Estándar de Programación PL/SQL

Estándar de Programación PL/SQL

Versión	Descripción	Autor	Fecha	Revisado por
1.0.0	Original	Administrador de Base de Datos		

INDICE

Contenido

Introducción

El objetivo de este documento es definir un estándar de creación de objetos dentro del ambiente de bases de datos Oracle y no Oracle. Este documento definirá las directivas de nomenclatura y estructuración de los objetos de la base de datos basadas en buenas prácticas, las cuales se recomendarán dentro de todos los proyectos de desarrollo.

2.000 Alcance

Guía para el desarrollo en Base de datos Oracle y no Oracle (para los casos en que puedan aplicarse). Estándar base de creación de objetos de base de datos para proyectos externos que necesiten de una definición de estándares personalizados.

3.000 Buenas Prácticas

3.010 Nomenclatura

3.010.1 Nombre de la aplicación

a) Regla

Definir un nombre para la aplicación o módulo de aplicación y seleccionar una abreviatura de 3 caracteres para la misma. Utilizar esta abreviatura para nombrar los objetos que son propios de la aplicación o el módulo de la aplicación.

b) Motivación

La abreviatura del nombre de la aplicación puede ser utilizada en varios estándares de nombres para objetos del sistema. Dado que los nombres de objetos generalmente tienen un límite de tamaño, es útil disponer de un código corto para referirse a la aplicación.
Por otro lado, es útil disponer de una abreviatura para referirse al sistema en informes y documentación.

c) Ejemplo

```
Nombre del sistema: Sistema de Gestión de
Recursos Humanos Abreviatura: SRH
```

d) Beneficios

Permite distinguir fácilmente con el prefijo asignado a los objetos a que aplicación pertenece.

Permite la reutilización de nombres de un mismo concepto representado en un objeto de base de datos para aplicaciones distintas.
Desde este punto del documento en adelante nos referiremos a la abreviación de la aplicación como [APP]

3.010.2 Lenguaje de nomenclatura

a) Regla

Definir el lenguaje que se aplicará en el desarrollo de la aplicación completa para mantener el mismo en todos los objetos que pertenezcan a la aplicación.

b) Motivación

Prolijidad y consistencia sobre los nombres de los objetos.

c) Ejemplo

No usar combinación de lenguajes:

```
T_SRHM_TYPE_EMPLEADOS
```

Objetos de la misma aplicación con distintos lenguajes

```
T_SRH_EMPLEADOS
T_SRH_EMPLYEE_TYPE
```

d) Beneficios

Facilidad de interpretación de los conceptos.
Prolijidad.
Evitar la creación duplicada de objetos que representan el mismo concepto con distintos nombres.

Desafío

Determinar las palabras que puedan ser utilizadas en forma estandarizada en un lenguaje determinado. Por ejemplo en el caso de palabras de uso común que convengan mantener en inglés sin importar el lenguaje elegido, como por ejemplo la palabra “FLAG” como sufijo de columnas que representan estados.

3.010.3 Mayúscula y minúscula

a) Regla

Definir una forma de notación consistente de los nombres de los objetos teniendo en cuenta que los objetos en la base de datos NO SON CASE SENSITIVE, es decir, los nombres no se diferencian por el uso de mayúsculas y minúsculas.

b) Motivación

Prolijidad y legibilidad de código SQL y/o PL/SQL. Identificación simple de objetos de datos dentro de una sentencia de código.

c) Ejemplo

En el estándar de codificación PL/SQL se define que los identificadores de los objetos de base de datos se escriban todos en minúscula.

En documentos en los que se mezcla lenguaje común con nombres de objetos se recomienda resaltar los nombres de los objetos con mayúscula o entre comillas, no siendo así para el caso de código PL/SQL en que se recomienda escribir los nombres de los objetos en minúscula y las palabras reservadas en mayúscula.

Los nombres de los objetos se presentarán en mayúscula con el fin de resaltar los mismos dentro del lenguaje común de escritura.

3.010.4 Uso de Prefijos en identificadores de objetos de código PL/SQL

a) Regla

Utilizar prefijos para nombrar los identificadores de objetos de código PL/SQL.

b) Motivación

Diferenciar columnas de tablas de variables o estructuras de datos e identificación sencilla de distintos objetos de código.

c) Listado de Prefijos Estándares

Tipo de identificador		Prefijo
Constantes		c_
Tipos	Type y Subtype	t_
	Index by table	ti_
	Nested Table Collection	tn_
	Varray	tv_
	Ref Cursor	trc_
	Variables	v_
	Excepciones	e_
Parametros		P_ (para el caso de parámetros de cursores se puede utilizar pc_ con el fin de diferenciar los parámetros del modulo del cursor en el pasaje de parámetros)
Cursores		Cur_

3.010.5 Uso de Prefijos para objetos de base de datos

d) Regla

Utilizar prefijos para nomenclar los identificadores de objetos base de datos. Esta regla también se aplica para procedimientos y funciones de paquetes.

e) Motivación

Identificar los distintos tipos de objetos dentro del código PL/SQL, en los mensajes de errores estándares y en las tareas de mantenimiento de los mismos dentro del diccionario de datos de la base de datos y/o herramientas de soporte de desarrollo como los versionadores de código fuente.

f) Listado de Prefijos Estándares

Tipo de Objeto	Sufijo
Procedure	PRC_
Function	FN_
Package	PKG_
Sequence	SEQ_
Trigger	TRG_
View	V_
Materialized view	MV_

Table	T_

3.010.6 No utilizar artículos ni preposiciones

a) Regla

No deben utilizar preposiciones ni artículos dentro de los identificadores.

b) b) Ejemplo

"detalle de las facturas" debería abreviarse a "det_fact"

3.010.7 Sintaxis de identificadores

a) Regla

Para el caso de procedimientos utilizar frases verbales que representen la acción que llevará a cabo el procedimiento.
Para el caso de funciones utilizar sustantivos que describan el resultado de la función.
Para el caso de paquetes el nombre debe ser genérico que represente un concepto que englobe el conjunto de objetos a definirse dentro del mismo.
Para el caso de variables y tipos se debe especificar una descripción de lo que representa.

b) Motivación

Facilitar la identificación de tipo de módulo dentro del código y facilitar el entendimiento de la lógica mediante la utilización de identificadores similares al lenguaje natural.

c) Ejemplo

```
PROCEDURE prc_calcular_ventas_prod_empl(
         p_id_empleado IN
t_srh_empleados.id_empleado%TYPE,
         p_cod_producto      IN
t_com_productos.cod_producto%TYPE,
         p_fecha_desde IN     DATE,
         p_fecha_hasta IN     DATE,
         p_monto_total_venta OUT
t_com_resumen_ventas.monto_total%TYPE,
         p_cant_total_venta    OUT
t_com_resumen_ventas.cant_total%TYPE,
         );

FUNCTION fn_identificador_empleado(
         p_cod_empleado      IN
t_srh_empleados.cod_empleado%TYPE
         )
RETURN t_srh_empleados.id_empleado%TYPE;

FUNCTION fn_existe_empleado(
         p_cod_empleado  IN
t_srh_empleados.cod_empleado%TYPE
         )
RETURN BOOLEAN;
```

Aplicados dentro del código:

```
     …
     -- verificamos si se existe el empleado del código
     cargado
     IF fn_existe_empleado(v_codigo_empleado)
     THEN
          -- buscamos el identificador del empleado para el
proceso de calculo de venta
```

```
          v_id_empleado :=
fn_identificador_empleado(v_codigo_empleado);

         -- calculamos el total de venta para el producto para el
empleado
         prc_calcular_ventas_prod_empl(
                p_id_empleado => v_id_empleado,
                p_cod_producto      => v_cod_producto,
                p_fecha_desde => v_fecha_desde,
                p_fecha_hasta => v_fecha_hasta
                p_monto_total_venta => v_monto_total_venta,
                p_cant_total_venta     => v_cant_total_venta,
                );

END IF;
….
```

Con esta forma de nombrar los módulos los comentarios pasan a ser redundantes dentro del código.

d) Desafío

Definir un conjunto estándar de prefijos para los nombres para mantener una consistencia en la generación de estos identificadores. Por ejemplo en para el caso de los procedimientos que recuperan un valor utilizar siempre el prefijo “recuperar_”: recuperar_listado_empleado, recuperar_identif_producto, etc.

