PL/SQL & SQL Coding Guidelines

Tips for Development & Operation

Document Version 3.2

BASEL ▪ BERN ▪ BRUGG ▪ DÜSSELDORF ▪ FRANKFURT A.M.
FREIBURG I. BR. ▪ GENF ▪ HAMBURG ▪ KOPENHAGEN ▪ LAUSANNE
MÜNCHEN ▪ STUTTGART ▪ WIEN ▪ ZÜRICH

trivadis
makes IT easier.

PL/SQL & SQL

Coding Guidelines

Trivadis AG

Document Version 3.2

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
President 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
Oracle Developer Advocate for PL/SQL

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
Principal Consultant Trivadis

License

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

Version	Who	Date	Comment
0.1	Troller	17.03.2009	Created.
0.2	Kulessa	04.05.2009	Extended.
0.3	Reiner	12.05.2009	Extended with comments in code.
0.4	Troller	14.05.2009	Extended formatting.
0.5	Kulessa	20.05.2009	Added more CodeXpert rules.
0.6	Troller	22.05.2009	Formatting changes. Added categories to rules.
0.7	Reiner	10.06.2009	Extended with example code commenting.
0.8	Troller	18.06.2009	Finalized.
0.9	Bushnell	23.06.2009	Translation
1.0	Troller	01.07.2009	Ready for inspection
1.1	Troller	19.08.2009	Added Inspection results AFl
1.2	Troller	21.08.2009	Added Inspection results ThM
1.3	Troller	April 2010	Several Corrections New Rule Oracle Supplied Packages
1.3.1	Troller	October 2010	Some formatting
2.0	Troller	August 2011	Added Error Handling Section Added Rule 23 Added Rule 30 Added Rule 31 Added Rule 43
3.0	Troller	1.2016	Added Rules for ORACLE 12c Added SQALE Metrics New guideline numbering system.
3.1	Troller	4.2016	Corrections
3.2	Salvisberg	1.2017	Various small corrections

Table of Contents

1. Introduction

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

1.1. Scope

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

1.2. 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.

1.2.1 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

1.2.2 Severity of the rule

Blocker	Will or may result in a bug.
Critical	Will have a high/direct impact on the maintenance cost.
Major	Will have a medium/potential impact on the maintenance cost.
Minor	Will have a low impact on the maintenance cost.
Info	Very low impact; it is just a remediation cost report.

1.2.3 Keywords used

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.

1.2.4 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.

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.

2. Naming Conventions

2.1. 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.

2.2. 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	P: g	g_version
Local Variable	P: l	l_version
Cursor	P: c	c_employees
Record	P: r	r_employee
Array / Table	P: t	t_employees
Object	P: o	o_employee
Cursor Parameter	P: p	p_empno
In Parameter	P: in	in_empno
Out Parameter	P: out	out_ename
In/Out Parameter	P: io	io_employee
Record Type Definitions	P: r / S: type	r_employee_type
Array/Table Type Definitions	P: t / S: type	t_employees_type
Exception	P: e	e_employee_exists
Constants	P: co	co_empno
Subtypes	S: type	big_string_type

2.3. 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.

Identifier	Naming Convention
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. **Examples:** 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.

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 → Before Row on Insert, Update and Delete
	  ○ _io_id → 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. **Examples:** 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. **Examples**: employee_by_id If more than one function provides the same outcome, you have to be more specific with the name.

Identifier	Naming Convention
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. **Examples**: 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". **Examples**: 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". **Examples:** ddl_audit_trg logon_trg

Identifier	Naming Convention
Table	Plural 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) 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 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. **Examples**: employees_name_uk departments_deptno_uk sct_contracts_uk sct_coli_uk sct_icmd_uk1
View	Plural 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) Add a comment to the database dictionary for every view and every column. Optionally prefixed by a project abbreviation. **Examples**: active_orders orders_v (a view to the orders table)

3. Coding Style

3.1. Formatting

PL/SQL Code Formatting

Rule	Description
1	Keywords are written uppercase, names are written in lowercase.
2	3 space indention.
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.

```
PROCEDURE set_salary(in_employee_id IN employees.employee_id%TYPE) IS
❷ CURSOR c_employees(p_employee_id IN employees.employee_id%TYPE) IS
      SELECT last_name
         ❺ ,first_name
           ,salary
    ❶  FROM employees
       WHERE employee_id = p_employee_id
       ORDER BY last_name
              ,first_name;

   r_employee     c_employees%ROWTYPE;
   l_new_salary   employees.salary%TYPE;
BEGIN
   OPEN  c_employees(p_employee_id => in_employee_id);
❸ FETCH c_employees INTO r_employee;
   CLOSE c_employees;

   new_salary (in_employee_id => in_employee_id
          ❻  ,out_salary     => l_new_salary);

❽ -- Check whether salary has changed
   IF r_employee.salary <>❾l_new_salary THEN
      UPDATE employees
    ❼   SET salary = l_new_salary
       WHERE employee_id = in_employee_id;
 ❹END IF;
END set_salary;
```

3.2. Code Commenting

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.

```
/**
Check whether we passed a valid sql name

@param   in_name  string to be checked
@return  in_name if the string represents a valid sql name
@throws  ORA-44003: invalid SQL name

<b>Call Example:</b>
<pre>
   SELECT TVDAssert.valid_sql_name('TEST') from dual;
   SELECT TVDAssert.valid_sql_name('123') from dual
</pre>
*/
```

4. SQL & PL/SQL Language Usage

4.1. General

1010 Try to label your sub blocks.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Example (bad):

```
BEGIN
   BEGIN
      NULL;
   END;

   BEGIN
      NULL;
   END;
END;
/
```

Example (good):

```
BEGIN
   <<prepare_data>>
   BEGIN
      NULL;
   END prepare_data;

   <<process_data>>
   BEGIN
      NULL;
   END process_data;
END good;
/
```

1020 Always have a matching loop or block label.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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):

```
DECLARE
   i INTEGER;
   co_min_value CONSTANT INTEGER := 1;
   co_max_value CONSTANT INTEGER := 10;
   co_increment CONSTANT INTEGER := 1;
BEGIN
   <<prepare_data>>
   BEGIN
      NULL;
   END;

   <<process_data>>
   BEGIN
      NULL;
   END;

   i := co_min_value;
   <<while_loop>>
   WHILE (i <= co_max_value)
   LOOP
      i := i + co_increment;
   END LOOP;

   <<basic_loop>>
   LOOP
      EXIT basic_loop;
   END LOOP;

   <<for_loop>>
   FOR i IN co_min_value..co_max_value
   LOOP
      sys.dbms_output.put_line(i);
   END LOOP;
END;
/
```

Example (good):

```
DECLARE
   i INTEGER;
   co_min_value CONSTANT INTEGER := 1;
   co_max_value CONSTANT INTEGER := 10;
   co_increment CONSTANT INTEGER := 1;
BEGIN
   <<prepare_data>>
   BEGIN
      NULL;
   END prepare_data;

   <<process_data>>
   BEGIN
      NULL;
   END process_data;

   i := co_min_value;
   <<while_loop>>
   WHILE (i <= co_max_value)
   LOOP
      i := i + co_increment;
   END LOOP while_loop;

   <<basic_loop>>
   LOOP
      EXIT basic_loop;
   END LOOP basic_loop;

   <<for_loop>>
   FOR i IN co_min_value..co_max_value
   LOOP
     sys.dbms_output.put_line(i);
   END LOOP for_loop;
END;
/
```

1030 Avoid defining variables that are not used.

Changeability	**Efficiency**	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Reason:
Unused variables decrease the maintainability and readability of your code.

Example (bad):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE my_proc IS
      l_last_name  employees.last_name%TYPE;
      l_first_name employees.first_name%TYPE;
      co_department_id CONSTANT departments.department_id%TYPE := 10;
      e_good EXCEPTION;
   BEGIN
      SELECT e.last_name
        INTO l_last_name
        FROM employees e
       WHERE e.department_id = co_department_id;
   EXCEPTION
      WHEN no_data_found THEN NULL; -- handle_no_data_found;
      WHEN too_many_rows THEN null; -- handle_too_many_rows;
   END my_proc;
END my_package;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE my_proc IS
      l_last_name  employees.last_name%TYPE;
      co_department_id CONSTANT departments.department_id%TYPE := 10;
      e_good EXCEPTION;
   BEGIN
      SELECT e.last_name
        INTO l_last_name
        FROM employees e
       WHERE e.department_id = co_department_id;

      RAISE e_good;
   EXCEPTION
      WHEN no_data_found THEN NULL; -- handle_no_data_found;
      WHEN too_many_rows THEN null; -- handle_too_many_rows;
   END my_proc;
END my_package;
/
```

1040 Avoid dead code.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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):

```
DECLARE
   co_dept_purchasing CONSTANT departments.department_id%TYPE := 30;
BEGIN
   IF 2=3 THEN
      NULL; -- some dead code here
   END IF;

   NULL; -- some enabled code here

   <<my_loop>>
   LOOP
      EXIT my_loop;
      NULL; -- some dead code here
   END LOOP my_loop;

   NULL; -- some other enabled code here

   CASE
      WHEN 1 = 1 AND 'x' = 'y' THEN
         NULL; -- some dead code here
      ELSE
         NULL; -- some further enabled code here
   END CASE;

   <<my_loop2>>
   FOR r_emp IN (SELECT last_name
                   FROM employees
                  WHERE department_id = co_dept_purchasing
                     OR commission_pct IS NOT NULL
                    AND 5=6)
                -- "OR commission_pct IS NOT NULL" is dead code
   LOOP
      SYS.dbms_output.put_line(r_emp.last_name);
   END LOOP my_loop2;

   RETURN;
   NULL; -- some dead code here
END;
/
```

Example (good):

```
DECLARE
   co_dept_admin CONSTANT dept.deptno%TYPE := 10;
BEGIN
   NULL; -- some enabled code here
   NULL; -- some other enabled code here
   NULL; -- some further enabled code here

   <<my_loop2>>
   FOR r_emp IN (SELECT last_name
                   FROM employees
                  WHERE department_id = co_dept_admin
                     OR commission_pct IS NOT NULL)
   LOOP
      sys.dbms_output.put_line(r_emp.last_name);
   END LOOP my_loop2;
END;
/
```

1050 Avoid using literals in your code.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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.

Example (bad):

```
DECLARE
   l_job employees.job_id%TYPE;
BEGIN
   SELECT e.job_id
     INTO l_job
     FROM employees e
    WHERE e.manager_id IS NULL;

   IF l_job = 'AD_PRES' THEN
      NULL;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      NULL; -- handle_no_data_found;
   WHEN TOO_MANY_ROWS THEN
      NULL; -- handle_too_many_rows;
END;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE constants_up IS
   co_president CONSTANT employees.job_id%TYPE := 'AD_PRES';
END constants_up;
/

DECLARE
   l_job employees.job_id%TYPE;
BEGIN
   SELECT e.job_id
     INTO l_job
     FROM employees e
    WHERE e.manager_id IS NULL;

   IF l_job = constants_up.co_president THEN
      NULL;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      NULL; -- handle_no_data_found;
   WHEN TOO_MANY_ROWS THEN
      NULL; -- handle_too_many_rows;
END;
/
```

1060 Avoid storing ROWIDs or UROWIDs in database tables.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

Reason:
It is an extremely dangerous practice to store ROWIDs 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):

```
BEGIN
   INSERT INTO employees_log (employee_id
                             ,last_name
                             ,first_name
                             ,rid)
   SELECT employee_id
         ,last_name
         ,first_name
         ,ROWID
     FROM employees;
END;
/
```

Example (good):

```
BEGIN
   INSERT INTO employees_log (employee_id
                             ,last_name
                             ,first_name)
   SELECT employee_id
         ,last_name
         ,first_name
     FROM employees;
END;
/
```

1070 Avoid nesting comment blocks.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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;
/
```

Example (good):

```
BEGIN
   /* comment one, comment two */
   NULL;
   -- comment three, comment four
   NULL;
END;
/
```

4.2. Variables & Types

4.2.1 General

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE my_proc IS
      l_last_name  VARCHAR2(20 CHAR);
      co_first_row CONSTANT INTEGER := 1;
   BEGIN
      SELECT e.last_name
        INTO l_last_name
        FROM employees e
       WHERE rownum = co_first_row;
   EXCEPTION
      WHEN no_data_found THEN NULL; -- handle no_data_found
      WHEN too_many_rows THEN NULL; -- handle too_many_rows (impossible)
   END my_proc;
END my_package;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE my_proc IS
      l_last_name  employees.last_name%TYPE;
      co_first_row CONSTANT INTEGER := 1;
   BEGIN
      SELECT e.last_name
        INTO l_last_name
        FROM employees e
       WHERE rownum = co_first_row;
   EXCEPTION
      WHEN no_data_found THEN NULL; -- handle no_data_found
      WHEN too_many_rows THEN NULL; -- handle too_many_rows (impossible)
   END my_proc;
END my_package;
/
```

2120 Try to have a single location to define your types.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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;
/
```

Example (good):

```
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;
/
```

2130 Try to use subtypes for constructs used often in your code.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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

Type	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;
/
```

Example (good):

```
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;
/
```

2140 Never initialize variables with NULL.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Reason:
Variables are initialized to NULL by default.

Example (bad):

