameter, use in; if the code body only assigns to the parameter, use out. If at the beginning of a project you expect a parameter to be both input and output and therefore choose in out just in case, but later development shows the parameter actually is only in or out, you should change the parameter mode accordingly.

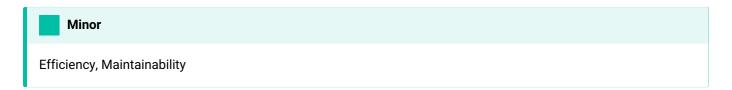
#### **EXAMPLE (BAD)**

```
create or replace package body employee_up is
      2
                       ,io_last_name
,io_email
3
4
                                        in out employees.email%type
                       5
6
7
                       in out employees.job_id%type, io_salary in out employees.salary%type
8
9
                       ,io_commission_pct in out employees.commission_pct%type
10
                       ,io_manager_id in out employees.manager_id%type
                       ,io_department_id in out employees.department_id%type
11
12
                       ,in_wait
                                        in
                                               integer) is
         l_status pls_integer;
13
         co_dflt_pipe_name constant string(30 char) := 'MyPipe';
14
15
         co_ok constant pls_integer := 1;
16
      begin
         -- Receive next message and unpack for each column.
17
18
         1_status := sys.dbms_pipe.receive_message(pipename => co_dflt_pipe_name
19
                                               , timeout => in_wait);
20
         if l_status = co_ok then
21
            sys.dbms_pipe.unpack_message (io_first_name);
22
            sys.dbms_pipe.unpack_message (io_last_name);
23
            sys.dbms_pipe.unpack_message (io_email);
24
            sys.dbms_pipe.unpack_message (io_phone_number);
25
            sys.dbms_pipe.unpack_message (io_hire_date);
26
            sys.dbms_pipe.unpack_message (io_job_id);
27
            sys.dbms_pipe.unpack_message (io_salary);
28
            sys.dbms_pipe.unpack_message (io_commission_pct);
29
            sys.dbms_pipe.unpack_message (io_manager_id);
30
            sys.dbms_pipe.unpack_message (io_department_id);
         end if;
31
32
      end rcv_emp;
33
   end employee_up;
34
```

```
1 create or replace package body employee_up is
      2
3
4
5
                     , \verb"out_phone_number" \verb"out employees.phone_number%type" \\
                     6
7
8
9
                     , out_commission_pct out employees.commission_pct%type
                     , out_manager_id out employees.manager_id%type
10
11
                     ,out_department_id out employees.department_id%type
12
                     ,in_wait in integer) is
13
        l_status pls_integer;
14
       co_dflt_pipe_name constant string(30 char) := 'MyPipe';
15
        co_ok constant pls_integer := 1;
16
    begin
    -- Receive next message and unpack for each column.
17
18
        l_status := sys.dbms_pipe.receive_message(pipename => co_dflt_pipe_name
19
                                            ,timeout => in_wait);
20
       if l_status = co_ok then
         sys.dbms_pipe.unpack_message (out_first_name);
21
          sys.dbms_pipe.unpack_message (out_last_name);
22
23
         sys.dbms_pipe.unpack_message (out_email);
24
         sys.dbms_pipe.unpack_message (out_phone_number);
25
         sys.dbms_pipe.unpack_message (out_hire_date);
26
         sys.dbms_pipe.unpack_message (out_job_id);
27
         sys.dbms_pipe.unpack_message (out_salary);
28
         sys.dbms_pipe.unpack_message (out_commission_pct);
29
         sys.dbms_pipe.unpack_message (out_manager_id);
30
          sys.dbms_pipe.unpack_message (out_department_id);
31
        end if;
32
     end rcv_emp;
33 end employee_up;
34 /
```

# **Packages**

G-7210: Try to keep your packages small. Include only few procedures and functions that are used in the same context.



#### **REASON**

The entire package is loaded into memory when the package is called the first time. To optimize memory consumption and keep load time small packages should be kept small but include components that are used together.

G-7220: Always use forward declaration for private functions and procedures.



Minor

Changeability

#### **REASON**

Having forward declarations allows you to order the functions and procedures of the package in a reasonable way.

#### **EXAMPLE (BAD)**

```
1 create or replace package department_api is
     procedure del (in_department_id in departments.department_id%type);
 3 end department_api;
 4 /
 5
 6 create or replace package body department_api is
7
     function does_exist (in_department_id in departments.department_id%type)
         return boolean is
8
 9
         l_return pls_integer;
10
     begin
11
         <<check_row_exists>>
12
        begin
13
           select 1
14
             into l_return
15
              from departments
16
             where department_id = in_department_id;
17
        exception
18
           when no_data_found or too_many_rows then
19
              l_return := 0;
20
        end check_row_exists;
21
         return l_return = 1;
22
23
     end does_exist;
24
25
       procedure del (in_department_id in departments.department_id%type) is
26
27
        if does_exist(in_department_id) then
28
          null;
29
        end if;
30
     end del;
31 end department_api;
32
```

```
1 create or replace package department_api is
 procedure del (in_department_id in departments.department_id%type);
3 end department_api;
4 /
 5
 6 create or replace package body department_api is
7 function does_exist (in_department_id in departments.department_id%type)
8
       return boolean;
9
10 procedure del (in_department_id in departments.department_id%type) is
11
     begin
12
      if does_exist(in_department_id) then
13
         null;
      end if;
14
15 end del;
16
     function does_exist (in_department_id in departments.department_id%type)
17
     return boolean is
18
19
        l_return pls_integer;
20 begin
21
     <<check_row_exists>>
22
       begin
23
        select 1
24
            into l_return
25
             from departments
26
            where department_id = in_department_id;
      exception
27
        when no_data_found or too_many_rows then
28
29
             l_return := 0;
      end check_row_exists;
30
31
32
        return l_return = 1;
    end does_exist;
33
34 end department_api;
35 /
```

#### G-7230: Avoid declaring global variables public.



#### **REASON**

You should always declare package-level data (non-constants) inside the package body. You can then define "get and set" methods (functions and procedures, respectively) in the package specification to provide controlled access to that data. By doing so you can guarantee data integrity, you can change your data structure implementation, and also track access to those data structures.

Data structures (scalar variables, collections, cursors) declared in the package specification (not within any specific program) can be referenced directly by any program running in a session with execute rights to the package.

Instead, declare all package-level data in the package body and provide "get and set" methods - a function to get the value and a procedure to set the value - in the package specification. Developers then can access the data using these methods - and will automatically follow all rules you set upon data modification.