3.020 Tablas

3.020.1 Prefijo de nombre de tabla

a) Regla

A los nombres de tablas se agregarán los siguientes prefijos: tipo de objeto, de nombre de aplicación y tipo de tabla.

b) Motivación

Evitar conflictos de nombres con otras aplicaciones.

c) Ejemplo

La tabla de personas deberá llamarse T_SRHM_PERSONAS.

d) Beneficios

Permite la reutilización de un mismo nombre de objeto en distintas aplicaciones.
No se producen conflictos de nomenclatura al momento del uso de sinónimos públicos para la administración de seguridad.

e) Desventajas

Se extiende el nombre de las tablas por el prefijo.

En el caso de que exista la posibilidad de reutilización de componentes o módulos del sistema, se deberán cambiar los nombres de los objetos. Por ejemplo si se trata de una tabla general de logueo de errores, se debe cambiar el nombre general de la tabla para cada aplicación.

f) Prefijos de procesos

Ejemplos:

Direccion general de Electricidad

Prefijo	Descripción
CON	Concesiones Electricas
SER	Servidumbre
MAE	Tablas Maestras codificadoreas

Direccion general de Mineria

Prefijo	Descripción
PIM	Acreditacion de la producción y/o Inversión Mínima
DAC	Declaración Anua Consolidada
EMM	Estadística Mensual Minero metálica
EMN	Estadística Mensual Minero no metálica
PRY	Ficha de proyecto de inversión
INV	Inversiones Trimestrales

g) Prefijos de tipo de tabla

Prefijo	Descripción
M	Main (maestra del módulo)
D	Detalle
L	Lookup Table (codifocadoras)
J	Join (unión)
T	Temporary (temporales)

Ejemplo

T_GENM_EMPRESAS

Donde:

T	Prefijo de tipo de objeto
_	Carácter estándar de separación
GEN	Prefijo de proceso
M	Indicador del tipo de tabla
EMPRESAS	Nombre significativo de la tabla

3.020.2 Nombre en plural

a) Regla

Los nombres de las tablas serán en plural. La pluralidad se debe aplicar sobre el nombre del concepto base.

b) Ejemplo

T_SRHM_DIRECCIONES_EMPLEADO la pluralidad se aplica solo sobre direcciones que es el concepto base.

3.020.3 Utilización de abreviaturas en el nombre

a) Regla

Se priorizará la claridad de la interpretación del concepto que representa la tabla ante la abreviación, sin embargo en caso de que se trate de conceptos con composición de palabras, se deberán utilizar abreviaturas para los nombres de tablas, tratando de no abreviar el concepto base.
La lista de abreviaturas utilizadas en el sistema deberá formar parte de la documentación.

b) Ejemplo

La tabla direcciones de personas T_SRHM_DIRECCIONES_EMPL, donde la dirección es el concepto base.

c) Desventaja

Muchas abreviaturas no distinguirán entre plural y singular.

d) Desafío

Definir un conjunto basto de abreviaciones y principalmente que sean de conocimiento público y concensuadas con el resto del equipo.
Utilizar nombres y abreviaturas uniformes en todo el sistema. Es decir reutilizar las mismas convenciones de nomenclatura en todo el desarrollo.

3.020.4 Separación de palabras en el nombre

a) Regla

Las palabras serán separadas por “_” (underscore) UNICAMENTE. Los nombres de los objetos en la base de datos no son CASE SENSITIVE.

b) Ejemplo

Los nombres que utilizan mayúscula y minúscula para separar las palabras NO son validos: T_SRhTiposEmpleado. En su lugar deberia ser : T_SRHM_TIPOS_EMPLEADO

3.020.5 Longitud de nombre

a) Regla

Los nombres de tabla no deben exceder los 20 caracteres más la longitud del prefijo de aplicación.

b) Beneficios

La limitación de la longitud máxima obliga al uso de abreviaturas. Sin embargo no se debe abreviar demasiado.
Facilita la escritura de sentencias SQL y código PLSQL.

3.020.6 Alias para la tabla

a) Regla

Además del nombre, cada tabla deberá tener asignado un alias. El mismo será una abreviatura del nombre de a lo sumo 5 caracteres que no incluirá el prefijo de aplicación y sin separación de underscore.

b) Motivación

Los alias de tablas son utilizados en estándares de nombres de otros objetos de base de datos. Dado que los nombres de objetos generalmente tienen un límite de tamaño, es útil disponer de un código corto para referirse a la tabla.

c) Ejemplo

Para la tabla T_SRHM_EMPLEADOS se define el alias EMPL.

3.020.7 Creación de Comment

a) Regla

Crear comentarios (COMMENT) para todas las tablas con una breve descripción del concepto que representa la misma.

b) Motivación

El esquema posee en si misma documentación, proporcionando una explicación rápida de los conceptos del mismo.
Nota: Aplicar esta práctica también a nivel de columnas de las tablas.

3.030 Vistas

3.030.1 Nombre en plural

a) Regla

Los nombres de vistas serán en plural.

3.030.2 Prefijo de nombre de vista

a) Regla

Utilizar la siguiente convención para los nombres de vistas:
V_[app]_[nombre lógico]

b) Ejemplo

V_SRHM_EMPLEADOS

3.030.3 Uso del mismo nombre que la tabla

a) Regla

En caso de que la vista represente la estructura básica de una tabla con fines de seguridad en acceso utilizar el mismo nombre que la tabla, agregando el sufijo de vista.

b) Ejemplo

Para la tabla T_SRHM_EMPLEADOS se crea la vista V_SRHM_EMPLEADOS

3.040 Columnas (tablas y vistas)

3.040.1 Nombre en singular

a) Regla

Los nombres de columnas serán en singular. La excepción a la regla son las columnas multivaluadas (collections, varrays).

b) Motivación

Un nombre de columna en plural indica la posible necesidad de adición de columnas o la normalización de la columna en una tabla hijo.

3.040.2 Reutilización de Definición de Columnas

a) Regla

Los campos de distintas tablas que hacen referencia al mismo concepto general deben utilizar el mismo nombre, con idéntico tipo y longitud.

b) Motivación

Permite identificar el mismo concepto representado por la columna en todas las tablas que lo utilizan.

c) Ejemplo

Un ejemplo común es definir las columnas de auditoria y utilizarlas en todas las tablas:

```
USUARIO_CREACION VARCHAR2(30)
FECHA_CREACION DATE
USUARIO_ULTIMA_MOD VARCHAR2(30)
FECHA_ULTIMA_MOD DATE
```

Otro ejemplo podría ser por ejemplo una columna que representa FECHA_EMISION para las tablas REMITOS y FACTURAS.

d) Desafío

Conocer la existencia de los conceptos representado en las columnas del esquema que pueden ser utilizados en otras tablas. Mantener la consistencia de los tipos de datos y/o los nombres con respecto al concepto que representa.

3.040.3 Ausencia de prefijo de tabla

a) Regla

No utilizar prefijo de tabla para los nombres de columnas (a excepción del ID). El prefijo no es necesario dado que en queries o vistas, las columnas deben ser referenciadas mediante el alias de tabla (Ver estándar de PL/SQL). Por ejemplo:

```
SELECT   prs.nombre
FROM     t_srhm_personas prs;
```

En el caso en que en una vista (o consulta) surjan conflictos de nombres de columnas duplicadas, las columnas seleccionadas deberán tener un alias compuesto de un prefijo de tabla y el nombre de columna. Por ejemplo:

```
SELECT   prs.id prs_id,
dpt.id dpt_id
FROM     t_srhm_personas prs,
departamentos dpt
WHERE .....
;
```

3.040.4 Separación de palabras en el nombre

a) Regla

Las palabras serán separadas por "_" (underscore).