```
DECLARE
   l_note big_string_type := NULL;
BEGIN
   sys.dbms_output.put_line(l_note);
END;
/
```

Example (good):

```
DECLARE
   l_note big_string_type;
BEGIN
   sys.dbms_output.put_line(l_note);
END;
/
```

2150 Avoid comparisons with NULL value, consider using IS [NOT] NULL.

Changeability	Efficiency	Maintainability	**Portability**	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	Minor	Info

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 and wrong):

```
DECLARE
   l_value INTEGER;
BEGIN
   IF l_value = NULL THEN
      NULL;
   END IF;
END;
/
```

Example (good):

```
DECLARE
   l_value INTEGER;
BEGIN
   IF l_value IS NULL THEN
      NULL;
   END IF;
END;
/
```

2160 Avoid initializing variables using functions in the declaration section.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	**Critical**	Major	Minor	Info

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;
/
```

Example (good):

```
DECLARE
   co_department_id  CONSTANT INTEGER := 100;
   co_unkown_name    CONSTANT departments.department_name%TYPE := 'unknown';
   l_department_name departments.department_name%TYPE;
BEGIN
   <<init>>
   BEGIN
      l_department_name := department_api.name_by_id(in_id => co_department_id);
   EXCEPTION
      WHEN value_error THEN
         l_department_name := co_unkown_name;
   END init;

   sys.dbms_output.put_line(l_department_name);
END;
/
```

2170 Never overload variables.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

Example (bad):

```
BEGIN
   <<main>>
   DECLARE
      co_main CONSTANT user_objects.object_name%TYPE := 'test_main';
      co_sub CONSTANT user_objects.object_name%TYPE := 'test_sub';
      co_sep CONSTANT user_objects.object_name%TYPE := ' - ';
      l_variable user_objects.object_name%TYPE := co_main;
   BEGIN
      <<sub>>
      DECLARE
         l_variable user_objects.object_name%TYPE := co_sub;
      BEGIN
         sys.dbms_output.put_line(l_variable || co_sep || main.l_variable);
      END sub;
   END main;
END;
/
```

Example (good):

```
BEGIN
   <<main>>
   DECLARE
      co_main CONSTANT user_objects.object_name%TYPE := 'test_main';
      co_sub CONSTANT user_objects.object_name%TYPE := 'test_sub';
      co_sep CONSTANT user_objects.object_name%TYPE := ' - ';
      l_main_variable user_objects.object_name%TYPE := co_main;
   BEGIN
      <<sub>>
      DECLARE
         l_sub_variable user_objects.object_name%TYPE := co_sub;
      BEGIN
         sys.dbms_output.put_line(l_sub_variable || co_sep || l_main_variable);
      END sub;
   END main;
END;
/
```

2180 Never use quoted identifiers.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

Reason:
Quoted identifiers make your code hard to read and maintain.

Example (bad):

```
DECLARE
   "sal+comm" INTEGER;
   "my constant" CONSTANT INTEGER := 1;
   "my exception" EXCEPTION;
BEGIN
   "sal+comm" := "my constant";
EXCEPTION
   WHEN "my exception" THEN
      NULL;
END;
/
```

Example (good):

```
DECLARE
   l_sal_comm     INTEGER;
   co_my_constant CONSTANT INTEGER := 1;
   e_my_exception EXCEPTION;
BEGIN
   l_sal_comm := co_my_constant;
EXCEPTION
   WHEN e_my_exception THEN
      NULL;
END;
/
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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):

```
DECLARE
   i INTEGER;
   c CONSTANT INTEGER := 1;
   e EXCEPTION;
BEGIN
   i := c;
EXCEPTION
   WHEN e THEN
      NULL;
END;
/
```

Example (good):

```
DECLARE
   l_sal_comm     INTEGER;
   co_my_constant CONSTANT INTEGER := 1;
   e_my_exception EXCEPTION;
BEGIN
   l_sal_comm := co_my_constant;
EXCEPTION
   WHEN e_my_exception THEN
      NULL;
END;
/
```

2190 Avoid using ROWID or UROWID.

Changeability	Efficiency	Maintainability	**Portability**	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
DECLARE
   l_department_name departments.department_name%TYPE;
   l_rowid ROWID;
BEGIN
   UPDATE departments
      SET department_name = l_department_name
    WHERE ROWID = l_rowid;
END;
/
```

Example (good):

```
DECLARE
   l_department_name  departments.department_name%TYPE;
   l_department_id    departments.department_id%TYPE;
BEGIN
   UPDATE departments
      SET department_name = l_department_name
    WHERE department_id = l_department_id;
END;
/
```

4.2.2 Numeric Data Types

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

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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;
/
```

Example (good):

```
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;
/
```

2220 Try to use PLS_INTEGER instead of NUMBER for arithmetic operations with integer values.

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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;
/
```

Example (good):

```
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;
/
```

2230 Try to use SIMPLE_INTEGER datatype when appropriate.

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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 SIMPLE_INTEGER has been introduced. It is a sub-type of PLS_INTEGER and covers the same range. The basic difference is that SIMPLE_INTEGER is always NOT NULL. When the value of the declared variable is never going to be null then you can declare it as SIMPLE_INTEGER. Another major difference is that you will never face a numeric overflow using SIMPLE_INTEGER as this data type wraps around without giving any error. SIMPLE_INTEGER data type gives major performance boost over PLS_INTEGER when code is compiled in 'NATIVE' mode, because arithmetic operations on SIMPLE_INTEGER 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;
/
```

Example (good):

```
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;
/
```

4.2.3 Character Data Types

2310 Avoid using CHAR data type.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE PACKAGE types_up
IS
   SUBTYPE description_type IS CHAR(200);
END types_up;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE types_up
IS
   SUBTYPE description_type IS VARCHAR2(200 CHAR);
END types_up;
/
```

2320 Avoid using VARCHAR data type.

Changeability	Efficiency	Maintainability	**Portability**	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE PACKAGE types_up IS
   SUBTYPE description_type IS VARCHAR(200);
END types_up;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE types_up IS
   SUBTYPE description_type IS VARCHAR2(200 CHAR);
END types_up;
/
```

2330 Never use zero-length strings to substitute NULL.

Changeability	Efficiency	Maintainability	**Portability**	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY constants_up IS

   FUNCTION empty_string RETURN VARCHAR2 IS
   BEGIN
       RETURN NULL;
   END empty_string;
END constants_up;
/
```

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

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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 a 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;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE types_up IS
   SUBTYPE description_type IS VARCHAR2(200 CHAR);
END types_up;
/
```

4.2.4 Boolean Data Types

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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;
/
```

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;
/
```

Example (good):

```
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;
/
```

4.2.5 Large Objects

2510 Avoid using the LONG and LONG RAW data types.

Changeability	Efficiency	Maintainability	**Portability**	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE PACKAGE example_package IS
   g_long LONG;
   g_raw  LONG RAW;

   PROCEDURE do_something;
END example_package;
/

CREATE OR REPLACE PACKAGE BODY example_package IS
   PROCEDURE do_something IS
   BEGIN
      NULL;
   END do_something;
END example_package;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE example_package IS
   PROCEDURE do_something;
END example_package;
/

CREATE OR REPLACE PACKAGE BODY example_package IS
   g_long CLOB;
   g_raw  BLOB;

   PROCEDURE do_something IS
   BEGIN
      NULL;
   END do_something;
END example_package;
/
```

4.3. DML and SQL

4.3.1 General

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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);
```

Example (good):

```
INSERT INTO departments (department_id
                        ,department_name
                        ,manager_id
                        ,location_id)
     VALUES (departments_seq.nextval
            ,'Support'
            ,100
            ,10);
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
SELECT last_name
      ,first_name
      ,department_name
  FROM      employees
       JOIN departments USING (department_id)
 WHERE EXTRACT(MONTH FROM hire_date) = EXTRACT(MONTH FROM SYSDATE);
```

Example (better):

```
SELECT e.last_name
      ,e.first_name
      ,d.department_name
  FROM      employees   e
       JOIN departments d ON (e.department_id = d.department_id)
 WHERE EXTRACT(MONTH FROM e.hire_date) = EXTRACT(MONTH FROM SYSDATE);
```

Example (good):
Using meaningful aliases improves the readability of your code.

```
SELECT emp.last_name
      ,emp.first_name
      ,dept.department_name
  FROM      employees   emp
       JOIN departments dept ON (emp.department_id = dept.department_id)
 WHERE EXTRACT(MONTH FROM emp.hire_date) = EXTRACT(MONTH FROM SYSDATE);
```

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.

```
SELECT last_name
      ,first_name
  FROM employees
 WHERE employee_id IN (SELECT employee_id
                         FROM jobs
                        WHERE job_title like '%Manager%');
```

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 `employee_id` column from the `jobs` table.

```
SELECT emp.last_name
      ,emp.first_name
  FROM employees emp
 WHERE emp.employee_id IN (SELECT j.employee_id
                             FROM jobs j
                            WHERE j.job_title like '%Manager%');
```

3130 Try to use ANSI-join syntax.

Changeability	Efficiency	**Maintainability**	**Portability**	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Reason:
ANSI join syntax supports the full outer join. A further advantage of the ANSI join syntax is the separation of the join condition from the query filters.

Example (bad):

```
SELECT e.employee_id
      ,e.last_name
      ,e.first_name
      ,d.department_name
  FROM employees e
      ,departments d
 WHERE e.department_id = d.department_id
   AND EXTRACT(MONTH FROM e.hire_date) = EXTRACT(MONTH FROM SYSDATE);
```

Example (good):

```
SELECT emp.employee_id
      ,emp.last_name
      ,emp.first_name
      ,dept.department_name
  FROM      employees   emp
       JOIN departments dept ON dept.department_id = emp.department_id
 WHERE EXTRACT(MONTH FROM emp.hire_date) = EXTRACT(MONTH FROM SYSDATE);
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
DECLARE
   CURSOR c_employees IS
      SELECT employee_id, first_name, last_name
        FROM employees;
   l_employee_id employees.employee_id%TYPE;
   l_first_name  employees.first_name%TYPE;
   l_last_name   employees.last_name%TYPE;
BEGIN
   OPEN c_employees;
   FETCH c_employees INTO l_employee_id, l_first_name, l_last_name;
   <<process_employees>>
   WHILE c_employees%FOUND
   LOOP
      -- do something with the data
      FETCH c_employees INTO l_employee_id, l_first_name, l_last_name;
   END LOOP process_employees;
   CLOSE c_employees;
END;
/
```

Example (good):

```
DECLARE
   CURSOR c_employees IS
      SELECT employee_id, first_name, last_name
        FROM employees;
   r_employee c_employees%ROWTYPE;
BEGIN
   OPEN c_employees;
   FETCH c_employees INTO r_employee;
   <<process_employees>>
   WHILE c_employees%FOUND
   LOOP
      -- do something with the data
      FETCH c_employees INTO r_employee;
   END LOOP process_employees;
   CLOSE c_employees;
END;
/
```

3150 Try to use identity columns for surrogate keys.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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):

```
CREATE TABLE locations (
   location_id        NUMBER(10)        NOT NULL
  ,location_name      VARCHAR2(60 CHAR) NOT NULL
  ,city               VARCHAR2(30 CHAR) NOT NULL
  ,CONSTRAINT locations_pk PRIMARY KEY (location_id)
  )
/

CREATE SEQUENCE location_seq START WITH 1 CACHE 20
/

CREATE OR REPLACE TRIGGER location_br_i
   BEFORE INSERT ON LOCATIONS
   FOR EACH ROW
BEGIN
   :new.location_id := location_seq.nextval;
END;
/
```

Example (good):

```
CREATE TABLE locations (
   location_id        NUMBER(10)  GENERATED ALWAYS AS IDENTITY
  ,location_name      VARCHAR2(60 CHAR) NOT NULL
  ,city               VARCHAR2(30 CHAR) NOT NULL
  ,CONSTRAINT locations_pk PRIMARY KEY (location_id))
/
```

3160 Avoid virtual columns to be visible.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

Restriction: **ORACLE 12c**

Reason:
In contrast to visible columns, invisible columns are not part of a record defined using %ROWTYPE 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):

```
ALTER TABLE employees
   ADD total_salary GENERATED ALWAYS AS (salary + NVL(commission_pct,0) * salary)
/

DECLARE
   r_employee employees%ROWTYPE;
   l_id employees.employee_id%TYPE := 107;
BEGIN
   r_employee := employee_api.employee_by_id(l_id);
   r_employee.salary := r_employee.salary * constants_up.small_increase();

   UPDATE employees
      SET ROW = r_employee
    WHERE employee_id = l_id;
END;
/

Error report -
ORA-54017: UPDATE operation disallowed ON virtual COLUMNS
ORA-06512: at line 9
```

Example (good):

```
ALTER TABLE employees
   ADD total_salary INVISIBLE GENERATED ALWAYS AS
      (salary + NVL(commission_pct,0) * salary)
/

DECLARE
   r_employee employees%ROWTYPE;
   co_id CONSTANT employees.employee_id%TYPE := 107;
BEGIN
   r_employee := employee_api.employee_by_id(co_id);
   r_employee.salary := r_employee.salary * constants_up.small_increase();