For package-level constants, consider whether the constant should be public and usable from other code, or if only relevant for code within the package. If the latter, declare the constant in the package body. If the former, it is typically good practice to place the constants in a package specification that only holds constants.

# **EXAMPLE (BAD)**

```
create or replace package employee_api as
 1
 2
     co_min_increase constant types_up.sal_increase_type := 0.01;
 3
     co_max_increase constant types_up.sal_increase_type := 0.5;
 4
      g_salary_increase types_up.sal_increase_type := co_min_increase;
 5
 6
       procedure set_salary_increase (in_increase in types_up.sal_increase_type);
 7
       function salary_increase return types_up.sal_increase_type;
 8 end employee_api;
9
10
11 create or replace package body employee_api as
12
    procedure set_salary_increase (in_increase in types_up.sal_increase_type) is
13
     beain
14
        g_salary_increase := greatest(least(in_increase,co_max_increase)
15
                                      , co_min_increase);
      end set_salary_increase;
16
17
18
       function salary_increase return types_up.sal_increase_type is
19
20
        return g_salary_increase;
21
       end salary_increase;
22 end employee_api;
23
```

```
1 create or replace package constants_up as
2 co_min_increase constant types_up.sal_increase_type := 0.01;
3
      co_max_increase constant types_up.sal_increase_type := 0.5;
4 end constants_up;
5 /
6
7
   create or replace package employee_api as
procedure set_salary_increase (in_increase in types_up.sal_increase_type);
9
      function salary_increase return types_up.sal_increase_type;
10 end employee_api;
11 /
12
13 create or replace package body employee_api as
14
      g_salary_increase types_up.sal_increase_type(4,2);
15
16
     procedure init;
17
18
     procedure set_salary_increase (in_increase in types_up.sal_increase_type) is
19
      begin
20
       g_salary_increase := greatest(least(in_increase
21
                                          , constants_up.co_max_increase)
22
                                     , constants_up.co_min_increase);
23
     end set_salary_increase;
24
25
     function salary_increase return types_up.sal_increase_type is
26
      begin
27
       return g_salary_increase;
28
     end salary_increase;
29
30
     procedure init
31
     is
32
     begin
33
       g_salary_increase := constants_up.co_min_increase;
34
      end init;
35 begin
36 init();
37 end employee_api;
38 /
```



Minor

Maintainability

#### **REASON**

The purpose of the initialization block of a package body is to set initial values of the global variables of the package (initialize the package state). Although return is syntactically allowed in this block, it makes no sense. If it is the last keyword of the block, it is superfluous. If it is not the last keyword, then all code after the return is unreachable and thus dead code.

#### **EXAMPLE (BAD)**

```
1 create or replace package body employee_api as
       g_salary_increase types_up.sal_increase_type(4,2);
 3
 4
       procedure set_salary_increase (in_increase in types_up.sal_increase_type) is
 5
       begin
 6
          g_salary_increase := greatest(least(in_increase
 7
                                             , constants_up.max_salary_increase())
 8
                                       , constants_up.min_salary_increase());
 9
      end set_salary_increase;
10
       function salary_increase return types_up.sal_increase_type is
11
12
       begin
13
         return g_salary_increase;
14
       end salary_increase;
15
16 begin
17
       g_salary_increase := constants_up.min_salary_increase();
18
19
       return:
20
       set_salary_increase(constants_up.min_salary_increase()); -- dead code
21
22
   end employee_api;
23
```

```
create or replace package body employee_api as
       g_salary_increase types_up.sal_increase_type(4,2);
 3
 4
       procedure set_salary_increase (in_increase in types_up.sal_increase_type) is
 5
      begin
 6
          g_salary_increase := greatest(least(in_increase
 7
                                             , constants_up.max_salary_increase())
 8
                                       , constants_up.min_salary_increase());
 9
       end set_salary_increase;
10
11
       function salary_increase return types_up.sal_increase_type is
12
       begin
13
         return g_salary_increase;
14
      end salary_increase;
15
16 begin
17
       g_salary_increase := constants_up.min_salary_increase();
18 end employee_api;
19
```

# **Procedures**

G-7310: Avoid standalone procedures – put your procedures in packages.



#### **REASON**

Use packages to structure your code, combine procedures and functions which belong together.

Package bodies may be changed and compiled without invalidating other packages. This is major advantage compared to standalone procedures and functions.

#### **EXAMPLE (BAD)**

```
1 create or replace procedure my_procedure is
2 begin
3 null;
4 end my_procedure;
5 /
```

```
1  create or replace package my_package is
2    procedure my_procedure;
3  end my_package;
4  /
5
6  create or replace package body my_package is
7    procedure my_procedure is
8    begin
9    null;
10    end my_procedure;
11  end my_package;
12  /
```

#### G-7320: Avoid using RETURN statements in a PROCEDURE.



Maintainability, Testability

#### **REASON**

Use of the return statement is legal within a procedure in PL/SQL, but it is very similar to a goto, which means you end up with poorly structured code that is hard to debug and maintain.

A good general rule to follow as you write your PL/SQL programs is "one way in and one way out". In other words, there should be just one way to enter or call a program, and there should be one way out, one exit path from a program (or loop) on successful termination. By following this rule, you end up with code that is much easier to trace, debug, and maintain.

### **EXAMPLE (BAD)**

```
1 create or replace package body my_package is
 2
     procedure my_procedure is
 3
        l_idx simple_integer := 1;
 4
        co_modulo constant simple_integer := 7;
     begin
 5
 6
        <<mod7_loop>>
 7
        loop
8
         if mod(l_idx, co_modulo) = 0 then
9
            return;
10
          end if;
11
          l_{idx} := l_{idx} + 1;
12
13
        end loop mod7_loop;
14
     end my_procedure;
15 end my_package;
16 /
```

```
1 create or replace package body my_package is
2
     procedure my_procedure is
3
        l_idx simple_integer := 1;
4
        co_modulo constant simple_integer := 7;
5
     begin
      <<mod7_loop>>
6
7
        loop
8
         exit mod7_loop when mod(l_idx,co_modulo) = 0;
9
10
          l_idx := l_idx + 1;
        end loop mod7_loop;
11
     end my_procedure;
12
13 end my_package;
14 /
```

# G-7330: Always assign values to OUT parameters.



Major

Maintainability, Testability

#### **REASON**

Marking a parameter for output means that callers will expect its value to be updated with a result from the execution of the procedure. Failing to update the parameter before the procedure returns is surely an error.