3.040.5 Longitud del nombre

a) Regla

Los nombres de columna no deben exceder los 20 caracteres.

3.040.6 Utilización de abreviaturas en el nombre

a) Regla

En caso de que sea posible, se deberán utilizar abreviaturas para los nombres de columnas.

b) Desafío

Definir un conjunto basto de abreviaciones y principalmente que sean de conocimiento público y concensuadas con el resto del equipo.

Utilizar nombres y abreviaturas uniformes en todo el sistema. Es decir reutilizar las mismas convenciones de nomenclatura en todo el desarrollo.

3.040.7 Identificadores internos de registro (claves primarias auto generadas)

a) Regla

Los Identificadores de registro interno se deberán llamar:

<nombre corto de la tabla>_“ID”.

b) Motivación

Identificar fácilmente estas columnas dentro del conjunto de columnas de la tabla y dentro de consultas.

c) Ventajas

Cuando estas columnas se deriven a otras tablas por la creación de Foreign Key se podrá identificar la tabla con la que se encuentra relacionado.

d) Desventajas

Cuando se trate de tablas con nombres conformado por varias palabras el nombre corto de la tabla puede no identificar fácilmente la tabla a la que pertenece la tabla o se deriva por FK.

3.040.8 Selección de Primary Keys

a) Regla

Las columnas de la primary key no deben tener semántica. Es preferible utilizar columnas que representen un identificador interno sin semántica.

b) Motivación

Se evitan claves primarias con muchas columnas como también columnas de FKs derivadas con muchas columnas.

Se evitan actualizaciones masivas en cascada de datos. (Pensar en una lista de teléfonos).

Si la clave primaria de una tabla fuera el número de teléfono, ¿que hubiera pasado cuando se agrego el 4 al número?). En muchos casos, la actualización de una columna parte de una clave primaria implica deshabilitar y habilitar foreign keys.

3.040.9 Foreign key columns

a) Regla

Utilizar el siguiente formato para nombre de columnas foreign key.

[alias de tabla padre]_[nombre de columna referenciada]

En el caso de que surjan conflictos (por ejemplo, cuando existe más de una foreign key a la misma tabla), se deberá utilizar el siguiente formato:

[Semántica de la relación]_[alias de tabla padre]_[nombre de columna referenciada]

Cuando una clave sea heredada con más de un nivel, no deberá anteponerse el alias de la tabla padre.

b) Ejemplo

Supongamos que existen tres tablas: t1, t2 y t3. La tabla t1 tiene una clave primaria compuesta por la columna id. La tabla t2 tiene una foreign key a t1 a través de una columna que deberá llamarse t1_id. Ahora, supongamos que t3 tiene una foreign key a la columna t1_id de la tabla t2. La columna deberá llamarse, entonces, t1_id (no deberá llamarse t2_t1_id).

3.040.10 Orden de columnas dentro de tablas

a) Regla

Las columnas deberán tener el siguiente orden dentro de las tablas:

1. Columnas pertenecientes a la clave primaria.
2. Columnas pertenecientes a claves únicas.
3. Columnas obligatorias
4. Columnas opcionales
5. Esta regla también deberá aplicarse a las vistas

b) Motivación

De esta forma, las columnas opcionales quedan al final de cada fila sin ocupar lugar en la base de datos.

Otra ventaja es que la estandarización del orden de las columnas hace que el esquema de datos sea más fácil de entender y verificar.
Cuando existan columnas que representan un valor y una columna de FK que referencia a una característica del valor, se deben poner juntas dentro del conjunto de columnas sin importar la obligatoriedad de las mismas.

c) Ejemplo

```
Tabla "T_VENTAS"
ID_VENTA
FECHA_VENTA
MONTO_VENTA
ID_MONEDA(una columna de FK a la tabla "monedas")
```

Las columnas "MONTO_VENTA" y "ID_MONEDA" se ubican juntas dado que "ID_MONEDA" especifica una característica del valor de la columna "MONTO_VENTA".

3.040.11 Tipos de datos de las columnas

a) Regla

No utilizar columnas tipo CHAR. Utilizar VARCHAR2.
Definir tamaño máximo de columnas NUMBER.

b) Motivación

Las columnas tipo CHAR pueden ahorrar espacio de almacenamiento en la base de datos, pero ocasionan muchos errores de programación dado que siempre son completados con espacios. Las columnas tipo NUMBER sin especificación de longitud, ocuparán 38 bytes.

3.040.12 Tipo de Dato en el nombre de columna

a) Regla

No utilizar prefijos que indiquen el tipo de dato de la columna como prefijo o sufijo (por ejemplo C_ CHAR, N_ NUMBER, etc).

3.050 Primary Keys

3.050.1 Nombre

a) Regla

Utilizar la siguiente nomenclatura para las primary keys:

PK_[APP]_[alias de tabla]

3.050.2 Orden de las columnas

a) Regla

Las columnas en la primary key deben tener el mismo orden que en la tabla.

Ejemplo

Para la tabla T_SRHM_PERSONAS el nombre de la clave primaria sería:
PK_SRHM_PERS

3.060 Unique Keys

3.060.1 Nombre

a) Regla

Utilizar la siguiente nomenclatura para las primary keys:

UK[numero de unique key] _[APP]_[alias de tabla]

b) Ejemplo

Para la tabla SRHM_PERSONAS el nombre de una de las claves únicas sería:
UK1_SRHM_PERS

3.060.2 Orden de las columnas

a) Regla

Las columnas en la unique key deben tener el mismo orden que en la tabla. Esto no es siempre posible, dado que una tabla puede tener varias unique Keys. Siempre que sea posible, esta regla debe respetarse.

3.070 Foreign Keys

3.070.1 Nombre

a) Regla

Utilizar la siguiente nomenclatura para las Foreign Keys:
FK_[APP]_[alias de tabla origen]_[alias de tabla referenciada]_[semántica de la relación]

b) Motivación

De esta forma es fácil determinar el tipo de constraint, las tablas involucradas y el significado de la relación. Además, la sección de "semántica de la relación" evitan problemas de unicidad en los nombres de las constraints cuando una tabla referencia a otras mas de una vez.

c) Ejemplo

Supongamos que la tabla T_SRHM_EMPLEADOS tiene una FK a la tabla T_SRH_AREAS que representa la relación "un empleado pertenece a una y solo una área de la empresa"

Nombre corto de la tabla T_SRHM_EMPLEADOS : "EMPL" , nombre corto de la tabla T_SRHM_AREAS: "AREA"

El nombre de la Foreign Key podría ser:

FK_SRHM_EMPL_AREA_PERTENECE.

Cuando la relación es muy obvia se puede omitir la semántica de la relación:
FK_SRHM_EMPL_AREA.

3.080 Check Constraints

3.080.1 Validación de reglas

a) Regla

Las check constraints no deberán validar más de una regla por vez.

b) Motivación

El mantenimiento de las reglas de negocio es más fácil de esta forma. Por ejemplo, cuando una regla de negocio se vuelve obsoleta, lo único que hay que hacer es eliminar la constraint que la implementa.

3.080.2 Nombre

a) Regla

Utilizar la siguiente nomenclatura para las Check constraints:
CK_[APP]_[alias de tabla]_[nombre lógico]

b) Motivación

De esta forma es más fácil distinguir de qué tipo de constraint se trata, la tabla a la que pertenece y el tipo de regla que implementa.

c) Ejemplo

Supongamos que la tabla T_SRHM_EMPLEADOS tiene una Check Constraint que verifica que los valores de la columna ESTADO esté dentro del conjuto de valores: [“A” Activo, “B” Dado de Baja], entonces el nombre de la constraint podría ser: CK_SRH_EMPL_ESTADO.

3.090 Sequencias

3.090.1 Nombre

a) Regla

Utilizar la siguiente nomenclatura para las Secuencias:

SEQ[numero de secuencia]_[APP]_[alias de tabla]

Donde el “número de secuencia” es un número correlativo comenzando desde 1 agregado al nombre completo de la secuencia que servirá para evitar problemas de unicidad en el caso de que una tabla utilice más de una secuencia.