   UPDATE employees
      SET ROW = r_employee
    WHERE employee_id = co_id;
END;
/
```

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

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE TABLE null_test (
   test_case        NUMBER(2) NOT NULL
  ,column_defaulted VARCHAR2(10) DEFAULT 'Default')
/
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);

SELECT * FROM null_test;

TEST_CASE  COLUMN_DEF
---------  -----------
        1  Value
        2  Default
        3
```

Example (good):

```
CREATE TABLE null_test (
   test_case        NUMBER(2) NOT NULL
  ,column_defaulted VARCHAR2(10) DEFAULT ON NULL 'Default')
/
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);

SELECT * FROM null_test;

 TEST_CASE COLUMN_DEF
---------- ----------
         1 Value
         2 Default
         3 Default
```

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

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
SELECT UPPER(first_name)
      ,last_name
      ,salary
      ,hire_date
  FROM employees
 ORDER BY 4,1,3;
```

Example (good):

```
SELECT upper(first_name) AS first_name
      ,last_name
      ,salary
      ,hire_date
  FROM employees
 ORDER BY hire_date
         ,first_name
         ,salary;
```

3190 Avoid using NATURAL JOIN.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
SELECT department_name
      ,last_name
      ,first_name
  FROM employees NATURAL JOIN departments
 ORDER BY department_name
         ,last_name;
DEPARTMENT_NAME                LAST_NAME                 FIRST_NAME
------------------------------ ------------------------- --------------------
Accounting                     Gietz                     William
Executive                      De Haan                   Lex
…

ALTER TABLE departments ADD modified_at DATE DEFAULT ON NULL SYSDATE;
ALTER TABLE employees ADD modified_at DATE DEFAULT ON NULL SYSDATE;

SELECT department_name
      ,last_name
      ,first_name
  FROM employees NATURAL JOIN departments
 ORDER BY department_name
         ,last_name;

No data found
```

Example (good):

```
SELECT d.department_name
      ,e.last_name
      ,e.first_name
  FROM employees   e
  JOIN departments d ON (e.department_id = d.department_id)
 ORDER BY d.department_name
         ,e.last_name;

DEPARTMENT_NAME                LAST_NAME                 FIRST_NAME
------------------------------ ------------------------- --------------------
Accounting                     Gietz                     William
Executive                      De Haan                   Lex
…
```

4.3.2 BULK OPERATIONS

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

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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
   t_employee_ids employee_api.t_employee_ids_type;
   co_increase CONSTANT employees.salary%type := 0.1;
   co_department_id CONSTANT departments.department_id%TYPE := 10;
BEGIN
   t_employee_ids := employee_api.employee_ids_by_department(
                        id_in => co_department_id
                     );
   <<process_employees>>
   FOR i IN 1..t_employee_ids.COUNT()
   LOOP
      UPDATE employees
         SET salary = salary + (salary * co_increase)
       WHERE employee_id = t_employee_ids(i);
   END LOOP process_employees;
END;
/
```

Example (good):

```
DECLARE
   t_employee_ids   employee_api.t_employee_ids_type;
   co_increase      CONSTANT employees.salary%type := 0.1;
   co_department_id CONSTANT departments.department_id%TYPE := 10;
BEGIN
   t_employee_ids := employee_api.employee_ids_by_department(
                        id_in => co_department_id
                     );
   <<process_employees>>
   FORALL i IN 1..t_employee_ids.COUNT()
      UPDATE employees
         SET salary = salary + (salary * co_increase)
       WHERE employee_id = t_employee_ids(i);
END;
/
```

4.4. Control Structures

4.4.1 CURSOR

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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

Example (bad):

```
DECLARE
   CURSOR c_employees IS
      SELECT last_name
            ,first_name
        FROM employees
       WHERE commission_pct IS NOT NULL;

   r_employee  c_employees%ROWTYPE;
BEGIN
   OPEN c_employees;

   <<read_employees>>
   LOOP
      FETCH c_employees INTO r_employee;
      EXIT read_employees WHEN NOT c_employees%FOUND;
   END LOOP read_employees;

   CLOSE c_employees;
END;
/
```

Example (good):

```
DECLARE
   CURSOR c_employees IS
      SELECT last_name
            ,first_name
        FROM employees
       WHERE commission_pct IS NOT NULL;

   r_employee  c_employees%ROWTYPE;
BEGIN
   OPEN c_employees;

   <<read_employees>>
   LOOP
      FETCH c_employees INTO r_employee;
      EXIT read_employees WHEN c_employees%NOTFOUND;
   END LOOP read_employees;

   CLOSE c_employees;
END;
/
```

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

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	**Critical**	Major	Minor	Info

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 NOTFOUND is set to true as soon as the number of rows to be fetched defined by the limit clause is not fulfilled anymore.

```
DECLARE
   CURSOR c_employees IS
      SELECT *
        FROM employees
       ORDER BY employee_id;

   TYPE t_employees_type IS TABLE OF c_employees%ROWTYPE;
   t_employees t_employees_type;
   co_bulk_size CONSTANT SIMPLE_INTEGER := 10;
BEGIN
   OPEN c_employees;

   <<process_employees>>
   LOOP
      FETCH c_employees BULK COLLECT INTO t_employees LIMIT co_bulk_size;
      EXIT process_employees WHEN c_employees%NOTFOUND;

      <<display_employees>>
      FOR i IN 1..t_employees.COUNT()
      LOOP
         sys.dbms_output.put_line(t_employees(i).last_name);
      END LOOP display_employees;
   END LOOP process_employees;

   CLOSE c_employees;
END;
/
```

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

```
DECLARE
   CURSOR c_employees IS
      SELECT *
        FROM employees
       ORDER BY employee_id;

   TYPE t_employees_type IS TABLE OF c_employees%ROWTYPE;
   t_employees t_employees_type;
   co_bulk_size CONSTANT SIMPLE_INTEGER := 10;
BEGIN
   OPEN c_employees;

   <<process_employees>>
   LOOP
      FETCH c_employees BULK COLLECT INTO t_employees LIMIT co_bulk_size;
      EXIT process_employees WHEN t_employees.COUNT() = 0;
      <<display_employees>>
      FOR i IN 1..t_employees.COUNT()
      LOOP
         sys.dbms_output.put_line(t_employees(i).last_name);
      END LOOP display_employees;
   END LOOP process_employees;

   CLOSE c_employees;
END;
/
```

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

```
DECLARE
   CURSOR c_employees IS
      SELECT *
        FROM employees
       ORDER BY employee_id;

   TYPE t_employees_type IS TABLE OF c_employees%ROWTYPE;
   t_employees t_employees_type;
   co_bulk_size CONSTANT SIMPLE_INTEGER := 10;
BEGIN
   OPEN c_employees;

   <<process_employees>>
   LOOP
      FETCH c_employees BULK COLLECT INTO t_employees LIMIT co_bulk_size;
      <<display_employees>>
      FOR i IN 1..t_employees.COUNT()
      LOOP
         sys.dbms_output.put_line(t_employees(i).last_name);
      END LOOP display_employees;
      EXIT process_employees WHEN t_employees.COUNT() <> co_bulk_size;
   END LOOP process_employees;

   CLOSE c_employees;
END;
/
```

4130 Always close locally opened cursors.

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE PACKAGE BODY employee_api AS
   FUNCTION department_salary (in_dept_id IN departments.department_id%TYPE)
      RETURN NUMBER IS
      CURSOR c_department_salary(p_dept_id IN departments.department_id%TYPE) IS
         SELECT sum(salary) AS sum_salary
           FROM employees
          WHERE department_id = p_dept_id;
      r_department_salary c_department_salary%rowtype;
   BEGIN
      OPEN c_department_salary(p_dept_id => in_dept_id);
      FETCH c_department_salary INTO r_department_salary;

      RETURN r_department_salary.sum_salary;
   END department_salary;
END employee_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY employee_api AS
   FUNCTION department_salary (in_dept_id IN departments.department_id%TYPE)
      RETURN NUMBER IS
      CURSOR c_department_salary(p_dept_id IN departments.department_id%TYPE) IS
         SELECT SUM(salary) AS sum_salary
           FROM employees
          WHERE department_id = p_dept_id;
      r_department_salary c_department_salary%rowtype;
   BEGIN
      OPEN c_department_salary(p_dept_id => in_dept_id);
      FETCH c_department_salary INTO r_department_salary;
      CLOSE c_department_salary;
      RETURN r_department_salary.sum_salary;
   END department_salary;
END employee_api;
/
```

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

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE PACKAGE BODY employee_api AS
   co_one CONSTANT SIMPLE_INTEGER := 1;

   PROCEDURE process_dept(in_dept_id IN departments.department_id%TYPE) IS
   BEGIN
      NULL;
   END process_dept;

   PROCEDURE remove_employee (in_employee_id IN employees.employee_id%TYPE) IS
      l_dept_id      employees.department_id%TYPE;
   BEGIN
      DELETE FROM employees
       WHERE employee_id = in_employee_id
       RETURNING department_id INTO l_dept_id;

      process_dept(in_dept_id => l_dept_id);

      IF SQL%ROWCOUNT > co_one THEN
         -- too many rows deleted.
         ROLLBACK;
      END IF;
   END remove_employee;
END employee_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY employee_api AS
   co_one CONSTANT SIMPLE_INTEGER := 1;

   PROCEDURE process_dept(in_dept_id IN departments.department_id%TYPE) IS
   BEGIN
      NULL;
   END process_dept;

   PROCEDURE remove_employee (in_employee_id IN employees.employee_id%TYPE) IS
      l_dept_id      employees.department_id%TYPE;
      l_deleted_emps SIMPLE_INTEGER;
   BEGIN
      DELETE FROM employees
       WHERE employee_id = in_employee_id
       RETURNING department_id INTO l_dept_id;

      l_deleted_emps := SQL%ROWCOUNT;

      process_dept(in_dept_id => l_dept_id);

      IF l_deleted_emps > co_one THEN
         -- too many rows deleted.
         ROLLBACK;
      END IF;
   END remove_employee;
END employee_api;
/
```

4.4.2 CASE / IF / DECODE / NVL / NVL2 / COALESCE

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	**Testability**	
Blocker	Critical	**Major**	Minor	Info

Reason:
IF statements containing multiple ELSIF tend to become complex quickly.

Example (bad):

```
DECLARE
   l_color VARCHAR2(7 CHAR);
BEGIN
   IF l_color = constants_up.co_red THEN
      my_package.do_red();
   ELSIF l_color = constants_up.co_blue THEN
      my_package.do_blue();
   ELSIF l_color = constants_up.co_black THEN
      my_package.do_black();
   END IF;
END;
/
```

Example (good):

```
DECLARE
   l_color types_up.color_code_type;
BEGIN
   CASE l_color
      WHEN constants_up.co_red   THEN
         my_package.do_red();
      WHEN constants_up.co_blue  THEN
         my_package.do_blue();
      WHEN constants_up.co_black THEN
         my_package.do_black();
      ELSE NULL;
   END CASE;
END;
/
```

4220 Try to use CASE rather than DECODE.

Changeability	Efficiency	**Maintainability**	**Portability**	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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.

Example (bad):

```
SELECT DECODE(dummy, 'X', 1
                     , 'Y', 2
                     , 'Z', 3
                     , 0)
  FROM dual;
```

Example (good):

```
SELECT CASE dummy
          WHEN 'X' THEN 1
          WHEN 'Y' THEN 2
          WHEN 'Z' THEN 3
          ELSE 0
       END
  FROM dual;
```

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.

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	**Critical**	Major	Minor	Info

Reason:
The NVL 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):

```
SELECT NVL(dummy, my_package.expensive_null(value_in => dummy))
  FROM dual;
```

Example (good):

```
SELECT COALESCE(dummy, my_package.expensive_null(value_in => dummy))
  FROM dual;
```

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.

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	**Critical**	Major	Minor	Info

Reason:
The NVL2 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):

```
SELECT NVL2(dummy, my_package.expensive_nn(value_in => dummy),
                     my_package.expensive_null(value_in => dummy))
  FROM dual;
```

Example (good):

```
SELECT CASE
          WHEN dummy IS NULL THEN
             my_package.expensive_null(value_in => dummy)
          ELSE
             my_package.expensive_nn(value_in => dummy)
       END
FROM dual;
```

4.4.3 Flow Control

4310 Never use GOTO statements in your code.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	**Testability**	
Blocker	Critical	**Major**	Minor	Info

Example (bad):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE password_check (in_password IN VARCHAR2) IS
      co_digitarray  CONSTANT STRING(10 CHAR)   := '0123456789';
      co_lower_bound CONSTANT SIMPLE_INTEGER := 1;
      co_errno       CONSTANT SIMPLE_INTEGER := -20501;
      co_errmsg      CONSTANT STRING(100 CHAR)  := 'Password must contain a digit.';
      l_isdigit      BOOLEAN     := FALSE;
      l_len_pw       PLS_INTEGER;
      l_len_array    PLS_INTEGER;
   BEGIN
      l_len_pw    := LENGTH(in_password);
      l_len_array := LENGTH(co_digitarray);

      <<check_digit>>
      FOR i IN co_lower_bound .. l_len_array
      LOOP
         <<check_pw_char>>
         FOR j IN co_lower_bound .. l_len_pw
         LOOP
            IF SUBSTR(in_password, j, 1) = SUBSTR(co_digitarray, i, 1) THEN
               l_isdigit := TRUE;
               GOTO check_other_things;
            END IF;
         END LOOP check_pw_char;
      END LOOP check_digit;

      <<check_other_things>>
      NULL;

      IF NOT l_isdigit THEN
         raise_application_error(co_errno, co_errmsg);
      END IF;
   END password_check;
END my_package;/
```