# **EXAMPLE (BAD)**

```
1 create or replace package body my_package is
procedure greet(
     in_name in varchar2
3
       , out_greeting out varchar2
4
4 , ou 5 ) is
6
        1_message varchar2(100 char);
7
     begin
    l_message := 'Hello, ' || in_name;
end my_procedure;
8
9
10 end my_package;
11 /
```

# **Functions**

G-7410: Avoid standalone functions – put your functions in packages.



#### **REASON**

Use packages to structure your code, combine procedures and functions which belong together.

Package bodies may be changed and compiled without invalidating other packages. This is major advantage compared to standalone procedures and functions.

# **EXAMPLE (BAD)**

```
1 create or replace function my_function return varchar2 is
2 begin
3 return null;
4 end my_function;
5 /
```

```
create or replace package body my_package is
function my_function return varchar2 is
begin
return null;
end my_function;
end my_package;
//
```

# G-7420: Always make the RETURN statement the last statement of your function.



#### **REASON**

The reader expects the return statement to be the last statement of a function.

#### **EXAMPLE (BAD)**

```
1 create or replace package body my_package is
     function my_function (in_from in pls_integer
                        , in_to in pls_integer) return pls_integer is
 3
 4
         l_ret pls_integer;
    begin
 5
        1_ret := in_from;
 6
7
        <<for_loop>>
       for i in in_from .. in_to
8
9
       loop
        l_ret := l_ret + i;
10
          if i = in_to then
11
             return l_ret;
12
13
          end if;
        end loop for_loop;
14
15
     end my_function;
16 end my_package;
17 /
```

```
1 create or replace package body my_package is
     function my_function (in_from in pls_integer
3
                         , in_to in pls_integer) return pls_integer is
4
         l_ret pls_integer;
5 begin
     l_ret := in_from;
<<for_loop>>
6
7
      for i in in_from .. in_to loop
8
9
10
         l_ret := l_ret + i;
       end loop for_loop;
11
12
        return l_ret;
13
     end my_function;
14 end my_package;
15
```

G-7430: Try to use no more than one RETURN statement within a function.



Major

Will have a medium/potential impact on the maintenance cost. Maintainability, Testability

#### **REASON**

A function should have a single point of entry as well as a single exit-point.

#### **EXAMPLE (BAD)**

```
1 create or replace package body my_package is
     function my_function (in_value in pls_integer) return boolean is
        co_yes constant pls_integer := 1;
3
4 begin
     if in_value = co_yes then
5
6
          return true;
      else
7
           return false;
8
       end if;
10 end my_function;
11 end my_package;
12 /
```

#### **EXAMPLE (BETTER)**

```
1 create or replace package body my_package is
    function my_function (in_value in pls_integer) return boolean is
3
       co_yes constant pls_integer := 1;
4
        l_ret boolean;
5 begin
     if in_value = co_yes then
6
7
          1_ret := true;
8 else
9
        l_ret := false;
      end if;
10
11
12
       return l_ret;
    end my_function;
13
14 end my_package;
15 /
```

```
create or replace package body my_package is
function my_function (in_value in pls_integer) return boolean is
co_yes constant pls_integer := 1;
begin
return in_value = co_yes;
end my_function;
end my_package;
///
```

# G-7440: Never use OUT parameters to return values from a function.



#### **REASON**

A function should return all its data through the return clause. Having an out parameter prohibits usage of a function within SQL statements.

# **EXAMPLE (BAD)**

```
create or replace package body my_package is
function my_function (out_date out date) return boolean is
begin
out_date := sysdate;
return true;
end my_function;
end my_package;
//
```

```
create or replace package body my_package is
function my_function return date is
begin
return sysdate;
end my_function;
end my_package;
//
```

#### G-7450: Never return a NULL value from a BOOLEAN function.



Reliability, Testability

#### **REASON**

If a boolean function returns null, the caller has do deal with it. This makes the usage cumbersome and more errorprone.

# **EXAMPLE (BAD)**

```
create or replace package body my_package is
function my_function return boolean is
begin
return null;
end my_function;
end my_package;
//
```

```
create or replace package body my_package is
function my_function return boolean is
begin
return true;
end my_function;
end my_package;
///
```

G-7460: Try to define your packaged/standalone function deterministic if appropriate.



#### **REASON**

A deterministic function (always return same result for identical parameters) which is defined to be deterministic will be executed once per different parameter within a SQL statement whereas if the function is not defined to be deterministic it is executed once per result row.

# **EXAMPLE (BAD)**

```
create or replace package department_api is
function name_by_id (in_department_id in departments.department_id%type)
return departments.department_name%type;
end department_api;
///
```

```
create or replace package department_api is
function name_by_id (in_department_id in departments.department_id%type)
return departments.department_name%type deterministic;
end department_api;
///
```

# **Oracle Supplied Packages**

G-7510: Always prefix ORACLE supplied packages with owner schema name.



#### **REASON**

The signature of oracle-supplied packages is well known and therefore it is quite easy to provide packages with the same name as those from oracle doing something completely different without you noticing it.

# **EXAMPLE (BAD)**

```
1 declare
2   co_hello_world constant string(30 char) := 'Hello World';
3 begin
4   dbms_output.put_line(co_hello_world);
5 end;
6 /
```

```
1 declare
2   co_hello_world constant string(30 char) := 'Hello World';
3 begin
4   sys.dbms_output.put_line(co_hello_world);
5 end;
6 /
```

# **Object Types**

There are no object type-specific recommendations to be defined at the time of writing.

# **Triggers**

# G-7710: Avoid cascading triggers.



#### **REASON**

Having triggers that act on other tables in a way that causes triggers on that table to fire lead to obscure behavior.

### **EXAMPLE (BAD)**

```
1 create or replace trigger dept_br_u
 2 before update on departments for each row
 3 begin
 4
       insert into departments_hist (department_id
 5
                                    , department_name
 6
                                    ,manager_id
                                   ,location_id
 7
8
                                    , modification_date)
 9
           values (:old.department_id
10
                  ,:old.department_name
                   ,:old.manager_id
11
12
                   ,:old.location_id
13
                   , sysdate);
14 end;
15 /
16 create or replace trigger dept_hist_br_i
17 before insert on departments_hist for each row
18 begin
19
      insert into departments_log (department_id
20
                                  ,department_name
21
                                  , modification_date)
                            values (:new.department_id
22
                                  ,:new.department_name
23
24
                                   , sysdate);
25 end;
26
```

```
1 create or replace trigger dept_br_u
 2 before update on departments for each row
 3 begin
 4
      insert into departments_hist (department_id
 5
                                   ,department_name
 6
                                   ,manager_id
 7
                                   ,location_id
8
                                   , modification_date)
9
           values (:old.department_id
10
               ,:old.department_name
                  ,:old.manager_id
11
                   ,:old.location_id
12
13
                   , sysdate);
14
       insert into departments_log (department_id
15
16
                                  , department_name
                                  , modification_date)
17
18
                           values (:old.department_id
19
                                  ,:old.department_name
20
                                  , sysdate);
21
22 end;
23 /
```

#### G-7720: Never use multiple UPDATE OF in trigger event clause.



Maintainability, Reliability, Testability

#### **REASON**

A DML trigger can have multiple triggering events separated by or like before insert or delete or update of some\_column. If you have multiple update of separated by or, only one of them (the last one) is actually used and you get no error message, so you have a bug waiting to happen. Instead you always should use a single update of with all columns comma-separated, or an update without of if you wish all columns.