Cuando se trate de la secuencia que genera los valores para el identificador interno de la tabla el nombre de la secuencia el nombre a asignar el siguiente:

SEQ_[APP]_[alias de tabla]_[alias identificador interno]

b) Motivación

De esta forma es fácil distinguir para qué tabla es utilizada la sequencia en cuestión y se evitan problemas de unicidad de nombres.

Para el caso de las secuencias de los identificadores internos se puede identificar fácilmente que secuencia corresponde al mismo.

c) Ejemplo

Si la columna del identificador interno de la tabla T_SRHM_EMPLEADOS es

ID_EMPLEADO la secuencia que genera los valores para esta columna podría llamarse T_SRHM_EMPL_ID.

3.0100 Indices

3.0100.1 Pautas de Indización

a) Regla

A la hora de crear un índice se deben tener en cuenta las pautas de indización:

- Crear índices solo cuando fuese necesario.
- Crear un índice para acelerar una consulta puede afectar a otras.
- Eliminar índices innecesarios.
- Evaluar el plan de ejecución a fin de corroborar la utilización de los índices por el optimizador.
- Las claves de los índices deben tener una alta selectividad. La selectividad de un índice esta dada por la cantidad de distintas claves, sobre el total de tuplas que posee la tabla. Una buena selectividad es lograda cuando son pocas las claves que tienen un mismo valor.
- Considere indizar claves utilizadas frecuentemente en cláusulas WHERE.
- Considere indizar columnas utilizadas frecuentemente en join's de tablas.
- No utilice índices B*-tree con claves o expresiones con pocos valores distintos ya que estos tienen baja selectividad y no mejoran la performance.
- No indizar columnas que son frecuentemente modificadas. Update / Insert y delete's que modifican tablas indexadas, demoran mas que tablas no indizadas.
- No indizar claves que aparezcan en una cláusula WHERE utilizando funciones. Una cláusula WHERE que utilice funciones que no sean MIN o MAX, no tendrán disponible el índice, excepto que el índice, sea basado en funciones.

b) Beneficios

Bien utilizados mejoran enormemente el rendimiento en consultas frecuentes.

c) Desafíos

Se deben tener en cuenta los efectos de utilizar índices en tablas cuando se realizan operaciones DML. Dichos efectos se listan a continuación:

Luego de periodos de alta actividad de DMLs, se deben reorganizar los índices B*-tree. Estos índices reconstruidos sobre datos existentes son más eficientes que los mantenidos implícitamente por el Oracle Server.

Los inserts sobre la tablas, genera la inserción de una entrada en el bloque correspondiente del índice. Pudiéndose generar un (block split).

Los deletes sobre tablas, genera la eliminación de una entrada en el bloque correspondiente del índice. Pudiéndose generar la liberación del bloque.

Los updates sobre tablas, genera la eliminación de la vieja entrada y la inserción de una nueva entrada en el bloque correspondiente del índice.

3.0100.2 Nombre de índices

a) Regla

Los índices creados para las foreign keys deberán tener el mismo nombre que la constraint con el prefijo “I_”

Los demás índices deberán utilizar la siguiente nomenclatura:

I[numero de secuencia_][APP]_[alias de tabla]

b) Ejemplo

Supongamos que la tabla T_SRHM_PERSONAS posee la columna NUMERO_DOCUMENTO_UNICO el nombre del índice podria ser: I01_SRH_PERS.

Supongamos que la tabla T_SRHM_EMPLEADOS posee la FK FK_SRH_EMPL_AREA_PERTENECE, entonces el índice de soporte de la FK podria llamarse FK_I_SRH_EMPL_AREA_PERTENECE.

3.0100.3 Indices y valores Null

a) Regla

Los valores NULL en índices se consideran distintos excepto cuando todos los valores not-null en dos o más filas de un índice son idénticos, en ese caso las filas se consideran idénticas.

Los índices Unique evitan que las filas que contienen valores NULL sean tratadas como idénticas. Oracle no pone en un índice las filas de la tabla en las cuales todas las columnas de la clave son NULL.

3.0100.4 Creación de índices para foreign keys

a) Regla

Crear un índice para todas las foreign keys de las tablas salvo para los siquientes casos:

- El diseñador determina que el índice no mejorará la performance de las queries.
- El diseñador determina que el índice ocasionará un alto overhead.
- Existe otro indice que incluye las columnas de la foreign key.

b) Motivación

El motor de base de datos, no crea automáticamente índices para claves foráneas.

Las columnas de una FK son utilizadas frecuentemente para Joinearlas con las tablas de referencia. La existencia de índices sobre estas columnas, teniendo en cuenta las excepciones antes mencionadas, mejora la performance de las queries en las que se Joinean estas tablas.

3.0100.5 No crear índices únicos

a) Regla

No deberán crearse índices únicos, para eso se deberán utilizarse las unique keys.

b) Motivación

El hecho de que se representen los casos de valores únicos con Unique Keys aporta semántica al significado de que un conjunto de columnas, esto se hace evidente cuando se visualiza un error de valores duplicados referenciando a una UK y no a un índice.

3.0100.6 Orden de columnas de índices para FKs

a) Regla

Para índices creados para constraints, utilizar el mismo orden secuencial de las columnas de la constraint.

b) Motivación

Esto previene la confusión y facilita la lectura de la semántica de las relaciones en las queries.

3.0100.7 Pautas para índices compuestos

a) Regla

- Coloque primero (“leading part”) las columnas de mayor selectividad.
- Coloque primero las columnas accedidas con mayor frecuencia.
- Si especificara la clave entera, coloque primero la columna más restrictiva.
- Utilice la opción COMPRESS cuando sea posible.

3.0100.8 Índices basados en funciones

a) Regla

Condiciones para utilizar índices basados en funciones:
La función debe ser determinística. Es decir, que dado y=f(x) para el mismo x, siempre devuelve el mismo y.

Ejemplo función NO deterministica: f(x) = to_char(sysdate,’ddmmyyyy’)||x
La sesión debe tener seteado el parámetro Query_rewrite_enabled=true.

b) Motivación

Los índices basados en funciones proporcionan un mecanismo eficiente para las consultas que contienen funciones en su cláusula WHERE.

Los índices basados en funciones precomputa el valor de la función o de la expresión y lo almacena en el índice. Se puede crear un índice basado en función como un b-tree o un índice Bitmap.

c) Ejemplo:

```
CREATE INDEX FBI_UPPER_LASTNAME ON
T_GENM_CLIENTES(upper(cust_last_name));
```

3.0100.9 Índices y Constraints

a) Regla

Cuando se define una primary o unique key constraint sobre una tabla, el motor genera automáticamente un índice (o utiliza uno existente) para soportarlo.

No se puede eliminar índices que se utilizan para cumplir con alguna constraint. Para hacer esto, se debe dropear primero la contraint y luego el índice.

Los índices responden por lo menos a dos propósitos:

Para realizar consultas más performantes

Hacer cumplir claves únicas y primarias.

3.0100.10 Índices de clave invertida

a) Regla

Crear un índice de clave invertida, comparado a un índice estándar, invierte los bytes de cada columna puesta en un índice (excepto el rowid) mientras que guarda el orden de la columna. Tal arreglo puede ayudar a evitar la contencion en disco cuando las modificaciones al índice se concentran en bloques de la última hoja.

Invirtiendo la clave del índice, las inserciones se distribuyen a través de todas las hojas del índice.

Utilizar un índice con clave invertida elimina la posibilidad de hacer un range scan sobre el mismo. Porque las claves lógicamente adyacentes no se almacenan de forma contigua. Sólo se pueden realizar consultas por clave o accesos full.

3.0100.11 Pautas para índices Bitmap

a) Regla

Usarlos en columnas con baja cardinalidad, es decir el numero de valores distintos es pequeño comparado con el nro de filas de la tabla. Ejemplo, Sexo, Estados, Tipos, etc.