Example (better):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE password_check (in_password IN VARCHAR2) IS
      co_digitarray  CONSTANT STRING(10 CHAR)   := '0123456789';
      co_lower_bound CONSTANT SIMPLE_INTEGER := 1;
      co_errno       CONSTANT SIMPLE_INTEGER := -20501;
      co_errmsg      CONSTANT STRING(100 CHAR)  := 'Password must contain a digit.';
      l_isdigit      BOOLEAN     := FALSE;
      l_len_pw       PLS_INTEGER;
      l_len_array    PLS_INTEGER;
   BEGIN
      l_len_pw    := LENGTH(in_password);
```

```
      l_len_array := LENGTH(co_digitarray);

      <<check_digit>>
      FOR i IN co_lower_bound .. l_len_array
      LOOP
         <<check_pw_char>>
         FOR j IN co_lower_bound .. l_len_pw
         LOOP
            IF SUBSTR(in_password, j, 1) = SUBSTR(co_digitarray, i, 1) THEN
               l_isdigit := TRUE;
               EXIT check_digit; -- early exit condition
            END IF;
         END LOOP check_pw_char;
      END LOOP check_digit;

      <<check_other_things>>
      NULL;

      IF NOT l_isdigit THEN
         raise_application_error(co_errno, co_errmsg);
      END IF;
   END password_check;
END my_package;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE password_check (in_password IN VARCHAR2) IS
      co_digitpattern CONSTANT STRING(10 CHAR)   := '\d';
      co_errno        CONSTANT SIMPLE_INTEGER := -20501;
      co_errmsg       CONSTANT STRING(100 CHAR)  := 'Password must contain a digit.';
   BEGIN
      IF NOT REGEXP_LIKE(in_password, co_digitpattern)
      THEN
         raise_application_error(co_errno, co_errmsg);
      END IF;
   END password_check;
END my_package;
/
```

4320 Always label your loops.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Example (bad):

```
DECLARE
   i INTEGER;
   co_min_value CONSTANT SIMPLE_INTEGER := 1;
   co_max_value CONSTANT SIMPLE_INTEGER := 10;
   co_increment CONSTANT SIMPLE_INTEGER := 1;
BEGIN
   i := co_min_value;
   WHILE (i <= co_max_value)
   LOOP
      i := i + co_increment;
   END LOOP;

   LOOP
      EXIT;
   END LOOP;

   FOR i IN co_min_value..co_max_value
   LOOP
      sys.dbms_output.put_line(i);
   END LOOP;

   FOR r_employee IN (SELECT last_name FROM employees)
   LOOP
      sys.dbms_output.put_line(r_employee.last_name);
   END LOOP;
END;
/
```

Example (good):

```
DECLARE
   i INTEGER;
   co_min_value CONSTANT SIMPLE_INTEGER := 1;
   co_max_value CONSTANT SIMPLE_INTEGER := 10;
   co_increment CONSTANT SIMPLE_INTEGER := 1;
BEGIN
   i := co_min_value;
   <<while_loop>>
   WHILE (i <= co_max_value)
   LOOP
      i := i + co_increment;
   END LOOP while_loop;

   <<basic_loop>>
   LOOP
      EXIT basic_loop;
   END LOOP basic_loop;

   <<for_loop>>
   FOR i IN co_min_value..co_max_value
   LOOP
      sys.dbms_output.put_line(i);
   END LOOP for_loop;

   <<process_employees>>
   FOR r_employee IN (SELECT last_name
                        FROM employees)
   LOOP
      sys.dbms_output.put_line(r_employee.last_name);
   END LOOP process_employees;
END;
/
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Example (bad):

```
DECLARE
   CURSOR c_employees IS
      SELECT employee_id, last_name
        FROM employees;
   r_employee c_employees%ROWTYPE;
BEGIN
   OPEN c_employees;

   <<read_employees>>
   LOOP
      FETCH c_employees INTO r_employee;
      EXIT read_employees WHEN c_employees%NOTFOUND;
      sys.dbms_output.put_line(r_employee.last_name);
   END LOOP read_employees;

   CLOSE c_employees;
END;
/
```

Example (good):

```
DECLARE
   CURSOR c_employees IS
      SELECT employee_id, last_name
        FROM employees;
BEGIN
   <<read_employees>>
   FOR r_employee IN c_employees
   LOOP
      sys.dbms_output.put_line(r_employee.last_name);
   END LOOP read_employees;
END;
/
```

4340 Always use a NUMERIC FOR loop to process a dense array.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Example (bad):

```
DECLARE
   TYPE t_employee_type IS VARRAY(10) OF employees.employee_id%TYPE;
   t_employees t_employee_type;
   co_himuro     CONSTANT INTEGER := 118;
   co_livingston CONSTANT INTEGER := 177;
   co_min_value  CONSTANT SIMPLE_INTEGER := 1;
   co_increment  CONSTANT SIMPLE_INTEGER := 1;
   i PLS_INTEGER;
BEGIN
   t_employees := t_employee_type(co_himuro, co_livingston);
   i           := co_min_value;

   <<process_employees>>
   LOOP
      EXIT process_employees WHEN i > t_employees.COUNT();
      sys.dbms_output.put_line(t_employees(i));
      i := i + co_increment;
   END LOOP process_employees;
END;
/
```

Example (good):

```
DECLARE
   TYPE t_employee_type IS VARRAY(10) OF employees.employee_id%TYPE;
   t_employees t_employee_type;
   co_himuro     CONSTANT INTEGER := 118;
   co_livingston CONSTANT INTEGER := 177;
BEGIN
   t_employees := t_employee_type(co_himuro, co_livingston);

   <<process_employees>>
   FOR i IN 1..t_employees.COUNT()
   LOOP
     sys.dbms_output.put_line(t_employees(i));
   END LOOP process_employees;
END;
/
```

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

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
DECLARE
   TYPE t_employee_type IS TABLE OF employees.employee_id%TYPE;
   t_employees t_employee_type := t_employee_type();
BEGIN
   <<process_employees>>
   FOR i IN t_employees.FIRST()..t_employees.LAST()
   LOOP
      sys.dbms_output.put_line(t_employees(i)); -- some processing
   END LOOP process_employees;
END;
/
```

Example (better):
Raise an unitialized collection error if t_employees is not initialized.

```
DECLARE
   TYPE t_employee_type IS TABLE OF employees.employee_id%TYPE;
   t_employees t_employee_type := t_employee_type();
BEGIN
   <<process_employees>>
   FOR i IN 1..t_employees.COUNT()
   LOOP
      sys.dbms_output.put_line(t_employees(i)); -- some processing
   END LOOP process_employees;
END;
/
```

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.

```
DECLARE
   TYPE t_employee_type IS TABLE OF employees.employee_id%TYPE;
   t_employees t_employee_type := t_employee_type();
BEGIN
   IF t_employees IS NOT NULL THEN
      <<process_employees>>
      FOR i IN 1..t_employees.COUNT()
      LOOP
         sys.dbms_output.put_line(t_employees(i)); -- some processing
      END LOOP process_employees;
   END IF;
END;
/
```

4360 Always use a WHILE loop to process a loose array.

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Reason:
When a loose 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):

```
DECLARE -- raises no_data_found when processing 2nd record
   TYPE t_employee_type IS TABLE OF employees.employee_id%TYPE;
   t_employees t_employee_type;
   co_rogers      CONSTANT INTEGER := 134;
   co_matos       CONSTANT INTEGER := 143;
   co_mcewen      CONSTANT INTEGER := 158;
   co_index_matos CONSTANT INTEGER := 2;
BEGIN
   t_employees := t_employee_type(co_rogers, co_matos, co_mcewen);
   t_employees.DELETE(co_index_matos);

   IF t_employees IS NOT NULL THEN
      <<process_employees>>
      FOR i IN 1..t_employees.COUNT()
      LOOP
         sys.dbms_output.put_line(t_employees(i));
      END LOOP process_employees;
   END IF;
END;
/
```

Example (good):

```
DECLARE
   TYPE t_employee_type IS TABLE OF employees.employee_id%TYPE;
   t_employees t_employee_type;
   co_rogers      CONSTANT INTEGER := 134;
   co_matos       CONSTANT INTEGER := 143;
   co_mcewen      CONSTANT INTEGER := 158;
   co_index_matos CONSTANT INTEGER := 2;
   l_index        PLS_INTEGER;
BEGIN
   t_employees := t_employee_type(co_rogers, co_matos, co_mcewen);
   t_employees.DELETE(co_index_matos);

   l_index := t_employees.FIRST();

   <<process_employees>>
   WHILE l_index IS NOT NULL
   LOOP
      sys.dbms_output.put_line(t_employees(l_index));
      l_index := t_employees.NEXT(l_index);
   END LOOP process_employees;
END;
/
```

4370 Avoid using EXIT to stop loop processing unless you are in a basic loop.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
DECLARE
   i INTEGER;
   co_min_value CONSTANT SIMPLE_INTEGER := 1;
   co_max_value CONSTANT SIMPLE_INTEGER := 10;
   co_increment CONSTANT SIMPLE_INTEGER := 1;
BEGIN
   i := co_min_value;
   <<while_loop>>
   WHILE (i <= co_max_value)
   LOOP
      i := i + co_increment;
      EXIT while_loop WHEN i > co_max_value;
   END LOOP while_loop;

   <<basic_loop>>
   LOOP
      EXIT basic_loop;
   END LOOP basic_loop;

   <<for_loop>>
   FOR i IN co_min_value..co_max_value
   LOOP
      NULL;
      EXIT for_loop WHEN i = co_max_value;
   END LOOP for_loop;

   <<process_employees>>
   FOR r_employee IN (SELECT last_name
                      FROM employees)
   LOOP
      sys.dbms_output.put_line(r_employee.last_name);
      NULL; -- some processing
      EXIT process_employees;
   END LOOP process_employees;
END;
/
```

Example (good):

```
DECLARE
   i INTEGER;
   co_min_value CONSTANT SIMPLE_INTEGER := 1;
   co_max_value CONSTANT SIMPLE_INTEGER := 10;
   co_increment CONSTANT SIMPLE_INTEGER := 1;
BEGIN
   i := co_min_value;
   <<while_loop>>
   WHILE (i <= co_max_value)
   LOOP
      i := i + co_increment;
   END LOOP while_loop;

   <<basic_loop>>
   LOOP
      EXIT basic_loop;
   END LOOP basic_loop;

   <<for_loop>>
   FOR i IN co_min_value..co_max_value
   LOOP
      sys.dbms_output.put_line(i);
   END LOOP for_loop;

   <<process_employees>>
   FOR r_employee IN (SELECT last_name
                        FROM employees)
   LOOP
      sys.dbms_output.put_line(r_employee.last_name); -- some processing
   END LOOP process_employees;
END;
/
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Example (bad):

```
DECLARE
   co_first_year CONSTANT PLS_INTEGER := 1900;
BEGIN
   <<process_employees>>
   LOOP
      my_package.some_processing();

      IF EXTRACT(year FROM SYSDATE) > co_first_year THEN
         EXIT process_employees;
      END IF;

      my_package.some_further_processing();
   END LOOP process_employees;
END;
/
```

Example (good):

```
DECLARE
   co_first_year CONSTANT PLS_INTEGER := 1900;
BEGIN
   <<process_employees>>
   LOOP
      my_package.some_processing();

      EXIT process_employees WHEN EXTRACT(YEAR FROM SYSDATE) > co_first_year;

      my_package.some_further_processing();
   END LOOP process_employees;
END;
/
```

4380 Try to label your EXIT WHEN statements.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Example (bad):

```
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 ';
   l_outerlp PLS_INTEGER;
   l_innerlp PLS_INTEGER;
BEGIN
   l_outerlp := co_init_loop;
   <<outerloop>>
   LOOP
      l_innerlp := co_init_loop;
      l_outerlp := NVL(l_outerlp,co_init_loop) + co_increment;
      <<innerloop>>
      LOOP
         l_innerlp := NVL(l_innerlp, co_init_loop) + co_increment;
         sys.dbms_output.put_line(co_outer_text || l_outerlp ||
                                  co_inner_text || l_innerlp);

         EXIT WHEN l_innerlp = co_exit_value;
      END LOOP innerloop;

      EXIT WHEN l_innerlp = co_exit_value;
   END LOOP outerloop;
END;
/
```

Example (good):

```
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 ';
   l_outerlp PLS_INTEGER;
   l_innerlp PLS_INTEGER;
BEGIN
   l_outerlp := co_init_loop;
   <<outerloop>>
   LOOP
      l_innerlp := co_init_loop;
      l_outerlp := NVL(l_outerlp,co_init_loop) + co_increment;
      <<innerloop>>
      LOOP
         l_innerlp := NVL(l_innerlp, co_init_loop) + co_increment;
         sys.dbms_output.put_line(co_outer_text || l_outerlp ||
                                  co_inner_text || l_innerlp);