#### **EXAMPLE (BAD)**

```
1 create or replace trigger dept_br_u
2 before update of department_id or update of department_name
3 on departments for each row
4 begin
      -- will only fire on updates of department_name
5
6
     insert into departments_log (department_id
7
                                  , department_name
8
                                  , modification_date)
9
                           values (:old.department_id
                                  ,:old.department_name
10
11
                                  , sysdate);
12 end;
13
```

```
1 create or replace trigger dept_br_u
2 before update of department_id, department_name
3 on departments for each row
4 begin
5
       insert into departments_log (department_id
                                  , department_name
6
7
                                  , modification_date)
8
                           values (:old.department_id
9
                                  ,:old.department_name
10
                                  , sysdate);
11
   end;
12
```

G-7730: Avoid multiple DML events per trigger if primary key is assigned in trigger.



#### **REASON**

If a trigger makes assignment to the primary key anywhere in the trigger code, that causes the session firing the trigger to take a lock on any child tables with a foreign key to this primary key. Even if the assignment is in for example an if inserting block and the trigger is fired by an update statement, such locks still happen unnecessarily. The issue is avoided by having one trigger for the insert containing the primary key assignment, and another trigger for the update. Or even better by handling the insert assignment as 'default on null' clauses, so that only an on update trigger is needed.

# **EXAMPLE (BAD)**

```
1 create or replace trigger dept_br_iu
2 before insert or update
3 on departments for each row
4 begin
    if inserting then
5
6
         :new.department_id := department_seq.nextval;
7
         :new.created_date := sysdate;
    end if;
8
9
     if updating then
10
      :new.changed_date := sysdate;
    end if;
11
12 end;
13 /
```

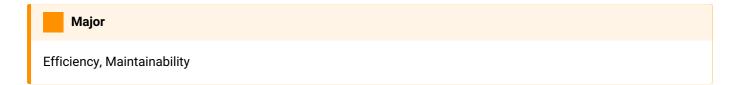
# **EXAMPLE (BETTER)**

```
1 create or replace trigger dept_br_i
2 before insert
3 on departments for each row
4 begin
5
      :new.department_id := department_seq.nextval;
      :new.created_date := sysdate;
6
7
   end;
8
9
10 create or replace trigger dept_br_u
11 before update
12 on departments for each row
13 begin
14
   :new.changed_date := sysdate;
15
   end;
```

```
1 alter table department modify department_id default on null department_seq.nextval;
2 alter table department modify created_date default on null sysdate;
3
4 create or replace trigger dept_br_u
5 before update
6 on departments for each row
7 begin
8 :new.changed_date := sysdate;
9 end;
10 /
```

# Sequences

G-7810: Never use SQL inside PL/SQL to read sequence numbers (or SYSDATE).



#### **REASON**

Since ORACLE 11g it is no longer needed to use a select statement to read a sequence (which would imply a context switch).

# **EXAMPLE (BAD)**

```
1 declare
2    l_sequence_number employees.emloyee_id%type;
3 begin
4    select employees_seq.nextval
5     into l_sequence_number
6     from dual;
7 end;
8 /
```

# **Patterns**

# Checking the Number of Rows

G-8110: Never use SELECT COUNT(\*) if you are only interested in the existence of a row.



#### REASON

If you do a select count(\*) all rows will be read according to the where clause, even if only the availability of data is of interest. For this we have a big performance overhead. If we do a select count(\*) ... where rownum = 1 there is also a overhead as there will be two communications between the PL/SQL and the SQL engine. See the following example for a better solution.

# **EXAMPLE (BAD)**

```
1 declare
 2
   l_count pls_integer;
 3
     co_zero constant simple_integer := 0;
 4
      co_salary constant employees.salary%type := 5000;
 5 begin
 6 select count(*)
 7
       into l_count
8
       from employees
9
      where salary < co_salary;</pre>
      if l_count > co_zero then
10
11
        <<emp_loop>>
12
         for r_emp in (select employee_id
13
                        from employees)
         loop
14
15
           if r_emp.salary < co_salary then</pre>
16
              my_package.my_proc(in_employee_id => r_emp.employee_id);
             end if;
17
18
         end loop emp_loop;
19
     end if;
20 end;
21 /
```

```
1 declare
 2
     co_salary constant employees.salary%type := 5000;
 3 begin
 4
     <<emp_loop>>
 5
      for r_emp in (select e1.employee_id
 6
                       from employees e1
 7
                      where exists(select e2.salary
8
                                    from employees e2
9
                                    where e2.salary < co_salary))</pre>
10
       loop
11
          my_package.my_proc(in_employee_id => r_emp.employee_id);
12
       end loop emp_loop;
13 end;
14
```

G-8120: Never check existence of a row to decide whether to create it or not.



#### **REASON**

The result of an existence check is a snapshot of the current situation. You never know whether in the time between the check and the (insert) action someone else has decided to create a row with the values you checked. Therefore, you should only rely on constraints when it comes to prevention of duplicate records.

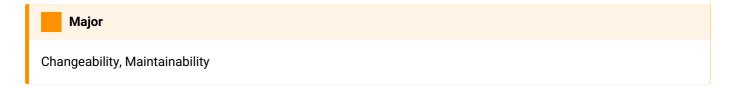
# **EXAMPLE (BAD)**

```
1 create or replace package body department_api is
 2
       procedure ins (in_r_department in departments%rowtype) is
 3
          l_count pls_integer;
     begin
 4
     select count(*)
 5
          into l_count
from departments
 6
 7
8
         where department_id = in_r_department.department_id;
9
     if l_{count} = 0 then
10
11
             insert into departments
12
                  values in_r_department;
13
          end if;
   end ins;
14
15 end department_api;
```

```
1 create or replace package body department_api is
    procedure ins (in_r_department in departments%rowtype) is
2
     begin
3
     insert into departments
4
5
             values in_r_department;
6
    exception
7
       when dup_val_on_index then null; -- handle exception
     end ins;
8
9 end department_api;
10
```

# Access objects of foreign application schemas

G-8210: Always use synonyms when accessing objects of another application schema.