Ocupan menos espacio y son mas eficientes que los índices B*Tree cuando la columna es de baja cardinalidad.

Los índices bitmap no son convenientes para los usos de OLTP con una gran cantidad de transacciones concurrentes que modifican los datos. Estos índices son performantes para DDS donde los usuarios consultan los datos con pocas actualizaciones diarias.

Este tipo de índices tienen tendencia a desbalancearse. Por lo tanto cuando se realizan grandes movimientos de datos es recomendable reconstruirlos.

En procesos batch que implique movimiento de muchos datos es recomendable hacer lo siguiente:

Deshabilitar Indices Bitmap
Ejecutar procesos Batch
Reconstruir indices Bitmap (rebuild Index)

b) Motivación

Oracle almacena cada valor de clave con cada rowid almacenado.

Cada bit en el BITMAP corresponde a un rowid posible. Si se fija el bit, entonces significa que la fila con el rowid correspondiente contiene la misma clave. Una función convierte la posición del bit a un rowid real, así que el índice Bitmap proporciona la misma funcionalidad que un índice regular aunque es distinto internamente.

c) Ejemplo

```
SELECT *  FROM   t_genm_products   WHERE  supplier_id = 3;
```

Suplier Id = 3	
0	
0	
1	Fila Retornada
0	
1	Fila Retornada

3.0100.12 Índices particionados.

a) Regla

Local index

En un local index, todas las claves en una partición particular del índice se refieren exclusivamente a las filas almacenadas en una sola partición de la tabla. Este tipo de índice tiene en algunas condiciones, mejores tiempos de respuesta y permite un mejor y mas rápido mantenimiento.

Local prefixed
Un local index es prefixed si las columnas de particionamiento de la tabla se corresponden con las primeras del índice.
Local nonprefixed

Un local index es non-prefixed si las columnas de particionamiento de la tabla no se encuentran a la izquierda del índice.

Global prefixed index (Oracle no suporta global non-prefixed indexes)

En un global index, las claves de una partición particular del índice pueden referir a las filas almacenadas en más de un partición o subpartition de la tabla.

3.0110 Triggers

3.0110.1 Nombre

a) Regla

Los nombres de los triggers se conformarán según la instancia en que se disparan, el evento que los disparan y el alcance de las sentencias:

Instancia
After: AFT
Before: BEF

Evento
Insert: INS
Update: UPD

Delete: DEL

Alcance
Each Row: RW
Statement: SMT

Entonces la conformación del nombre sería:

TRG_[APP]_[alias de tabla]_[instancia]_[evento]_[alcance]

b) Motivación

Poder determinar a partir del nombre del trigger la tabla a la que pertenece, la instancia en que se dispara, el evento que lo dispara y el alcance de las operaciones dentro del trigger.

c) Ejemplo

TRG_SRH_PERS_BEF_INS_RW: trigger de la tabla T_SRHM_PERSONAS de la aplicación SRH que se dispara antes de una operación INSERT sobre la tabla por cada registro.

TRG_SRH_EMPL_AFT_DEL_SMT: trigger de la tabla T_SRHM_EMPLEADOS de la aplicación SRH que se dispara después de una operación DELETE sobre la tabla por sentencia.

3.0110.2 Lógica simple

a) Regla

No cargar el trigger con lógica compleja.

b) Motivación

No sumar un ovesrhead considerable a la operación que lo dispara.

3.0120 Excepciones

3.0120.1 Utilizar excepciones

a) Regla

Para emitir errores las rutinas deben utilizar excepciones en lugar de códigos de éxito/fracaso.

b) Motivación

Las excepciones son una característica que no todos los lenguajes tienen. No utilizarla hace que los programas sean más difíciles de codificar y mantener pues la lógica de negocios suele mezclarse con la del control de errores.

c) Ejemplo

Ejemplo de lo que no se debe hacer

```
PROCEDURE prc_test_procedure(x NUMBER, y NUMBER,
errcode NUMBER) IS
BEGIN
 ...
 IF ... THEN
  errorcode = 0;
 ELSIF ... THEN
  errorcode = 1;
 ELSIF ... THEN
  errorcode = N;
 END IF;
END;
```

```
BEGIN
 ...
 test_procedure(x, y, errcode);
 --
 IF errorcode = 0 THEN
  ... continuar ...
 ELSE
  ...
 END IF;
END;
```

Ejemplo utilizando excepciones

```
PROCEDURE prc_test_procedure(x NUMBER, y NUMBER,
errcode NUMBER) IS
BEGIN
 ...
 RAISE(exception1_cod);
 ...
RAISE(exception2_cod);
END;

BEGIN
 ...
 test_procedure(x, y);
 ... continuar ...
EXCEPTION
 WHEN xxx THEN
  ...
END;
```

d) Beneficios

El código generado es más mantenible, menos propenso a errores y menos costoso.

e) Desafíos

Esto significará tener una disciplina y una planificación avanzada. Antes de comenzar a construir su aplicación, crear un conjunto de packages que definan las reglas de negocios y fórmulas para las distintas áreas de funcionalidad.

3.0120.2 Utilizar el package “err_msg” de excepciones

a) Regla

Todas las excepciones deben ser emitidas con un package construido para tal fin llamado err_mgr. No debe utilizarse RAISE_APPLICATION_ERROR.

b) Motivación

Estandariza la emisión de excepciones y hace que los códigos y mensajes de error sean estándares en el sistema, pues los mismos son obtenidos de una tabla. Además, los códigos de error representan una situación puntual conocida y documentada para el sistema.

c) Ejemplo

```
CREATE OR REPLACE PACKAGE pkg_err_mgr IS
    PROCEDURE raise_with_msg(p_exc_cod PLS_INTEGER);
    FUNCTION errcode RETURN PLS_INTEGER;
END;
/

CREATE OR REPLACE PACKAGE BODY pkg_err_mgr IS
    v_last_errcode PLS_INTEGER;

    PROCEDURE prc_raise_with_msg(p_exc_cod PLS_INTEGER) IS
        v_pls_exception_code NUMBER;
        v_err_msg VARCHAR2(2000);
    BEGIN
        --
        v_last_errcode := p_exc_cod;
        raise_application_error(
            v_pls_exception_code,
            v_err_msg,
            TRUE);
    END;

    FUNCTION fn_errcode RETURN PLS_INTEGER IS
    BEGIN
        return v_last_errcode;
    END;
END;
/
```

```
CREATE OR REPLACE PACKAGE pkg_exc_test3 IS
  exception1_cod CONSTANT PLS_INTEGER := 1001;
  e_test3 exception;
  pragma exception_init(e_test3, -20001);

  PROCEDURE proc1;
  PROCEDURE endproc1;
END;
/
  CREATE OR REPLACE PACKAGE BODY pck_exc_test3 IS
    PROCEDURE proc1 IS
    BEGIN
        -- Logica....
        -- ...
        -- Ocurrió un caso de excepción...
        err_mgr.raise_with_msg(exception1_cod);
```

```
        -- Logica ...
        -- ...
    END;
```

```
    PROCEDURE endproc1 IS
        v_errm VARCHAR2(1000);
    BEGIN
        -- Esto puede ser un job, por ejemplo,
        -- que recorre una lista y por cada uno ejecuta la
           lógica de proc1
        -- FOR I IN CURSOR XXX LOOP
            proc1;
        -- END LOOP;
    EXCEPTION
        WHEN e_test3 THEN
            IF err_mgr.errcode = exception1_cod THEN
                -- Manejar la exception... en
                  este ejemplo solo vuelvo a emitirla
                dbms_output.put_line('Handling ' ||
to_char(exception1_cod));
                RAISE;
            --ELSIF errcode = exception2_cod
            -- ... HANDLER EXCEPTION2
            --..
            END IF;
    END;
END;
/
```

3.0120.3 Una excepción debe representar un único error

a) Regla

Cada excepción debe identificar a un solo error.

b) Motivación

Mayor claridad en el manejo de errores e identificación de problemas. Por ejemplo, el mensaje de error “Persona inexistente o tipo de documento inválido” debería sustituirse por dos errores: “Persona inexistente” y “Tipo de documento inválido”.