         EXIT outerloop WHEN l_innerlp = co_exit_value;
      END LOOP innerloop;
   END LOOP outerloop;
END;
/
```

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

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

Example (bad):

```
DECLARE
   l_employee_found BOOLEAN := FALSE;
   CURSOR c_employees IS
      SELECT employee_id, last_name
        FROM employees;
BEGIN
   <<check_employees>>
   FOR r_employee IN c_employees
   LOOP
      l_employee_found := TRUE;
   END LOOP check_employees;
END;
/
```

Example (good):

```
DECLARE
   l_employee_found BOOLEAN := FALSE;
   CURSOR c_employees IS
      SELECT employee_id, last_name
        FROM employees;
   r_employee c_employees%ROWTYPE;
BEGIN
   OPEN c_employees;
   FETCH c_employees INTO r_employee;
   l_employee_found := c_employees%FOUND;
   CLOSE c_employees;
END;
/
```

4390 Avoid use of unreferenced FOR loop indexes.

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
DECLARE
   l_row   PLS_INTEGER;
   l_value PLS_INTEGER;
   co_lower_bound CONSTANT SIMPLE_INTEGER           := 1;
   co_upper_bound CONSTANT SIMPLE_INTEGER           := 5;
   co_row_incr    CONSTANT SIMPLE_INTEGER           := 1;
   co_value_incr  CONSTANT SIMPLE_INTEGER           := 10;
   co_delimiter   CONSTANT types_up.short_text_type := ' ';
   co_first_value CONSTANT SIMPLE_INTEGER           := 100;
BEGIN
   l_row := co_lower_bound;
   l_value := co_first_value;
   <<for_loop>>
   FOR i IN co_lower_bound .. co_upper_bound
   LOOP
      sys.dbms_output.put_line(l_row || co_delimiter || l_value);
      l_row   := l_row + co_row_incr;
      l_value := l_value + co_value_incr;
   END LOOP for_loop;
END;
/
```

Example (good):

```
DECLARE
   co_lower_bound CONSTANT SIMPLE_INTEGER           := 1;
   co_upper_bound CONSTANT SIMPLE_INTEGER           := 5;
   co_value_incr  CONSTANT SIMPLE_INTEGER           := 10;
   co_delimiter   CONSTANT types_up.short_text_type := ' ';
   co_first_value CONSTANT SIMPLE_INTEGER           := 100;
BEGIN
   <<for_loop>>
   FOR i IN co_lower_bound .. co_upper_bound
   LOOP
      sys.dbms_output.put_line(i || co_delimiter ||
                               to_char(co_first_value + i * co_value_incr));
   END LOOP for_loop;
END;
/
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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):

```
BEGIN
   <<for_loop>>
   FOR i IN 1..5
   LOOP
      sys.dbms_output.put_line(i);
   END LOOP for_loop;
END;
/
```

Example (good):

```
DECLARE
   co_lower_bound CONSTANT SIMPLE_INTEGER := 1;
   co_upper_bound CONSTANT SIMPLE_INTEGER := 5;
BEGIN
   <<for_loop>>
   FOR i IN co_lower_bound..co_upper_bound
   LOOP
      sys.dbms_output.put_line(i);
   END LOOP for_loop;
END;
/
```

4.5. Exception Handling

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

Changeability	Efficiency	Maintainability	Portability	
Reliability	**Reusability**	Security	**Testability**	
Blocker	**Critical**	Major	Minor	Info

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

5020 Never handle unnamed exceptions using the error number.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	**Critical**	Major	Minor	Info

Example (bad):

```
DECLARE
   co_no_data_found CONSTANT INTEGER := -1;
BEGIN
   my_package.some_processing(); -- some code which raises an exception
EXCEPTION
   WHEN TOO_MANY_ROWS THEN
      my_package.some_further_processing();
   WHEN OTHERS THEN
      IF SQLCODE = co_no_data_found THEN
         NULL;
      END IF;
END;
/
```

Example (good):

```
BEGIN
   my_package.some_processing(); -- some code which raises an exception
EXCEPTION
   WHEN TOO_MANY_ROWS THEN
      my_package.some_further_processing();
   WHEN NO_DATA_FOUND THEN
      NULL; -- handle no_data_found
END;
/
```

5030 Never assign predefined exception names to user defined exceptions.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	**Testability**	
Blocker	Critical	Major	Minor	Info

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 has not the exception number assigned, which is raise when the SELECT statement does not find any rows.

```
DECLARE
   l_dummy dual.dummy%TYPE;
   no_data_found    EXCEPTION;
   co_rownum        CONSTANT SIMPLE_INTEGER           := 0;
   co_no_data_found CONSTANT types_up.short_text_type := 'no_data_found';
BEGIN
   SELECT dummy
     INTO l_dummy
     FROM dual
    WHERE ROWNUM = co_rownum;

   IF l_dummy IS NULL THEN
      RAISE no_data_found;
   END IF;
EXCEPTION
   WHEN no_data_found THEN
      sys.dbms_output.put_line(co_no_data_found);
END;
/

Error report -
ORA-01403: no data found
ORA-06512: at line 5
01403. 00000 -  "no data found"
*Cause:    No data was found from the objects.
*Action:   There was no data from the objects which may be due to end of fetch.
```

Example (good)

```
DECLARE
   l_dummy dual.dummy%TYPE;
   empty_value      EXCEPTION;
   co_rownum        CONSTANT simple_integer           := 0;
   co_empty_value   CONSTANT types_up.short_text_type := 'empty_value';
   co_no_data_found CONSTANT types_up.short_text_type := 'no_data_found';
BEGIN
   SELECT dummy
     INTO l_dummy
     FROM dual
    WHERE rownum = co_rownum;

   IF l_dummy IS NULL THEN
      RAISE empty_value;
   END IF;
EXCEPTION
   WHEN empty_value THEN
      sys.dbms_output.put_line(co_empty_value);
   WHEN no_data_found THEN
      sys.dbms_output.put_line(co_no_data_found);
END;
/
```

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

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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

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;
/
```

5050 Avoid use of the RAISE_APPLICATION_ERROR built-in procedure with a hard-coded -20nnn error number or hard-coded message.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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;
/
```

Example (good):

```
BEGIN
   err_up.raise(in_error => err.co_invalid_employee_id);
END;
/
```

5060 Avoid unhandled exceptions.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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)
      RETURN departments.department_name%TYPE IS
      l_department_name departments.department_name%TYPE;
   BEGIN
      SELECT department_name
        INTO l_department_name
        FROM departments
       WHERE department_id = in_id;

      RETURN l_department_name;
   END name_by_id;
END department_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY department_api IS
   FUNCTION name_by_id (in_id IN departments.department_id%TYPE)
      RETURN departments.department_name%TYPE IS
      l_department_name departments.department_name%TYPE;
   BEGIN
      SELECT department_name
        INTO l_department_name
        FROM departments
       WHERE department_id = in_id;

      RETURN l_department_name;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN RETURN NULL;
      WHEN TOO_MANY_ROWS THEN RAISE;
   END name_by_id;
END department_api;
/
```

5070 Avoid using Oracle predefined exceptions.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	**Critical**	Major	Minor	Info

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;
/
```

Example (good):

```
DECLARE
   my_exception EXCEPTION;
BEGIN
   RAISE my_exception;
END;
/
```

4.6. Dynamic SQL

6010 Always use a character variable to execute dynamic SQL.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	**Testability**	
Blocker	Critical	**Major**	Minor	Info

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;
BEGIN
   EXECUTE IMMEDIATE 'select employees_seq.nextval from dual' INTO l_next_val;
END;
/
```

Example (good):

```
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;
/
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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
   PROCEDURE upd_salary (in_employee_id  IN     employees.employee_id%TYPE
                        ,in_increase_pct IN     types_up.percentage
                        ,out_new_salary     OUT employees.salary%TYPE)
   IS
      co_sql_stmt CONSTANT types_up.big_string_type := '
           UPDATE employees SET salary = salary + (salary / 100 * :1)
            WHERE employee_id = :2
        RETURNING salary INTO :3';
   BEGIN
     EXECUTE IMMEDIATE co_sql_stmt
           USING in_increase_pct, in_employee_id, OUT out_new_salary;
   END upd_salary;
END employee_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY employee_api IS
   PROCEDURE upd_salary (in_employee_id  IN     employees.employee_id%TYPE
                        ,in_increase_pct IN     types_up.percentage
                        ,out_new_salary     OUT employees.salary%TYPE)
   IS
      co_sql_stmt CONSTANT types_up.big_string_type :=
          'UPDATE employees SET salary = salary + (salary / 100 * :1)
            WHERE employee_id = :2
        RETURNING salary INTO :3';
   BEGIN
     EXECUTE IMMEDIATE co_sql_stmt
           USING in_increase_pct, in_employee_id
           RETURNING INTO out_new_salary;
   END upd_salary;
END employee_api;
/
```

4.7. Stored Objects

4.7.1 General

7110 Try to use named notation when calling program units.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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, NVL, ROUND, etc.) but should be followed for any other stored object having more than one parameter.

Example (bad):

```
DECLARE
   r_employee employees%rowtype;
   co_id CONSTANT employees.employee_id%type := 107;
BEGIN
   r_employee := employee_api.employee_by_id(co_id);
END;
/
```

Example (good):

```
DECLARE
   r_employee employees%rowtype;
   co_id CONSTANT employees.employee_id%type := 107;
BEGIN
   r_employee := employee_api.employee_by_id(in_employee_id => co_id);
END;
/
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Example (bad):

```
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;
   BEGIN
      SELECT *
        INTO r_employee
        FROM employees
       WHERE employee_id = in_employee_id;

      RETURN r_employee;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         NULL;
      WHEN TOO_MANY_ROWS THEN
         RAISE;
   END;
END;
/
```

Example (good):

```
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;
   BEGIN
      SELECT *
        INTO r_employee
        FROM employees
       WHERE employee_id = in_employee_id;

      RETURN r_employee;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         NULL;
      WHEN TOO_MANY_ROWS THEN
         RAISE;
   END employee_by_id;
END employee_api;/
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	**Testability**	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE PACKAGE BODY EMPLOYEE_API IS
   PROCEDURE calc_salary (in_employee_id IN employees.employee_id%TYPE) IS
      r_emp employees%rowtype;

      FUNCTION commission RETURN NUMBER IS
         l_commission employees.salary%TYPE := 0;
      BEGIN
         IF r_emp.commission_pct IS NOT NULL
         THEN
            l_commission := r_emp.salary * r_emp.commission_pct;
         END IF;

         RETURN l_commission;
      END commission;
   BEGIN
      SELECT *
        INTO r_emp
        FROM employees
       WHERE employee_id = in_employee_id;

      SYS.DBMS_OUTPUT.PUT_LINE(r_emp.salary + commission());
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         NULL;
      WHEN TOO_MANY_ROWS THEN
         NULL;
   END calc_salary;
END employee_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY EMPLOYEE_API IS
   PROCEDURE calc_salary (in_employee_id IN employees.employee_id%TYPE) IS
      r_emp employees%rowtype;

      FUNCTION commission (in_salary   IN employees.salary%TYPE
                          ,in_comm_pct IN employees.commission_pct%TYPE)
         RETURN NUMBER IS
         l_commission employees.salary%TYPE := 0;
      BEGIN
         IF in_comm_pct IS NOT NULL THEN
            l_commission := in_salary * in_comm_pct;
         END IF;

         RETURN l_commission;
      END commission;
   BEGIN
      SELECT *
        INTO r_emp
        FROM employees
       WHERE employee_id = in_employee_id;

      SYS.DBMS_OUTPUT.PUT_LINE(
         r_emp.salary + commission(in_salary   => r_emp.salary
                                  ,in_comm_pct => r_emp.commission_pct)
      );
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         NULL;
      WHEN TOO_MANY_ROWS THEN
         NULL;
   END calc_salary;
END employee_api;
/
```

7140 Always ensure that locally defined procedures or functions are referenced.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE my_procedure IS
      FUNCTION my_func RETURN NUMBER IS
         co_true CONSTANT INTEGER := 1;
      BEGIN
         RETURN co_true;
      END my_func;
   BEGIN
       NULL;
   END my_procedure;
END my_package;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE my_procedure IS
      FUNCTION my_func RETURN NUMBER IS
         co_true CONSTANT INTEGER := 1;
      BEGIN
         RETURN co_true;
      END my_func;
   BEGIN
       sys.dbms_output.put_line(my_func());
   END my_procedure;
END my_package;
/
```

7150 Try to remove unused parameters.

Changeability	**Efficiency**	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Reason:
You should go through your programs and remove any partameter that is no longer used.

Example (bad):

```
CREATE OR REPLACE PACKAGE BODY department_api IS
   FUNCTION name_by_id (in_department_id IN departments.department_id%TYPE
                       ,in_manager_id    IN departments.manager_id%TYPE)
      RETURN departments.department_name%TYPE IS
      l_department_name departments.department_name%TYPE;
   BEGIN
      <<find_department>>
      BEGIN
         SELECT department_name
           INTO l_department_name
           FROM departments
          WHERE department_id = in_department_id;
      EXCEPTION
         WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN
            l_department_name := NULL;
      END find_department;

      RETURN l_department_name;
   END name_by_id;
END department_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY department_api IS
   FUNCTION name_by_id (in_department_id IN departments.department_id%TYPE)
      RETURN departments.department_name%TYPE IS
      l_department_name departments.department_name%TYPE;
   BEGIN
      <<find_department>>
      BEGIN
         SELECT department_name
           INTO l_department_name
           FROM departments
          WHERE department_id = in_department_id;
      EXCEPTION
         WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN
            l_department_name := NULL;
      END find_department;

      RETURN l_department_name;
   END name_by_id;
END department_api;/
```

4.7.2 Packages

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

Changeability	**Efficiency**	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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.