#### **REASON**

If a connection is needed to a table that is placed in a foreign schema, using synonyms is a good choice. If there are structural changes to that table (e.g. the table name changes or the table changes into another schema) only the synonym has to be changed no changes to the package are needed (single point of change). If you only have read access for a table inside another schema, or there is another reason that does not allow you to change data in this table, you can switch the synonym to a table in your own schema. This is also good practice for testers working on test systems.

# **EXAMPLE (BAD)**

```
1 declare
2
     1_product_name oe.products.product_name%type;
3
      co_price constant oe.products@list_price%type := 1000;
4 begin
5
     select p.product_name
6
       into l_product_name
       from oe.products p
7
      where list_price > co_price;
8
9 exception
    when no_data_found then
10
11
       null; -- handle_no_data_found;
12
     when too_many_rows then
13
       null; -- handle_too_many_rows;
14 end;
15
```

```
1 create synonym oe_products for oe.products;
3 declare
     1_product_name oe_products.product_name%type;
4
5
      co_price constant oe_products.list_price%type := 1000;
6 begin
    select p.product_name
7
8
       into l_product_name
       from oe_products p
9
      where list_price > co_price;
10
11 exception
    when no_data_found then
12
13
       null; -- handle_no_data_found;
14
     when too_many_rows then
15
        null; -- handle_too_many_rows;
16 end;
17 /
```

# Validating input parameter size

G-8310: Always validate input parameter size by assigning the parameter to a size limited variable in the declaration section of program unit.



Minor

Maintainability, Reliability, Reusability, Testability

#### REASON

This technique raises an error (value\_error) which may not be handled in the called program unit. This is the right way to do it, as the error is not within this unit but when calling it, so the caller should handle the error.

#### **EXAMPLE (BAD)**

```
create or replace package body department_api is
     function dept_by_name (in_dept_name in departments.department_name%type)
2
3
         return departments%rowtype is
4
         1_return departments%rowtype;
     begin
5
      if
6
              in_dept_name is null
7
            or length(in_dept_name) > 20
         then
8
9
             raise err.e_param_to_large;
10
         end if;
11
          -- get the department by name
         select *
12
13
           from departments
14
          where department_name = in_dept_name;
15
16
          return l_return;
     end dept_by_name;
17
18 end department_api;
19
```

# **EXAMPLE (GOOD)**

```
1 create or replace package body department_api is
2
     function dept_by_name (in_dept_name in departments.department_name%type)
3
         return departments%rowtype is
         l_dept_name departments.department_name%type not null := in_dept_name;
4
5
         1_return departments%rowtype;
     begin
6
7
          -- get the department by name
         select *
8
9
           from departments
10
           where department_name = 1_dept_name;
11
12
          return l_return;
13
      end dept_by_name;
14 end department_api;
15
```

### **FUNCTION CALL**

```
1 ...
2    r_department := department_api.dept_by_name('Far to long name of a department');
3    ...
4    exception
5    when value_error then ...
```

#### Ensure single execution at a time of a program unit

G-8410: Always use application locks to ensure a program unit is only running once at a given time.



#### **REASON**

This technique allows us to have locks across transactions as well as a proven way to clean up at the end of the session.

The alternative using a table where a "Lock-Row" is stored has the disadvantage that in case of an error a proper cleanup has to be done to "unlock" the program unit.

#### **EXAMPLE (BAD)**

```
1 -- Bad
 2 /* Example */
 3 create or replace package body lock_up is
 4 -- manage locks in a dedicated table created as follows:
      -- CREATE TABLE app_locks (
 5
           lock_name VARCHAR2(128 CHAR) NOT NULL primary key
 6
 7
      -- );
8
9
     procedure request_lock (in_lock_name in varchar2) is
10
     begin
      -- raises dup_val_on_index
11
        insert into app_locks (lock_name) values (in_lock_name);
12
13
     end request_lock;
14
     procedure release_lock(in_lock_name in varchar2) is
15
16
17
      delete from app_locks where lock_name = in_lock_name;
18
     end release_lock;
19 end lock_up;
20 /
21
22 /* Call bad example */
23 declare
   co_lock_name constant varchar2(30 char) := 'APPLICATION_LOCK';
24
25 begin
26
     lock_up.request_lock(in_lock_name => co_lock_name);
27
      -- processing
28
      lock_up.release_lock(in_lock_name => co_lock_name);
29 exception
30 when others then
        -- log error
        lock_up.release_lock(in_lock_name => co_lock_name);
34 end;
```

#### **EXAMPLE (GOOD)**

```
1 /* Example */
 2 create or replace package body lock_up is
3 function request_lock(
4
       in_lock_name
                            in varchar2,
 5
         in_release_on_commit in boolean := false)
6
     return varchar2 is
7
        l_lock_handle varchar2(128 char);
8
     begin
9
     sys.dbms_lock.allocate_unique(
10
          lockname => in_lock_name,
11
            lockhandle
                         => l_lock_handle,
12
           expiration_secs => constants_up.co_one_week
13
14
         if sys.dbms_lock.request(
15
              lockhandle => l_lock_handle,
              lockmode
                              => sys.dbms_lock.x_mode,
16
              timeout
                              => sys.dbms_lock.maxwait,
17
              release_on_commit => coalesce(in_release_on_commit, false)
18
19
           ) > 0
20
        then
21
         raise err.e_lock_request_failed;
22
       end if;
23
        return l_lock_handle;
24
     end request_lock;
25
26
     procedure release_lock(in_lock_handle in varchar2) is
27
     begin
28
       if sys.dbms_lock.release(lockhandle => in_lock_handle) > 0 then
29
           raise err.e_lock_request_failed;
30
         end if;
31
      end release_lock;
32 end lock_up;
33 /
34
35 /* Call good example */
36 declare
37
     l_handle varchar2(128 char);
38
      co_lock_name constant varchar2(30 char) := 'APPLICATION_LOCK';
39 begin
40
     l_handle := lock_up.request_lock(in_lock_name => co_lock_name);
41
       -- processing
42
      lock_up.release_lock(in_lock_handle => l_handle);
43 exception
44
   when others then
45
         -- log error
46
         lock_up.release_lock(in_lock_handle => l_handle);
47
48 end;
49 /
```

Use dbms\_application\_info package to follow progress of a process

G-8510: Always use dbms\_application\_info to track program process transiently.