3.0120.4 Manejar excepciones que puedan ser anticipadas

a) Regla

Ante la llamada a un procedimiento, la rutina llamadora debe manejar las posibles excepciones emitidas por el procedimiento llamado. Para esto, los procedimientos deben tener documentadas las excepciones que pueden emitir.

b) Motivación

Evitar terminaciones no esperadas de programas. Corrección.

3.0120.5 No hardcodear códigos de error

a) Regla

No hardcodear códigos de error. Se debe utilizar la tabla de errores de la aplicación. Por cada excepción declarada debe existir una constante que la identifique. El nombre de la constante debe ser de la siguiente forma: “nombreExcepcion_cod”.

b) Motivación

Diseñar los programas de la manera que estos tengan un propósito claro y bien definido.

c) Ejemplo

```
CREATE OR REPLACE PACKAGE pck_err_mgr IS
  PROCEDURE raise_with_msg(p_exc_cod PLS_INTEGER);
  FUNCTION errcode RETURN PLS_INTEGER;
END;
/
```

```
CREATE OR REPLACE PACKAGE BODY pck_err_mgr IS
    v_last_errcode PLS_INTEGER;

    PROCEDURE raise_with_msg(p_exc_cod PLS_INTEGER) IS
        v_pls_exception_code NUMBER;
        v_err_msg VARCHAR2(2000);
    BEGIN
        ---
        v_last_errcode := p_exc_cod;
        raise_application_error(
            v_pls_exception_code,
            v_err_msg,
            TRUE);
    END;

    FUNCTION errcode RETURN PLS_INTEGER IS
    BEGIN
        return v_last_errcode;
    END;
END;
/
```

```
CREATE OR REPLACE PACKAGE pck_exc_test3 IS
  exception1_cod CONSTANT PLS_INTEGER := 1001;
  e_test3 exception;
  pragma exception_init(e_test3, -20001);

  PROCEDURE proc1;
  PROCEDURE endproc1;
END;
/

CREATE OR REPLACE PACKAGE BODY pck_exc_test3 IS
    PROCEDURE proc1 IS
    BEGIN
        -- Logica....
        -- ...
        -- Ocurrio un caso de excepcion...
        err_mgr.raise_with_msg(exception1_cod);

        -- Logica ...
        -- ...
    END;

    PROCEDURE endproc1 IS
        v_errm VARCHAR2(1000);
    BEGIN
        -- Esto puede ser un job, por ejemplo,
        -- que recorre una lista y por cada uno
ejecuta la logica de proc1
        -- FOR I IN CURSOR XXX LOOP
            proc1;
        -- END LOOP;
    EXCEPTION
        WHEN e_test3 THEN
            IF err_mgr.errcode = exception1_cod THEN
                -- Manejar la exception... en este
ejemplo solo vuelvo a emitirla
                dbms_output.put_line('Handling ' ||
to_char(exception1_cod));
                RAISE;
            --ELSIF errcode = exception2_cod
            -- ... HANDLER EXCEPTION2
            --..
            END IF;
    END;
END;
/
```

3.0120.6 Captura de excepciones

a) Regla

Capturar excepciones por nombre. No utilizar WHEN OTHERS.
La sección de manejo de excepciones debe capturar la excepción emitida por nombre y tener una cláusula CASE para discriminar el código de error de la excepción emitida.

b) Motivación

Capturar de esta forma los errores es una buena forma de emular que todas las excepciones sean nombradas.

c) Ejemplo

```
CREATE OR REPLACE PACKAGE pck_err_mgr IS
  PROCEDURE raise_with_msg(p_exc_cod PLS_INTEGER);
  FUNCTION errcode RETURN PLS_INTEGER;
END;
/

CREATE OR REPLACE PACKAGE BODY pck_err_mgr IS
    v_last_errcode PLS_INTEGER;

    PROCEDURE raise_with_msg(p_exc_cod PLS_INTEGER) IS
        v_pls_exception_code NUMBER;
        v_err_msg VARCHAR2(2000);
    BEGIN
        --
        v_last_errcode := p_exc_cod;
        raise_application_error(
            v_pls_exception_code,
            v_err_msg,
            TRUE);
    END;
```

```
    FUNCTION errcode RETURN PLS_INTEGER IS
    BEGIN
        return v_last_errcode;
    END;
END;
/
```

```
CREATE OR REPLACE PACKAGE pck_exc_test3 IS
  exception1_cod CONSTANT PLS_INTEGER := 1001;
  e_test3 exception;
  pragma exception_init(e_test3, -20001);

  PROCEDURE proc1;
  PROCEDURE endproc1;
END;
/
```

```
CREATE OR REPLACE PACKAGE BODY pck_exc_test3 IS
    PROCEDURE proc1 IS
    BEGIN
        -- Logica....
        -- ...
        -- Ocurrio un caso de excepcion...
        err_mgr.raise_with_msg(exception1_cod);

        -- Logica ...
        -- ...
    END;
```

```
PROCEDURE endproc1 IS
      v_errm VARCHAR2(1000);
  BEGIN
        -- Esto puede ser un job, por ejemplo,
        -- que recorre una lista y por cada uno
ejecuta la logica de proc1
        -- FOR I IN CURSOR XXX LOOP
            proc1;
        -- END LOOP;
    EXCEPTION
        WHEN e_test3 THEN
            IF err_mgr.errcode = exception1_cod THEN
                -- Manejar la exception... en este
ejemplo solo vuelvo a emitirla
                dbms_output.put_line('Handling ' ||
 to_char(exception1_cod));
                RAISE;

--ELSIF errcode = exception2_cod
            -- ... HANDLER EXCEPTION2
            --..
            END IF;
    END;
END;
/
```

3.0120.7 Emular excepciones nombradas por constantes en packages

a) Regla

Las excepciones deben declararse como constantes en el PACKAGE que las emite o bien en otro package que contenga todas las excepciones del package principal.

Cada constante que represente un error deberá tener a su vez una excepción (excepción PLSQL propiamente dicha) asociada.

b) Motivación

Dado que PL/SQL sólo reserva 1000 códigos de error para usuarios (desde el -20000 al -21000) se determina que las excepciones se representarán con constantes pues 1000 códigos son pocos. Por otro lado, asociar excepciones PLSQL a cada código de error permite que la captura se realice por nombre en lugar de utilizar WHEN OTHERS.

Esta forma de manejar excepciones emula la forma en que el lenguaje lo hace naturalmente.

c) Ejemplo

```
CREATE OR REPLACE PACKAGE pck_err_mgr IS
  PROCEDURE raise_with_msg(p_exc_cod PLS_INTEGER);
  FUNCTION errcode RETURN PLS_INTEGER;
END;
```

```
CREATE OR REPLACE PACKAGE BODY pck_err_mgr IS
    v_last_errcode PLS_INTEGER;
    PROCEDURE raise_with_msg(p_exc_cod PLS_INTEGER) IS
        v_pls_exception_code NUMBER;
        v_err_msg VARCHAR2(2000);
    BEGIN
        /*
        SELECT
        INTO  v_pls_exception_code,
                    v_err_msg
        FROM errtab
        WHERE exc_cod = p_exc_cod;
        */
        --
        v_last_errcode := p_exc_cod;
        raise_application_error(
            v_pls_exception_code,
            v_err_msg,
            TRUE);
    END;
```

```
    FUNCTION errcode RETURN PLS_INTEGER IS
    BEGIN
        return v_last_errcode;
    END;
END;
/

CREATE OR REPLACE PACKAGE pck_exc_test3 IS
  exception1_cod CONSTANT PLS_INTEGER := 1001;
  e_test3 exception;
  pragma exception_init(e_test3, -20001);

  PROCEDURE proc1;
  PROCEDURE endproc1;
END;
/
```

```
CREATE OR REPLACE PACKAGE BODY pck_exc_test3 IS
    PROCEDURE proc1 IS
    BEGIN
        -- Logica....
        -- ...
        -- Ocurrio un caso de excepcion...
        err_mgr.raise_with_msg(exception1_cod);

        -- Logica ...
        -- ...
    END;

    PROCEDURE endproc1 IS
        v_errm VARCHAR2(1000);
    BEGIN
        -- Esto puede ser un job, por ejemplo,
        -- que recorre una lista y por cada uno ejecuta la lógica
```

```
 de proc1
        -- FOR I IN CURSOR XXX LOOP
            proc1;
        -- END LOOP;
    EXCEPTION
        WHEN e_test3 THEN
            IF err_mgr.errcode = exception1_cod THEN
                -- Manejar la exception... en este ejemplo solo
 vuelvo a emitirla
                dbms_output.put_line('Handling ' ||
to_char(exception1_cod));
                RAISE;
            --ELSIF errcode = exception2_cod
            -- ... HANDLER EXCEPTION2
            --..
            END IF;
    END;
END;
/
```