7220 Always use forward declaration for private functions and procedures.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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

Example (bad):

```
CREATE OR REPLACE PACKAGE department_api IS
   PROCEDURE del (in_department_id IN departments.department_id%TYPE);
END department_api;
/

CREATE OR REPLACE PACKAGE BODY department_api IS
   FUNCTION does_exist (in_department_id IN departments.department_id%TYPE)
      RETURN BOOLEAN IS
      l_return PLS_INTEGER;
   BEGIN
      <<check_row_exists>>
      BEGIN
         SELECT 1
           INTO l_return
           FROM departments
          WHERE department_id = in_department_id;
      EXCEPTION
         WHEN no_data_found OR too_many_rows THEN
            l_return := 0;
      END check_row_exists;

      RETURN l_return = 1;
   END does_exist;

   PROCEDURE del (in_department_id IN departments.department_id%TYPE) IS
   BEGIN
      IF does_exist(in_department_id) THEN
        NULL;
      END IF;
   END del;
END department_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE department_api IS
   PROCEDURE del (in_department_id IN departments.department_id%TYPE);
END department_api;
/

CREATE OR REPLACE PACKAGE BODY department_api IS
   FUNCTION does_exist (in_department_id IN departments.department_id%TYPE)
      RETURN BOOLEAN;

   PROCEDURE del (in_department_id IN departments.department_id%TYPE) IS
   BEGIN
      IF does_exist(in_department_id) THEN
        NULL;
      END IF;
   END del;

   FUNCTION does_exist (in_department_id IN departments.department_id%TYPE)
      RETURN BOOLEAN IS
      l_return PLS_INTEGER;
   BEGIN
      <<check_row_exists>>
      BEGIN
         SELECT 1
           INTO l_return
           FROM departments
          WHERE department_id = in_department_id;
      EXCEPTION
         WHEN no_data_found OR too_many_rows THEN
            l_return := 0;
      END check_row_exists;

      RETURN l_return = 1;
   END does_exist;
END department_api;
/
```

7230 Avoid declaring global variables public.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

Reason:
You should always declare package-level data 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.

Example (bad):

```
CREATE OR REPLACE PACKAGE employee_api AS
   co_min_increase CONSTANT types_up.sal_increase_type := 0.01;
   co_max_increase CONSTANT types_up.sal_increase_type := 0.5;
   g_salary_increase types_up.sal_increase_type := co_min_increase;

   PROCEDURE set_salary_increase (in_increase IN types_up.sal_increase_type);
   FUNCTION salary_increase RETURN types_up.sal_increase_type;
END employee_api;
/

CREATE OR REPLACE PACKAGE BODY employee_api AS
   PROCEDURE set_salary_increase (in_increase IN types_up.sal_increase_type) IS
   BEGIN
      g_salary_increase := GREATEST(LEAST(in_increase,co_max_increase)
                                   ,co_min_increase);
   END set_salary_increase;

   FUNCTION salary_increase RETURN types_up.sal_increase_type IS
   BEGIN
      RETURN g_salary_increase;
   END salary_increase;
END employee_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE employee_api AS
   PROCEDURE set_salary_increase (in_increase IN types_up.sal_increase_type);
   FUNCTION salary_increase RETURN types_up.sal_increase_type;
END employee_api;
/

CREATE OR REPLACE PACKAGE BODY employee_api AS
   g_salary_increase types_up.sal_increase_type(4,2);

   PROCEDURE init;

   PROCEDURE set_salary_increase (in_increase IN types_up.sal_increase_type) IS
   BEGIN
      g_salary_increase := GREATEST(LEAST(in_increase
                                          ,constants_up.max_salary_increase())
                                    ,constants_up.min_salary_increase());
   END set_salary_increase;

   FUNCTION salary_increase RETURN types_up.sal_increase_type IS
   BEGIN
      RETURN g_salary_increase;
   END salary_increase;

   PROCEDURE init
   IS
   BEGIN
      g_salary_increase := constants_up.min_salary_increase();
   END init;
BEGIN
   init();
END employee_api;
/
```

7240 Avoid using an IN OUT parameter as IN or OUT only.

Changeability	**Efficiency**	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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.

Example (bad):

```
-- Bad
CREATE OR REPLACE PACKAGE BODY employee_up IS
   PROCEDURE rcv_emp (io_first_name     IN OUT employees.first_name%TYPE
                     ,io_last_name      IN OUT employees.last_name%TYPE
                     ,io_email          IN OUT employees.email%TYPE
                     ,io_phone_number   IN OUT employees.phone_number%TYPE
                     ,io_hire_date      IN OUT employees.hire_date%TYPE
                     ,io_job_id         IN OUT employees.job_id%TYPE
                     ,io_salary         IN OUT employees.salary%TYPE
                     ,io_commission_pct IN OUT employees.commission_pct%TYPE
                     ,io_manager_id     IN OUT employees.manager_id%TYPE
                     ,io_department_id  IN OUT employees.department_id%TYPE
                     ,in_wait                  INTEGER) IS
      l_status PLS_INTEGER;
      co_dflt_pipe_name CONSTANT STRING(30 CHAR) := 'MyPipe';
      co_ok CONSTANT PLS_INTEGER := 1;
   BEGIN
      -- Receive next message and unpack for each column.
      l_status := SYS.dbms_pipe.receive_message(pipename => co_dflt_pipe_name
                                               ,timeout  => in_wait);
      IF l_status = co_ok THEN
         SYS.dbms_pipe.unpack_message (io_first_name);
         SYS.dbms_pipe.unpack_message (io_last_name);
         SYS.dbms_pipe.unpack_message (io_email);
         SYS.dbms_pipe.unpack_message (io_phone_number);
         SYS.dbms_pipe.unpack_message (io_hire_date);
         SYS.dbms_pipe.unpack_message (io_job_id);
         SYS.dbms_pipe.unpack_message (io_salary);
         SYS.dbms_pipe.unpack_message (io_commission_pct);
         SYS.dbms_pipe.unpack_message (io_manager_id);
         SYS.dbms_pipe.unpack_message (io_department_id);
      END IF;
   END rcv_emp;
END employee_up;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY employee_up IS
   PROCEDURE rcv_emp (OUT_first_name     OUT employees.first_name%TYPE
                     ,OUT_last_name      OUT employees.last_name%TYPE
                     ,OUT_email          OUT employees.email%TYPE
                     ,OUT_phone_number   OUT employees.phone_number%TYPE
                     ,OUT_hire_date      OUT employees.hire_date%TYPE
                     ,OUT_job_id         OUT employees.job_id%TYPE
                     ,OUT_salary         OUT employees.salary%TYPE
                     ,OUT_commission_pct OUT employees.commission_pct%TYPE
                     ,OUT_manager_id     OUT employees.manager_id%TYPE
                     ,OUT_department_id  OUT employees.department_id%TYPE
                     ,in_wait            IN     INTEGER) IS
      l_status PLS_INTEGER;
      co_dflt_pipe_name CONSTANT STRING(30 CHAR) := 'MyPipe';
      co_ok CONSTANT PLS_INTEGER := 1;
   BEGIN
      -- Receive next message and unpack for each column.
      l_status := SYS.dbms_pipe.receive_message(pipename => co_dflt_pipe_name
                                               ,timeout  => in_wait);
      IF l_status = co_ok THEN
         SYS.dbms_pipe.unpack_message (out_first_name);
         SYS.dbms_pipe.unpack_message (out_last_name);
         SYS.dbms_pipe.unpack_message (out_email);
         SYS.dbms_pipe.unpack_message (out_phone_number);
         SYS.dbms_pipe.unpack_message (out_hire_date);
         SYS.dbms_pipe.unpack_message (out_job_id);
         SYS.dbms_pipe.unpack_message (out_salary);
         SYS.dbms_pipe.unpack_message (out_commission_pct);
         SYS.dbms_pipe.unpack_message (out_manager_id);
         SYS.dbms_pipe.unpack_message (out_department_id);
      END IF;
   END rcv_emp;
END employee_up;
/
```

4.7.3 Procedures

7310 Avoid standalone procedures – put your procedures in packages.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Example (bad):

```
CREATE OR REPLACE PROCEDURE my_procedure IS
BEGIN
   NULL;
END my_procedure;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE my_package IS
   PROCEDURE my_procedure;
END my_package;
/

CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE my_procedure IS
   BEGIN
      NULL;
   END my_procedure;
END my_package;
/
```

7320 Avoid using RETURN statements in a PROCEDURE.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	**Testability**	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE my_procedure IS
      l_idx SIMPLE_INTEGER := 1;
      co_modulo CONSTANT SIMPLE_INTEGER := 7;
   BEGIN
      <<mod7_loop>>
      LOOP
        IF MOD(l_idx,co_modulo) = 0 THEN
           RETURN;
        END IF;

        l_idx := l_idx + 1;
      END LOOP mod7_loop;
   END my_procedure;
END my_package;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   PROCEDURE my_procedure IS
      l_idx SIMPLE_INTEGER := 1;
      co_modulo CONSTANT SIMPLE_INTEGER := 7;
   BEGIN
      <<mod7_loop>>
      LOOP
        EXIT mod7_loop WHEN MOD(l_idx,co_modulo) = 0;

        l_idx := l_idx + 1;
      END LOOP mod7_loop;
   END my_procedure;
END my_package;
/
```

4.7.4 Functions

7410 Avoid standalone functions – put your functions in packages.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

Example (bad):

```
CREATE OR REPLACE FUNCTION my_function RETURN VARCHAR2 IS
BEGIN
   RETURN NULL;
END my_function;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   FUNCTION my_function RETURN VARCHAR2 IS
   BEGIN
      RETURN NULL;
   END my_function;
END my_package;
/
```

7420 Always make the RETURN statement the last statement of your function.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

Example (bad):

```
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
      l_ret PLS_INTEGER;
   BEGIN
      l_ret := in_from;
      <<for_loop>>
      FOR i IN in_from .. in_to
      LOOP
         l_ret := l_ret + i;
         IF i = in_to THEN
            RETURN l_ret;
         END IF;
      END LOOP for_loop;
   END my_function;
END my_package;
/
```

Example (good):

```
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
      l_ret PLS_INTEGER;
   BEGIN
      l_ret := in_from;
      <<for_loop>>
      FOR i IN in_from .. in_to
      LOOP
         l_ret := l_ret + i;
      END LOOP for_loop;
      RETURN l_ret;
   END my_function;
END my_package;
/
```

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	**Testability**	
Blocker	Critical	**Major**	Minor	Info

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

Example (bad):

```
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
      IF in_value = co_yes THEN
         RETURN TRUE;
      ELSE
         RETURN FALSE;
      END IF;
   END my_function;
END my_package;
/
```

Example (better):

```
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;
      l_ret BOOLEAN;
   BEGIN
      IF in_value = co_yes THEN
         l_ret := TRUE;
      ELSE
         l_ret := FALSE;
      END IF;

      RETURN l_ret;
   END my_function;
END my_package;
/
```

Example (good):

```
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;
/
```

7440 Never use OUT parameters to return values from a function.

Changeability	Efficiency	Maintainability	Portability	
Reliability	**Reusability**	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   FUNCTION my_function RETURN DATE IS
   BEGIN
      RETURN SYSDATE;
   END my_function;
END my_package;
/
```

7450 Never return a NULL value from a BOOLEAN function.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	**Testability**	
Blocker	Critical	**Major**	Minor	Info

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;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY my_package IS
   FUNCTION my_function RETURN BOOLEAN IS
   BEGIN
      RETURN TRUE;
   END my_function;
END my_package;
/
```

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

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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;
/
```

Example (good):

```
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;
/
```

4.7.5 Oracle Supplied Packages

7510 Always prefix ORACLE supplied packages with owner schema name.

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	**Security**	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
DECLARE
   co_hello_world CONSTANT STRING(30 CHAR) := 'Hello World';
BEGIN
   dbms_output.put_line(co_hello_world);
END;
/
```

Example (good):

```
DECLARE
   co_hello_world CONSTANT STRING(30 CHAR) := 'Hello World';
BEGIN
   sys.dbms_output.put_line(co_hello_world);
END;
/
```

4.7.6 Object Types

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

4.7.7 Trigger

7710 Avoid cascading triggers.

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	Reusability	Security	**Testability**	
Blocker	Critical	**Major**	Minor	Info

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):