#### **REASON**

This technique allows us to view progress of a process without having to persistently write log data in either a table or a file. The information is accessible through the v\$session view.

#### **EXAMPLE (BAD)**

```
create or replace package body employee_api is
procedure process_emps is
begin

cemployees>>
for emp_rec in (select employee_id
from employees
for employee
for e
```

#### **EXAMPLE (GOOD)**

```
1 create or replace package body employee_api is
   procedure process_emps is
 2
 3
      begin
      sys.dbms_application_info.set_module(module_name => $$plsql_unit
 4
                                          ,action_name => 'Init');
 5
      <<employees>>
 6
7
        for emp_rec in (select employee_id
8
                         from employees
9
                         order by employee_id)
10
         loop
11
         sys.dbms_application_info.set_action('Processing ' || emp_rec.employee_id);
12
        end loop employees;
13
    end process_emps;
14 end employee_api;
15 /
```

# Complexity Analysis

Using software metrics like complexity analysis will guide you towards maintainable and testable pieces of code by reducing the complexity and splitting the code into smaller chunks.

#### Halstead Metrics

#### Calculation

First, we need to compute the following numbers, given the program:

- $n_1$  = the number of distinct operators
- $n_2$  = the number of distinct operands
- $N_1$  = the total number of operators
- $N_2$  = the total number of operands

From these numbers, five measures can be calculated:

• Program length:

$$N = N_1 + N_2$$

• Program vocabulary:

$$n=n_1+n_2$$

• Volume:

$$V = N \cdot log_2 n$$

• Difficulty:

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$$

• Effort:

$$E = D \cdot V$$

The difficulty measure

D is related to the difficulty of the program to write or understand, e.g. when doing code review.

The volume measure

 ${\it V}$  describes the size of the implementation of an algorithm.

## McCabe's Cyclomatic Complexity

#### Description

Cyclomatic complexity (or conditional complexity) is a software metric used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code.

Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. For instance, if the source code contains no decision points, such as <code>if</code> statements or <code>for</code> loops, the complexity would be 1, since there is only a single path through the code. If the code has a single <code>if</code> statement containing a single condition there would be two paths through the code, one path where the <code>if</code> statement is evaluated as <code>true</code> and one path where the <code>if</code> statement is evaluated as <code>false</code>.

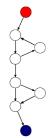
#### Calculation

Mathematically, the cyclomatic complexity of a structured program is defined with reference to a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second (the control flow graph of the program). The complexity is then defined as:

$$M = E - N + 2P$$

where

- M = cyclomatic complexity
- E = the number of edges of the graph
- N = the number of nodes of the graph
- P = the number of connected components.



Take, for example, a control flow graph of a simple program. The program begins executing at the red node, then enters a loop (group of three nodes immediately below the red node). On exiting the loop, there is a conditional statement (group below the loop), and finally the program exits at the blue node. For this graph,

E=9.

N=8 and

P=1, so the cyclomatic complexity of the program is

3.

```
begin
2
      for i in 1...3
3
4
        dbms_output.put_line('in loop');
      end loop;
5
6
7
       if 1 = 1
8
       then
9
        dbms_output.put_line('yes');
10
      end if:
11
12
       dbms_output.put_line('end');
13
   end;
```

For a single program (or subroutine or method), P is always equal to 1. Cyclomatic complexity may, however, be applied to several such programs or subprograms at the same time (e.g., to all of the methods in a class), and in these cases P will be equal to the number of programs in question, as each subprogram will appear as a disconnected subset of the graph.

It can be shown that the cyclomatic complexity of any structured program with only one entrance point and one exit point is equal to the number of decision points (i.e., if statements or conditional loops) contained in that program plus

one.

Cyclomatic complexity may be extended to a program with multiple exit points; in this case it is equal to:

$$\pi = s + 2$$

#### Where

- $\bullet$   $\ \pi$  is the number of decision points in the program, and
- ullet s is the number of exit points.

## Code Reviews

Code reviews check the results of software engineering. According to IEEE-Norm 729, a review is a more or less planned and structured analysis and evaluation process. Here we distinguish between code review and architect review.

To perform a code review means that after or during the development one or more reviewer proof-reads the code to find potential errors, potential areas for simplification, or test cases. A code review is a very good opportunity to save costs by fixing issues before the testing phase.

What can a code-review be good for?

- · Code quality
- · Code clarity and maintainability
- · Quality of the overall architecture
- · Quality of the documentation
- · Quality of the interface specification

For an effective review, the following factors must be considered:

- Definition of clear goals.
- Choice of a suitable person with constructive critical faculties.
- · Psychological aspects.
- · Selection of the right review techniques.
- Support of the review process from the management.
- Existence of a culture of learning and process optimization.

#### Requirements for the reviewer:

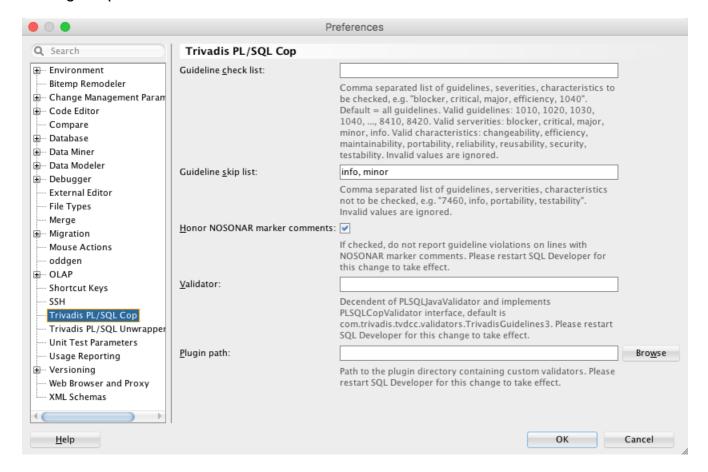
- He must not be the owner of the code.
- Code reviews may be unpleasant for the developer, as he could fear that his code will be criticized. If the critic is not considerate, the code writer will build up rejection and resistance against code reviews.