3.0120.8 Las rutinas deben documentar las excepciones

a) Regla

Para cada procedimiento o función dentro de un package se deben agregar como comentarios en la especificación del package cuales son las excepciones que emite.

b) Ejemplo

```
PACKAGE pck_exc_test3 IS
  e_test3 exception;
  pragma exception_init(e_test3, -20001);
  exception1_cod CONSTANT PLS_INTEGER := 1001;
  exception2_cod CONSTANT PLS_INTEGER := 1002;

  -- Excepciones que emite:
  --  exception1_cod (e_test3): Se emite cuando....
  --  exception2_cod (e_test3): Se emite cuando....
  PROCEDURE proc1;
END;
```

3.0120.9 Utilizar PRAGMA_EXCEPTION INIT

a) Regla

Asignar cada excepción definida por el usuario a un código de error Oracle mediante PRAGMA EXCEPTION_INIT. El código deberá ser único en el sistema.

b) Ejemplo

```
CREATE OR REPLACE PACKAGE pck_err_mgr IS
  PROCEDURE raise_with_msg(p_exc_cod PLS_INTEGER);
  FUNCTION errcode RETURN PLS_INTEGER;
END;
/
```

```
CREATE OR REPLACE PACKAGE BODY pck_err_mgr IS
    v_last_errcode PLS_INTEGER;

    PROCEDURE raise_with_msg(p_exc_cod PLS_INTEGER) IS
        v_pls_exception_code NUMBER;
        v_err_msg VARCHAR2(2000);
    BEGIN
        /*
        SELECT
        INTO
            v_pls_exception_code,
            v_err_msg
        FROM
            errtab
        WHERE
            exc_cod = p_exc_cod
        ;
        */
        --
        v_last_errcode := p_exc_cod;
        raise_application_error(
            v_pls_exception_code,
            v_err_msg,
            TRUE);
    END;
```

```
    FUNCTION errcode RETURN PLS_INTEGER IS
    BEGIN
        return v_last_errcode;
    END;
END;
/
```

```
CREATE OR REPLACE PACKAGE pck_exc_test3 IS
  exception1_cod CONSTANT PLS_INTEGER := 1001;
  e_test3 exception;
  pragma exception_init(e_test3, -20001);

  PROCEDURE proc1;
  PROCEDURE endproc1;
END;
/
```

```
   CREATE OR REPLACE PACKAGE BODY pck_exc_test3 IS

       PROCEDURE proc1 IS
       BEGIN
           -- Logica....
           -- ...
           -- Ocurrio un caso de excepcion...
           err_mgr.raise_with_msg(exception1_cod);

           -- Logica ...         -- ...
       END;

       PROCEDURE endproc1 IS
           v_errm VARCHAR2(1000);
       BEGIN
           -- Esto puede ser un job, por ejemplo,
           -- que recorre una lista y por cada uno ejecuta la logica
de proc1
           -- FOR I IN CURSOR XXX LOOP
               proc1;
           -- END LOOP;
        EXCEPTION
            WHEN e_test3 THEN
                IF err_mgr.errcode = exception1_cod THEN
                    -- Manejar la exception... en este ejemplo solo
vuelvo a emitirla
                    dbms_output.put_line('Handling ' ||
to_char(exception1_cod));
                    RAISE;
                --ELSIF errcode = exception2_cod
                -- ... HANDLER EXCEPTION2
                --..
                END IF;
        END;
    END;
    /
```

4.000 Estilo de Programación

4.010 Generalidades

4.010.1 Mayúscula y minúscula

a) Regla

Los identificadores deben estar escritos en minúscula, solamente se escriben en mayúscula las palabras reservadas de codificación PL/SQL. Esto incluye también nombres de stored procedures y objetos de base de datos como tablas, vistas, secuencias, etc.

b) Ejemplo

```
FUNCTION fn_retrieve_employee_name(
         p_employee_code IN srh_employees.employee_code%TYPE
         )
IS
         v_employee_name srh_employees.employee_name%TYPE;
BEGIN
         SELECT emp.employee_name
         INTO          v_employee_name
         FROM          srh_employees emp
         WHERE         emp.employee_code = p_employee_code;

         RETURN v_employee_name;
END retrieve_employee_name_fn;
```

c) Listado de Palabras Reservadas

A continuación se enlistan algunas de las palabras reservadas de codificación PL/SQL.

Las palabras seguidas de asterisco (*) corresponden también a palabras reservadas de ANSI SQL.

ACCESS	ADD*	ALL*	ALTER*	AND*
ANY*	ARRAY	AS*	ASC*	BETWEEN*
AUDIT	AT	AUTHID	AVG	BEGIN
BINARY_INTEGER	BODY	BOOLEAN	BULK	BY*
CASE	CHAR*	CHAR_BASE	CHECK*	CLOSE
CLUSTER*	COALESCE	COLLECT	COLUMN	COMMENT*
COMMIT	COMPRESS*	CONNECT*	CONSTANT	CREATE*
CURRENT*	CURRVAL	CURSOR	DATE*	DAY
DECIMAL*	DECLARE	DEFAULT*	DELETE*	DESC*
DISTINCT*	DO	DROP*	ELSE*	ELSIF

END	EXCEPTION	EXCLUSIVE*	EXECUTE	EXISTS*
EXIT	EXTENDS	EXTRACT	FALSE	FETCH
FILE	FLOAT*	FOR*	FORALL	FROM*
FUNCTION	GOTO	GRANT*	GROUP*	HAVING*
HEAP	HOUR	IDENTIFIED	IF	IMMEDIATE*
IN*	INCREMENT	INDEX*	INDICATOR	INITIAL
INSERT*	INTEGER*	INTERFACE	INTERSECT*	INTERVAL
INTO*	IS*	ISOLATION	JAVA	LEVEL*
LIKE*	LIMITED	LOCK*	LONG*	LOOP
MAX	MAXEXTENTS .MIN	MINUS*	MINUTE	
MLSLABEL*	MOD	MODE*	MODIFY ñMONTH	
NATURAL	NATURALN	NEW	NEXTVAL	NOAUDIT
NOCOMPRESS	NOCOPY	NOT*	NOWAIT*	NULL*
NULLIF	NUMBER*	NUMBER_BASE	OCIROWID	OF*
OFFLINE	ON*	ONLINE	OPAQUE	OPEN
OPERATOR	OPTION*	OR*	ORDER*	ORGANIZATIO N
OTHERS	OUT	PACKAGE	PARTITION	PCTFREE
PLS_INTEGER	POSITIVE	POSITIVEN	PRAGMA	PRIOR*
PRIVATE	PRIVILEGES*	PROCEDURE	PUBLIC*	RAISE
RANGE	RAW*	REAL	RECORD	REF
RELEASE	RENAME	RETURN	RESOURCE	REVERSE
REVOKE*	ROW*	ROWID*	ROWNUM*	ROWS*
ROWTYPE	SAVEPOINT	SECOND	SELECT	SEPARATE
SESSION*	SET*	SHARE*	SIZE*	SMALLINT*
SPACE	SQL	SQLCODE	SQLERRM	START*
STDDEV	SUBTYPE	SUCCESSFUL*	SUM	SYNONYM*
SYSDATE*	TABLE*	THEN*	TIME	TIMESTAMP
TIMEZONE_REGIO N	TIMEZONE_ABB R	TIMEZONE_MINUT E		
TIMEZONE_HOUR	TO*	TRIGGER*	TRUE TYPE	VALIDATE*
UID*	UNION*	UNIQUE*	UPDATE*	USE
VALUES*	VARCHAR*	VARCHAR2*	VARIANCE	VIEW*
WHEN	WHENEVER*	WHERE*	WHILE	WITH*
WORK	WRITE	YEAR	ZONE	USER*

4.010.2 Empaquetado de Procedimientos y Funciones

a) Regla

Salvo en caso de procedimientos y/o funciones con usos muy particulares o con restricciones en su uso por parte de lenguajes externos de la base de datos, agrupar y empaquetar los módulos dentro de paquetes.

b) Motivación

Evitar la proliferación de módulos “sueltos” facilitando la búsqueda y el mantenimiento de los mismos.