```
CREATE OR REPLACE TRIGGER dept_br_u
BEFORE UPDATE ON departments FOR EACH ROW
BEGIN
   INSERT INTO departments_hist (department_id
                                ,department_name
                                ,manager_id
                                ,location_id
                                ,modification_date)
        VALUES (:OLD.department_id
               ,:OLD.department_name
               ,:OLD.manager_id
               ,:OLD.location_id
               ,SYSDATE);
END;
/
CREATE OR REPLACE TRIGGER dept_hist_br_i
BEFORE INSERT ON departments_hist FOR EACH ROW
BEGIN
   INSERT INTO departments_log (department_id
                               ,department_name
                               ,modification_date)
                        VALUES (:NEW.department_id
                               ,:NEW.department_name
                               ,SYSDATE);
END;
/
```

Example (good):

```
CREATE OR REPLACE TRIGGER dept_br_u
BEFORE UPDATE ON departments FOR EACH ROW
BEGIN
   INSERT INTO departments_hist (department_id
                                ,department_name
                                ,manager_id
                                ,location_id
                                ,modification_date)
        VALUES (:OLD.department_id
               ,:OLD.department_name
               ,:OLD.manager_id
               ,:OLD.location_id
               ,SYSDATE);

   INSERT INTO departments_log (department_id
                               ,department_name
                               ,modification_date)
                        VALUES (:OLD.department_id
                               ,:OLD.department_name
                               ,SYSDATE);

END;
/
```

4.7.8 Sequences

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

Changeability	**Efficiency**	**Maintainability**	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
DECLARE
   l_sequence_number employees.emloyee_id%type;
BEGIN
   SELECT employees_seq.NEXTVAL
     INTO l_sequence_number
     FROM DUAL;
END;
```

Example (good):

```
DECLARE
   l_sequence_number employees.emloyee_id%type;
BEGIN
   l_sequence_number := employees_seq.NEXTVAL;
END;
```

4.8. Patterns

4.8.1 Checking the Number of Rows

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

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
DECLARE
   l_count PLS_INTEGER;
   co_zero   CONSTANT SIMPLE_INTEGER := 0;
   co_salary CONSTANT employees.salary%TYPE := 5000;
BEGIN
   SELECT count(*)
     INTO l_count
     FROM employees
    WHERE salary < co_salary;
    IF l_count > co_zero THEN
       <<emp_loop>>
       FOR r_emp IN (SELECT employee_id
                       FROM employees)
       LOOP
          IF r_emp.salary < co_salary THEN
             my_package.my_proc(in_employee_id => r_emp.employee_id);
          END IF;
       END LOOP emp_loop;
    END IF;
END;
/
```

Example (good):

```
DECLARE
   co_salary CONSTANT employees.salary%TYPE := 5000;
BEGIN
    <<emp_loop>>
    FOR r_emp IN (SELECT e1.employee_id
                    FROM employees e1
                   WHERE EXISTS(SELECT e2.salary
                                  FROM employees e2
                                 WHERE e2.salary < co_salary))
    LOOP
       my_package.my_proc(in_employee_id => r_emp.employee_id);
    END LOOP emp_loop;
END;
/
```

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

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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 preventioin of duplicate records.

Example (bad):

```
CREATE OR REPLACE PACKAGE BODY department_api IS
   PROCEDURE ins (in_r_department IN departments%ROWTYPE) IS
      l_count PLS_INTEGER;
   BEGIN
      SELECT count(*)
        INTO l_count
        FROM departments
       WHERE department_id = in_r_department.department_id;

       IF l_count = 0 THEN
          INSERT INTO departments
               VALUES in_r_department;
       END IF;
   END ins;
END department_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY department_api IS
   PROCEDURE ins (in_r_department IN departments%ROWTYPE) IS
   BEGIN
      INSERT INTO departments
           VALUES in_r_department;
   EXCEPTION
      WHEN dup_val_on_index THEN NULL; -- handle exception
   END ins;
END department_api;
/
```

4.8.2 Access objects of foreign application schemas

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

Changeability	Efficiency	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	**Major**	Minor	Info

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):

```
DECLARE
   l_product_name oe.products.product_name%TYPE;
   co_price CONSTANT oe.products@list_price%TYPE := 1000;
BEGIN
   SELECT p.product_name
     INTO l_product_name
     FROM oe.products p
    WHERE list_price > co_price;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      NULL; -- handle_no_data_found;
   WHEN TOO_MANY_ROWS THEN
      NULL; -- handle_too_many_rows;
END;
/
```

Example (good):

```
CREATE SYNONYM oe_products FOR oe.products;

DECLARE
   l_product_name oe_products.product_name%TYPE;
   co_price CONSTANT oe_products.list_price%TYPE := 1000;
BEGIN
   SELECT p.product_name
     INTO l_product_name
     FROM oe_products p
    WHERE list_price > co_price;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      NULL; -- handle_no_data_found;
   WHEN TOO_MANY_ROWS THEN
      NULL; -- handle_too_many_rows;
END;
/
```

4.8.3 Validating input parameter size

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

Changeability	Efficiency	**Maintainability**	Portability	
Reliability	**Reusability**	Security	**Testability**	
Blocker	Critical	Major	**Minor**	Info

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)
      RETURN departments%ROWTYPE IS
      l_return departments%rowtype;
   BEGIN
       IF    in_dept_name IS NULL
          OR LENGTH(in_dept_name) > 20
       THEN
          RAISE err.e_param_to_large;
       END IF;
       -- get the department by name
       SELECT *
         FROM departments
        WHERE department_name = in_dept_name;

       RETURN l_return;
   END dept_by_name;
END department_api;
/
```

Example (good):

```
CREATE OR REPLACE PACKAGE BODY department_api IS
   FUNCTION dept_by_name (in_dept_name IN departments.department_name%TYPE)
      RETURN departments%ROWTYPE IS
      l_dept_name departments.department_name%TYPE NOT NULL := in_dept_name;
      l_return departments%rowtype;
   BEGIN
       -- get the department by name
       SELECT *
         FROM departments
        WHERE department_name = l_dept_name;

       RETURN l_return;
   END dept_by_name;
END department_api;
/
```

Function call:

```
...
   r_deparment := department_api.dept_by_name('Far to long name of a department');
...
EXCEPTION
   WHEN VALUE_ERROR THEN ...
```

4.8.4 Ensure single execution at a time of a program unit

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

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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:

```
CREATE OR REPLACE PACKAGE BODY lock_up IS
   FUNCTION request_lock (in_lock_name         IN VARCHAR2
                         ,in_release_on_commit IN BOOLEAN := FALSE) RETURN VARCHAR2 IS
      l_lock_handle VARCHAR2(128 CHAR);
   BEGIN
      SYS.DBMS_LOCK.ALLOCATE_UNIQUE (lockname        => in_lock_name
                                    ,lockhandle      => l_lock_handle
                                    ,expiration_secs => constants_up.co_one_week);

      IF sys.DBMS_LOCK.REQUEST(lockhandle        => l_lock_handle
                              ,lockmode          => sys.DBMS_LOCK.X_MODE
                              ,TIMEOUT           => sys.DBMS_LOCK.MAXWAIT
                              ,release_on_commit => COALESCE(in_release_on_commit
                                                            ,FALSE)) > 0 THEN
         RAISE err.e_lock_request_failed;
      END IF;

      RETURN l_lock_handle;
   END request_lock;

   PROCEDURE release_lock (in_lock_handle IN VARCHAR2) IS
   BEGIN
     IF sys.DBMS_LOCK.RELEASE(lockhandle => in_lock_handle) > 0 THEN
         RAISE err.e_lock_request_failed;
     END IF;
   END release_lock;
END lock_up;
/
```

Call example:

```
DECLARE
   l_handle VARCHAR2(128 CHAR);
   co_lock_name CONSTANT VARCHAR2(30 CHAR) := 'APPLICATION_LOCK';
BEGIN
   l_handle := lock_up.request_lock(in_lock_name => co_lock_name);
   -- processing
   lock_up.release_lock(in_lock_handle => l_handle);
EXCEPTION
   WHEN OTHERS THEN
      -- log error
      lock_up.release_lock(in_lock_handle => l_handle);
      RAISE;
END;
/
```

4.8.5 Use dbms_application_info package to follow progress of a process

8420 Always use dbms_application_info to track program process transiently.

Changeability	**Efficiency**	Maintainability	Portability	
Reliability	Reusability	Security	Testability	
Blocker	Critical	Major	**Minor**	Info

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 trough V$SESSION view.

Example:

```
CREATE OR REPLACE PACKAGE BODY employee_api IS
   PROCEDURE process_emps IS
   BEGIN
      SYS.DBMS_APPLICATION_INFO.SET_MODULE(module_name => $$PLSQL_UNIT
                                          ,action_name => 'Init');
      <<employees>>
      FOR emp_rec IN (SELECT employee_id
                        FROM employees
                       ORDER BY employee_id)
      LOOP
         SYS.DBMS_APPLICATION_INFO.SET_ACTION('Processing ' || emp_rec.employee_id);
      END LOOP employees;
   end process_emps;
END employee_api;
/
```

5. 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.

5.1. Halstead Metric

5.1.1 Calculation

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

- n1 = the number of distinct operators
- n2 = the number of distinct operands
- N1 = the total number of operators
- N2 = the total number of operands

From these numbers, five measures can be calculated:

- Program length: $N = N1 + N2$
- Program vocabulary: $n = n1 + n2$
- Volume: $V = N \times \log_2 n$
- Difficulty : $D = \frac{n1}{2} \times \frac{N2}{n2}$
- Effort: $E = D \times V$

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 V describes the size of the implementation of an algorithm.

5.2. Cyclomatic Complexity (McCabe's)

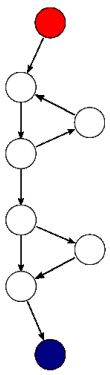
Cyclomatic complexity (or conditional complexity) is a software metric used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code.
Cyclomatic complexity is computed using the control flow graph of the program: the nodes of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

5.2.1 Description

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 IF statements or FOR loops, the complexity would be 1, since there is only a single path through the code. If the code has a single IF statement containing a single condition there would be two paths through the code, one path where the IF statement is evaluated as TRUE and one path where the IF statement is evaluated as FALSE.

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.



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
   FOR i IN 1..3
   LOOP
      dbms_output.put_line('in loop');
   END LOOP;
   --
   IF 1 = 1
   THEN
      dbms_output.put_line('yes');
   END IF;
   --
   dbms_output.put_line('end');
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 π is the number of decision points in the program, and s is the number of exit points.

6. 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.