# **Tool Support**

# Development

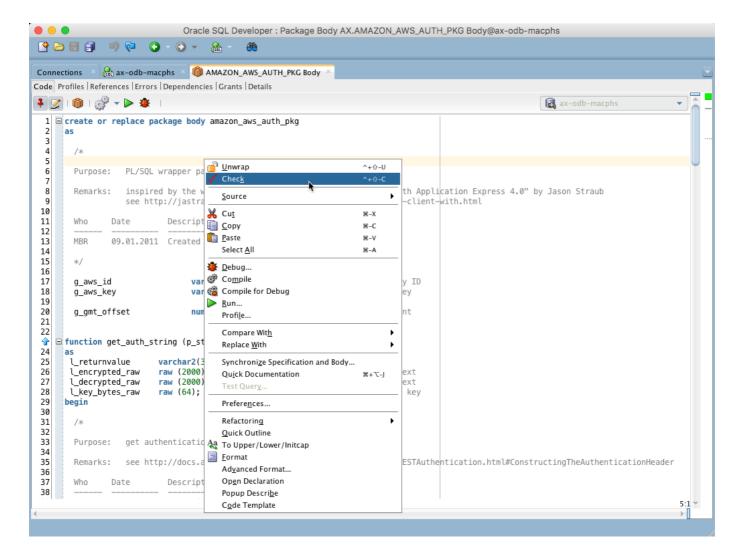
Trivadis offers a cost-free extension to ORACLE SQL Developer to test compliance with this coding guideline. The extension may be parameterized to your preferred set of rules and allows checking this set against a program unit.

#### Setting the preferences



There is an include list as well as an exclude list to define which rules to be checked or ignored.

#### Activate PLSQL Cop using context menu

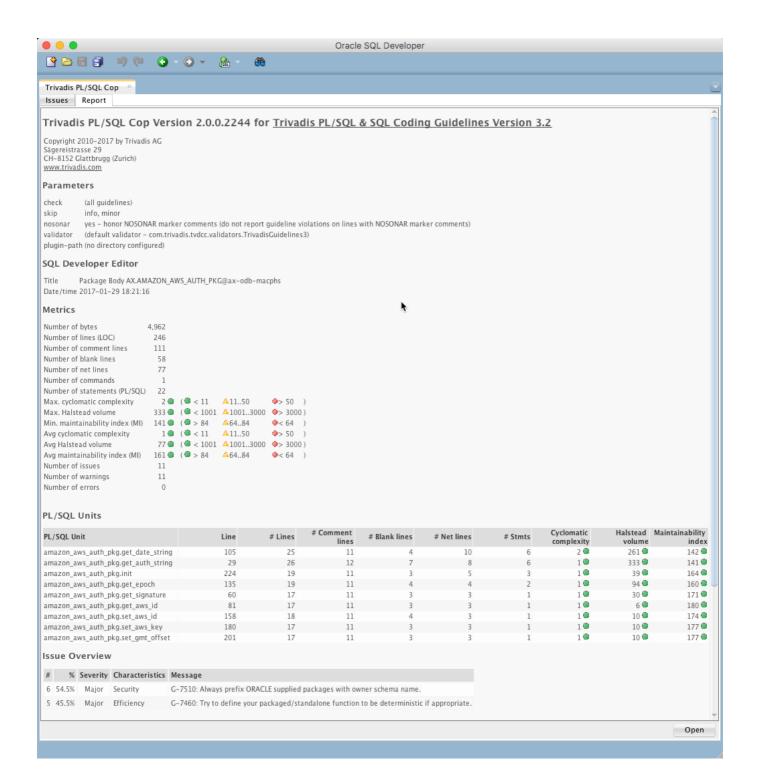


The result of the ckecking process is a list of violations with direct links to the place in the code as well as software metrics like:

- · Cyclomatic complexity
- · Halstead volume
- Maintainability Index
- · Number of lines of code
- · Number of comment lines
- Issue Overview

This statistics are gathered for each program unit in the reviewed code.

#### Software metrics



# **Appendix**

# A - PL/SQL & SQL Coding Guidelines as PDF

These guidelines are primarily produced in HTML using Material for MkDocs.

However, we provide these guidelines also as PDF produced by wkhtmltopdf.



The formatting is not perfect, but it should be adequate for those who want to work with offline documents.

## B - Mapping new guidelines to prior versions

Old Id	New Id	Text	Severity	Change- ability	Efficiency	Maintain- ability	Portab
1	1010	Try to label your sub blocks.	Minor			X	
2	1020	Always have a matching loop or block label.	Minor			X	
3	1030	Avoid defining variables that are not used.	Minor		X	X	
4	1040	Avoid dead code.	Minor			X	
5	1050	Avoid using literals in your code.	Minor	Х			
6	1060	Avoid storing ROWIDs or UROWIDs in database tables.	Major				
7	1070	Avoid nesting comment blocks.	Minor			X	
8	2110	Try to use anchored declarations for variables, constants and types.	Major			х	

9	2120	Try to have a single location to define your types.	Minor	X			
10	2130	Try to use subtypes for constructs used often in your code.	Minor	X			
11	2140	Never initialize variables with NULL.	Minor			Х	
12	2150	Avoid comparisons with NULL value, consider using IS [NOT] NULL.	Blocker				
13	2160	Avoid initializing variables using functions in the declaration section.	Critical				
14	2170	Never overload variables.	Major				
15	2180	Never use quoted identifiers.	Major			Х	
16	2185	Avoid using overly short names for explicitly or implicitly declared identifiers.	Minor			х	
17	2190	Avoid the use of ROWID or UROWID.	Major				
18	2210	Avoid declaring NUMBER variables or subtypes with no precision.	Minor		X		
19	2220	Try to use PLS_INTEGER instead of NUMBER for arithmetic operations with integer values.	Minor		х		
n/a	2230	Try to use SIMPLE_INTEGER datatype when appropriate.	Minor		Х		
20	2310	Avoid using CHAR data type.	Major				
21	2320	Avoid using VARCHAR data type.	Major				
22	2330	Never use zero-length strings to substitute NULL.	Major				

23	2340	Always define your VARCHAR2 variables using CHAR SEMANTIC (if not defined anchored).	Minor				
24	2410	Try to use boolean data type for values with dual meaning.	Minor			X	
25	2510	Avoid using the LONG and LONG RAW data types.	Major				X
26	3110	Always specify the target columns when coding an insert statement.	Major			X	
27	3120	Always use table aliases when your SQL statement involves more than one source.	Major			х	
28	3130	Try to use ANSI SQL-92 join syntax.	Minor			X	X
29	3140	Try to use anchored records as targets for your cursors.	Major			X	
n/a	3150	Try to use identity columns for surrogate keys.	Minor			X	
n/a	3160	Avoid virtual columns to be visible.	Major			X	
n/a	3170	Always use DEFAULT ON NULL declarations to assign default values to table columns if you refuse to store NULL values.	Major				
n/a	3180	Always specify column names instead of positional references in ORDER BY clauses.	Major	X			
n/a	3190	Avoid using NATURAL JOIN.	Major	Х			
30	3210	Always use BULK OPERATIONS (BULK COLLECT, FORALL) whenever you have to execute a DML statement more than 4 times.	Major		X		