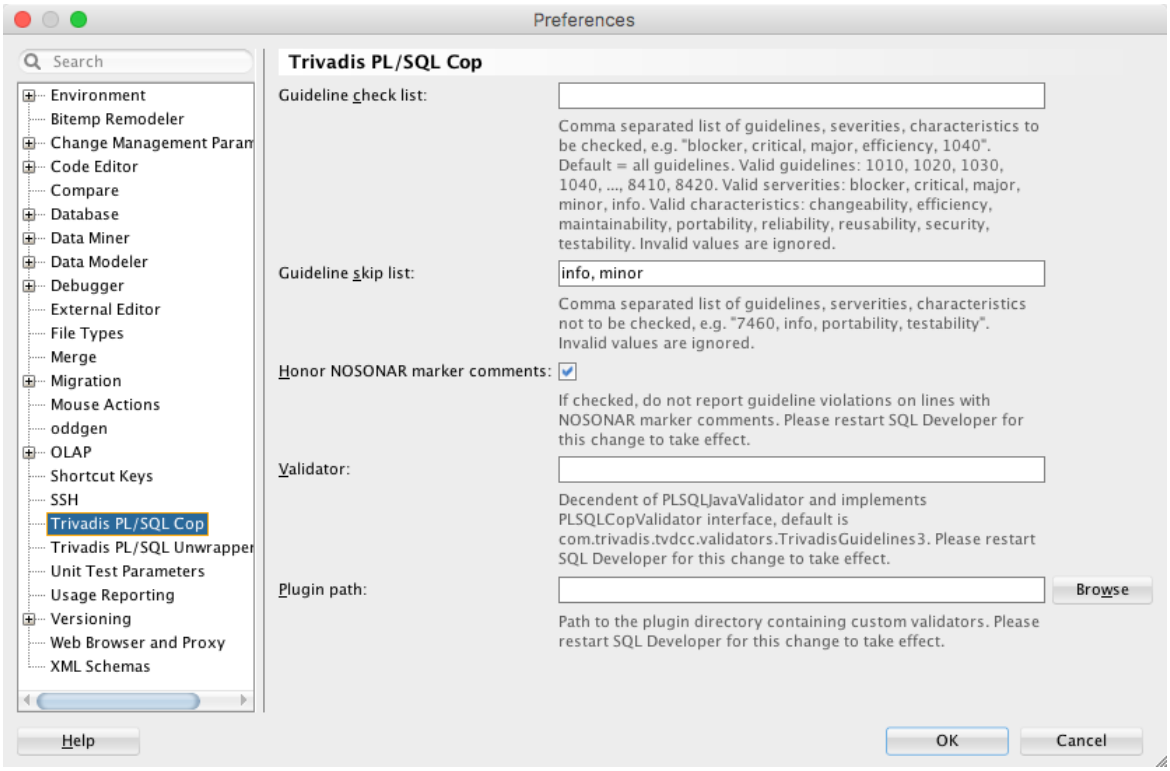
7. Tool Support

7.1. Development

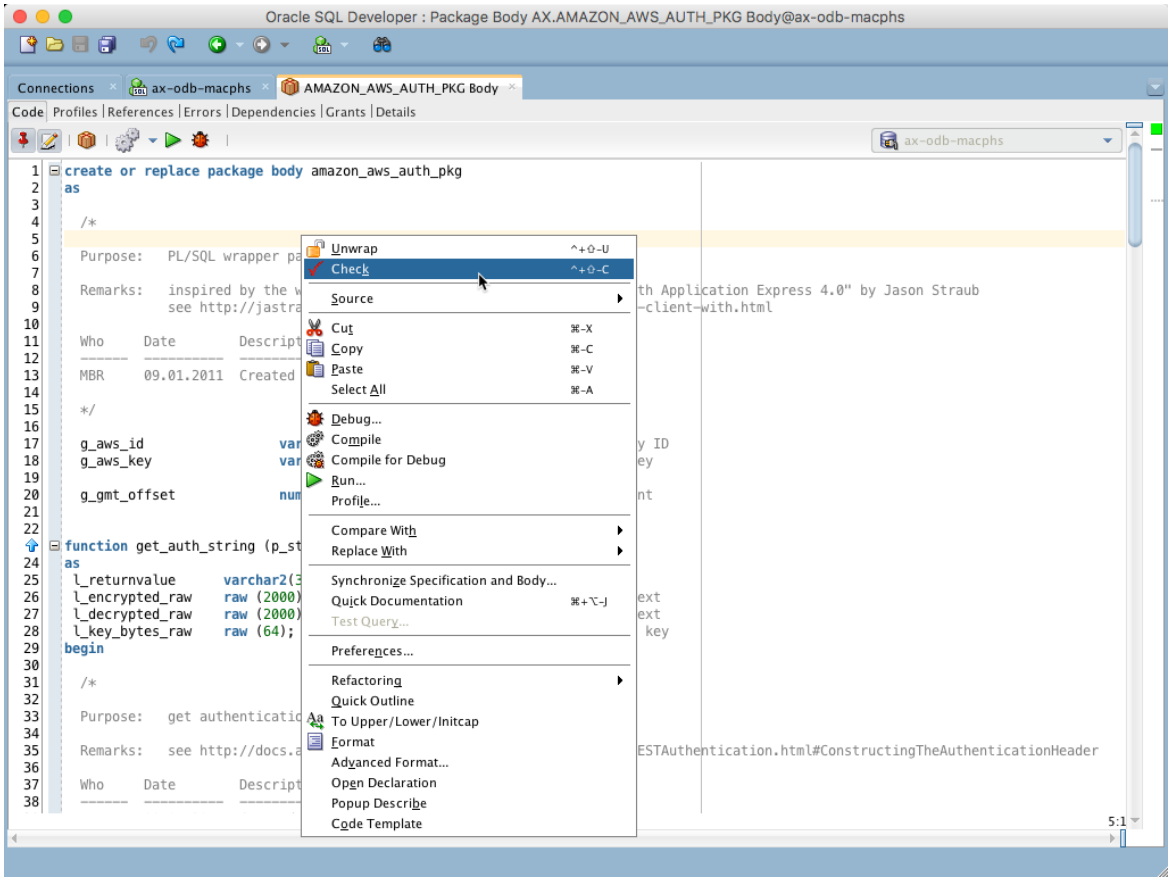
Trivadis offers a cost-free extension to ORACLE SQL Developer to test compliance with this coding guideline. The extension may be parameterized to your preferred set of rules and allows checking this set against a program unit.

7.1.1 Setting the preferences

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



7.1.2 Activate PLSQL Cop using context menu

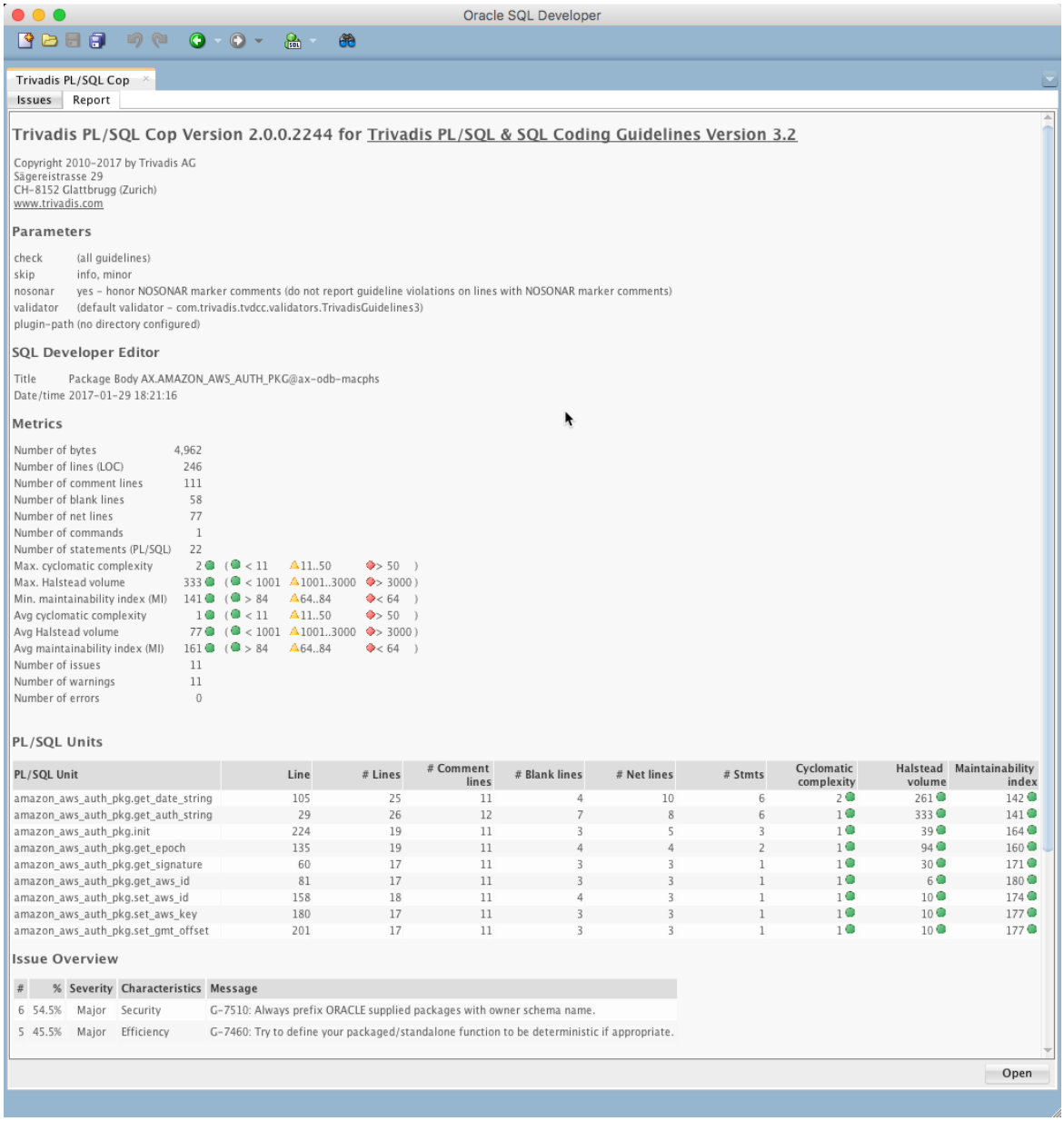


The result of the ckecking process is a list of violations with direct links to the place in the code as well as software metrics like:

- 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.

7.1.3 Software metrics



Appendix A – Mapping

Mapping new guidelines to prior versions

Old Id	New Id	Text	Severity	Changeability	Efficiency	Maintainability	Portability	Reliability	Reusability	Security	Testability
1	1010	Try to label your sub blocks.	Minor			×					
2	1020	Always have a matching loop or block label.	Minor			×					
3	1030	Avoid defining variables that are not used.	Minor		×	×					
4	1040	Avoid dead code.	Minor			×					
5	1050	Avoid using literals in your code.	Minor	×							
6	1060	Avoid storing ROWIDs or UROWIDs in database tables.	Major					×			
7	1070	Avoid nesting comment blocks.	Minor			×					
8	2110	Try to use anchored declarations for variables, constants and types.	Major			×		×			
9	2120	Try to have a single location to define your types.	Minor	×							
10	2130	Try to use subtypes for constructs used often in your code.	Minor	×							
11	2140	Never initialize variables with NULL.	Minor			×					
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			×					
16	2185	Avoid using overly short names for explicitly or implicitly declared identifiers.	Minor			×					
17	2190	Avoid the use of ROWID or UROWID.	Major				×	×			
18	2210	Avoid declaring NUMBER variables or subtypes with no precision.	Minor		×						
19	2220	Try to use PLS_INTEGER instead of NUMBER for arithmetic operations with integer values.	Minor		×						
n/a	2230	Try to use SIMPLE_INTEGER datatype when appropriate.	Minor		×						
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			×					
25	2510	Avoid using the LONG and LONG RAW data types.	Major				×				
26	3110	Always specify the target columns when coding an insert statement.	Major			×		×			
27	3120	Always use table aliases when your SQL statement involves more than one source.	Major			×					
28	3130	Try to use ANSI-join syntax.	Minor			×	×				
29	3140	Try to use anchored records as targets for your cursors.	Major			×		×			

Old Id	New Id	Text	Severity	Changeability	Efficiency	Maintainability	Portability	Reliability	Reusability	Security	Testability
n/a	3150	Try to use identity columns for surrogate keys.	Minor			×		×			
n/a	3160	Avoid virtual columns to be visible.	Major			×		×			
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	×				×			
n/a	3190	Avoid using NATURAL JOIN.	Major	×				×			
30	3210	Always use BULK OPERATIONS (BULK COLLECT, FORALL) whenever you have to execute a DML statement more than 4 times.	Major		×						
31	4110	Always use %NOTFOUND instead of NOT %FOUND to check whether a cursor returned data.	Minor			×					
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		×			×			
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			×					×
36	4220	Try to use CASE rather than DECODE.	Minor			×	×				
37	4230	Always use COALESCE instead of NVL, if parameter 2 of the NVL function is a function call or a SELECT statement.	Critical		×			×			
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		×			×			
39	4310	Never use GOTO statements in your code.	Major			×					×
40	4320	Always label your loops.	Minor			×					
41	4330	Always use a CURSOR FOR loop to process the complete cursor results unless you are using bulk operations.	Minor			×					
42	4340	Always use a NUMERIC FOR loop to process a dense array.	Minor			×					
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		×						
45	4370	Avoid using EXIT to stop loop processing unless you are in a basic loop.	Major			×					
46	4375	Always use EXIT WHEN instead of an IF statement to exit from a loop.	Minor			×					
47	4380	Try to label your EXIT WHEN statements.	Minor			×					
48	4385	Never use a cursor for loop to check whether a cursor returns data.	Major		×						
49	4390	Avoid use of unreferenced FOR loop indexes.	Major		×						
50	4395	Avoid hard-coded upper or lower bound values with FOR loops.	Minor	×		×					
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			×					

Old Id	New Id	Text	Severity	Changeability	Efficiency	Maintainability	Portability	Reliability	Reusability	Security	Testability
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	×		×					
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			×					×
59	6020	Try to use output bind arguments in the RETURNING INTO clause of dynamic DML statements rather than the USING clause.	Minor			×					
60	7110	Try to use named notation when calling program units.	Major	×		×					
61	7120	Always add the name of the program unit to its end keyword.	Minor			×					
62	7130	Always use parameters or pull in definitions rather than referencing external variables in a local program unit.	Major			×		×			×
63	7140	Always ensure that locally defined procedures or functions are referenced.	Major			×		×			
64	7150	Try to remove unused parameters.	Minor		×	×					
65	7210	Try to keep your packages small. Include only few procedures and functions that are used in the same context.	Minor		×	×					
66	7220	Always use forward declaration for private functions and procedures.	Minor	×							
67	7230	Avoid declaring global variables public.	Major					×			
68	7240	Avoid using an IN OUT parameter as IN or OUT only.	Major		×	×					
69	7310	Avoid standalone procedures – put your procedures in packages.	Minor			×					
70	7320	Avoid using RETURN statements in a PROCEDURE.	Major			×					×
71	7410	Avoid standalone functions – put your functions in packages.	Minor			×					
73	7420	Always make the RETURN statement the last statement of your function.	Major			×					
72	7430	Try to use no more than one RETURN statement within a function.	Major			×					×
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		×						
76	7510	Always prefix ORACLE supplied packages with owner schema name.	Major							×	
77	7710	Avoid cascading triggers.	Major			×					×
n/a	7810	Do not use SQL inside PL/SQL to read sequence numbers (or SYSDATE)	Major		×	×					
78	8110	Never use SELECT COUNT(*) if you are only	Major		×						

Old Id	New Id	Text	Severity	Changeability	Efficiency	Maintainability	Portability	Reliability	Reusability	Security	Testability
		interested in the existence of a row.									
n/a	8120	Never check existence of a row to decide whether to create it or not.	Major		×			×			
79	8210	Always use synonyms when accessing objects of another application schema.	Major	×		×					
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			×		×	×		×
n/a	8410	Always use application locks to ensure a program unit only running once at a given time.	Minor		×			×			
n/a	8420	Always use dbms_application_info to track program process transiently	Minor		×			×			