31	4110	Always use %NOTFOUND	Minor		X	
		instead of NOT %FOUND to check whether a cursor returned data.				
32	4120	Avoid using %NOTFOUND directly after the FETCH when working with BULK OPERATIONS and LIMIT clause.	Critical			
33	4130	Always close locally opened cursors.	Major	X		
34	4140	Avoid executing any statements between a SQL operation and the usage of an implicit cursor attribute.	Major			
35	4210	Try to use CASE rather than an IF statement with multiple ELSIF paths.	Major		X	
36	4220	Try to use CASE rather than DECODE.	Minor		X	х
37	4230	Always use COALESCE instead of NVL, if parameter 2 of the NVL function is a function call or a SELECT statement.	Critical	X		
38	4240	Always use CASE instead of NVL2 if parameter 2 or 3 of NVL2 is either a function call or a SELECT statement.	Critical	х		
39	4310	Never use GOTO statements in your code.	Major		X	
40	4320	Always label your loops.	Minor		X	
41	4330	Always use a CURSOR FOR loop to process the complete cursor results unless you are using bulk operations.	Minor		X	
42	4340	Always use a NUMERIC FOR loop to process a dense array.	Minor		х	

43	4350	Always use 1 as lower and COUNT() as upper bound when looping through a dense array.	Major				
44	4360	Always use a WHILE loop to process a loose array.	Minor		Х		
45	4370	Avoid using EXIT to stop loop processing unless you are in a basic loop.	Major			х	
46	4375	Always use EXIT WHEN instead of an IF statement to exit from a loop.	Minor			х	
47	4380	Try to label your EXIT WHEN statements.	Minor			Х	
48	4385	Never use a cursor for loop to check whether a cursor returns data.	Major		X		
49	4390	Avoid use of unreferenced FOR loop indexes.	Major		Х		
50	4395	Avoid hard-coded upper or lower bound values with FOR loops.	Minor	Х		х	
n/a	5010	Try to use a error/logging framework for your application.	Critical				
51	5020	Never handle unnamed exceptions using the error number.	Critical			х	
52	5030	Never assign predefined exception names to user defined exceptions.	Blocker				
53	5040	Avoid use of WHEN OTHERS clause in an exception section without any other specific handlers.	Major				
54	n/a	Avoid use of EXCEPTION_INIT pragma for a 20nnn error.	Major				

55	5050	Avoid use of the RAISE_APPLICATION_ERROR built-in procedure with a hard-coded 20nnn error number or hard-coded message.	Major	X		X	
56	5060	Avoid unhandled exceptions	Major				
57	5070	Avoid using Oracle predefined exceptions	Critical				
58	6010	Always use a character variable to execute dynamic SQL.	Major			X	
59	6020	Try to use output bind arguments in the RETURNING INTO clause of dynamic DML statements rather than the USING clause.	Minor			X	
60	7110	Try to use named notation when calling program units.	Major	Х		X	
61	7120	Always add the name of the program unit to its end keyword.	Minor			Х	
62	7130	Always use parameters or pull in definitions rather than referencing external variables in a local program unit.	Major			X	
63	7140	Always ensure that locally defined procedures or functions are referenced.	Major			х	
64	7150	Try to remove unused parameters.	Minor		X	X	
65	7210	Try to keep your packages small. Include only few procedures and functions that are used in the same context.	Minor		Х	X	
66	7220	Always use forward declaration for private functions and procedures.	Minor	X			

67	7230	Avoid declaring global variables public.	Major			
68	7240	Avoid using an IN OUT parameter as IN or OUT only.	Major	X	Х	
69	7310	Avoid standalone procedures  – put your procedures in packages.	Minor		х	
70	7320	Avoid using RETURN statements in a PROCEDURE.	Major		Х	
71	7410	Avoid standalone functions – put your functions in packages.	Minor		х	
73	7420	Always make the RETURN statement the last statement of your function.	Major		х	
72	7430	Try to use no more than one RETURN statement within a function.	Major		х	
74	7440	Never use OUT parameters to return values from a function.	Major			
75	7450	Never return a NULL value from a BOOLEAN function.	Major			
n/a	7460	Try to define your packaged/standalone function to be deterministic if appropriate.	Major	х		
76	7510	Always prefix ORACLE supplied packages with owner schema name.	Major			
77	7710	Avoid cascading triggers.	Major		X	
n/a	7810	Do not use SQL inside PL/SQL to read sequence numbers (or SYSDATE)	Major	X	X	
78	8110	Never use SELECT COUNT(*) if you are only interested in the existence of a row.	Major	X		

n/a	8120	Never check existence of a row to decide whether to create it or not.	Major		x		
79	8210	Always use synonyms when accessing objects of another application schema.	Major	X		X	
n/a	8310	Always validate input parameter size by assigning the parameter to a size limited variable in the declaration section of program unit.	Minor			X	
n/a	8410	Always use application locks to ensure a program unit only running once at a given time.	Minor		Х		
n/a	8510	Always use dbms_application_info to track program process transiently	Minor		Х		

- 1. We see a table and a view as a collection. A jar containing beans is labeled "beans". In Java we call such a collection also "beans" (List<Bean> beans) and name an entry "bean" (for (Bean bean : beans) {...}). An entry of a table is a row (singular) and a table can contain an unbounded number of rows (plural). This and the fact that the Oracle database uses the same concept for their tables and views lead to the decision to use the plural to name a table or a view.
- 2. It used to be good practice to use uppercase keywords and lowercase names to help visualize code structure. But practically all editors support more or less advanced color highlighting of code, similar to the examples in these guidelines. Hence as of version 4.0 we are now recommending all lowercase, as this is easier and faster for the brain to process. You may choose to prefer the old rule however, it is important to always be consistent, like for example keywords always in uppercase and names always in lowercase.
- 3. Tabs are not used because the indentation depends on the editor configuration. We want to ensure that the code looks the same, independent of the editor used. Hence, no tabs. But why not use 8 spaces? That's the traditional value for a tab. When writing a package function the code in the body has an indentation of 3. That's 24 characters as a starting point for the code. We think it's too much. Especially if we try to keep a line below 100 or 80 characters. Other good options would be 2 or 4 spaces. We settled for 3 spaces as a compromise. The indentation is still good visible, but does not use too much space.