4.010.3 Modularización

a) Motivación

Se promueve el reuso y se logran mejoras en el mantenimiento de código.

4.010.4 Llamada a Stored Procedures y Cursores

a) Regla

Los parámetros deben pasarse por nombre y no en forma posicional. Se opta por esta modalidad porque tiene la ventaja de que el código es más fácil de interpretar, pues la persona que esta revisando el código, no debe referirse a la declaración del procedimiento función llamados para conocer el significado de los parámetros.

b) Motivación

Facilitar el mantenimiento y lectura del código evitando la necesidad de revisar la declaración del módulo o cursor que se esta invocando para conocer el significado de los parámetros. Esta necesidad aumenta con el aumento en la cantidad de parámetros que se necesitan, aun más cuando se declaran parámetros definidos con valores DEFAULT.
Disminuir la posibilidad de errores en la asignación de parámetros a los módulos o cursores por mal posicionamiento en la invocación de los mismos

c) Ejemplo

En lugar de:

```
DECLARE
         v_id_empleado     rsh_empleados.id_empleado%TYPE;
         v_cod_producto    ven_productos.cod_producto%TYPE;
         v_fecha_inicio_ejercicio   DATE;
         v_fecha_actual             DATE:= SYSDATE;
         v_total_venta              NUMBER(10,2);
         v_cant_total_venta  NUMBER(10);
BEGIN
         prc_calcular_ventas_prod_empl(
                v_id_empleado,
                v_cod_producto,
                v_fecha_inicio_ejercicio,
                v_fecha_actual,
                v_ total_venta,
                v_cant_total_venta,
                );
```

Invocar los procedimientos de la siguiente manera:

```
calcular_ventas_prod_empl_prc(
            p_id_empleado       => v_id_empleado,
            p_cod_producto             => v_cod_producto,
            p_fecha_desde       => v_fecha_inicio_ejercicio,
            p_fecha_hasta       => v_fecha_actual
            p_monto_total_venta        => v_ total_venta,
            p_cant_total_venta         => v_cant_total_venta,
            );
```

4.010.5 Especificación del nombre del modulo en la sentencia END

a) Regla

Agregar el nombre del programa o módulo en la sentencia END del final del bloque.

b) Motivación

Mejora la legibilidad.

c) Ejemplo

```
CREATE OR REPLACE PACKAGE BODY <package name>
IS
    PROCEDURE <procedure name>
    IS BEGIN
    …
    END <procedure name>;
    ---------------------------------------------------------------
    FUNCTION <function name>
    IS
    …
    END <function name>;

END <package name>;
```

4.010.6 Definición de secciones para paquetes

a) Regla

Los paquetes deben estructurarse siempre de la siguiente forma, aún cuando algunas secciones no posean componentes que especificar:

Package Specification:
- Definición de Excepciones
- Definición de Tipos y Subtipos Públicos
- Definición de Constantes Públicas
- Definición de Variables Públicas
- Definición de Módulos Públicos

Package Body:
- Definición de Tipos y Subtipos Privados
- Definición de Constantes Privadas
- Definición de Variables Privadas
- Definición de Módulos Privados

Implementación de Módulos Privados
Implementación de Módulos Públicos

b) Motivación

Dado que los paquetes pueden ser muy extensos y poseer varios tipos distintos de componentes, esto permite la fácil ubicación de los mismos sin tener que recorrer todo el paquete. En el anexo es posible ver un template de ejemplo de creación de un paquete.

4.010.7 Especificación de Parámetros

a) Regla

Los parámetros de procedimientos, funciones y cursores deben comenzar con el prefijo "p_". Para los parámetros de cursores se recomienda agregar además "c_" para que los parámetros de los cursores no se confundan con los nombres de los parámetros de los stored procedures que lo declaran.

Para el caso de procedimientos se debe explicitar el modo de parámetro: IN, IN OUT, OUT.

b) Ejemplo

Procedimiento

```
PROCEDURE prc_calcular_ventas_prod_empl(
         p_id_empleado IN     srh_empleados.id_empleado%TYPE,
         p_cod_producto      IN
 com_productos.cod_producto%TYPE,
         p_fecha_desde IN     DATE,
         p_fecha_hasta IN     DATE,
         p_monto_total_venta OUT
com_resumen_ventas.monto_total%TYPE,
         p_cant_total_venta    OUT
com_resumen_ventas.cant_total%TYPE,
         );
```

Función

```
FUNCTION fn_identificador_empleado(
         p_cod_empleado      IN
  srh_empleados.cod_empleado%TYPE
         )
RETURN srh_empleados.id_empleado%TYPE;
Cursor

CURSOR cur_departamentos(
p_c_cod_departamento   srh_departamentos.codigo_departamento%TYPE
)
IS
SELECT   id_departamento,
codigo_departamento,
nombre_departamento
FROM     srh_departamentos
WHERE    (
p_c_cod_departamento IS NULL
                OR
```

```
  (
p_c_cod_departamento IS NOT NULL
                        AND
                        codigo_departamento =
p_c_codigo_departamento
                        )
                );
```

4.010.8 Descripciones de cabecera de Stored Procedures y Módulos

a) Regla

Estandarizar la cabecera de los módulos y stored procedures. Secciones de explicación de propósito, especificación de historial de cambio.
Para el caso de Packages que poseen módulos de tipo procedure y function, se debe extender esta regla a cada uno de estos módulos, sean públicos o privados.

b) Motivación

Documentación autocontenida dentro de los módulos de programa.
Ayuda al mantenimiento del código.

c) Ejemplo

Procedures (standalone o módulos de packages)

```
/*
Nombre del programa: <nombre del proceso>
Objetivo: <descripción del objetivo del procedimiento>
Parámetros de entrada:
          [<parámetro 1>:<descripción>]
          …
          [<parámetro n>:<descripción>]

Parámetros de salida:
          [<parámetro 1>:<descripción>]
          …
          [<parámetro n>:<descripción>]

Notas: <Aclaraciones y supuestos que se deban hacer sobre el módulo>
Autor: <iniciales del diseñador 1[,...]>

Historial:  <se  deberá  colocar  fecha,  iniciales  del  autor  y
descripción breve de las
modificaciones realizadas, comenzando por la versión original>

Fecha            Autor    Descripción
=================================================
DD/MM/AAAA       XYZ      ..............................
*/
```

Function (standalone o módulos de packages)

```
/*
```

```
Nombre del programa: <nombre del proceso>
Objetivo: <descripción del objetivo de la función>
Parámetros de entrada:
         [<parámetro 1>:<descripción>]
         …
         [<parámetro n>:<descripción>]

Parámetros de salida:
         [<salida de función>:<descripción>]

Notas: <Aclaraciones y supuestos que se deban hacer sobre el módulo>
Autor: <iniciales del diseñador 1[,...]>

Historial:  <se  deberá  colocar  fecha,  iniciales  del  autor  y
descripción breve de las
modificaciones realizadas, comenzando por la versión original>

Fecha           Autor   Descripción
=================================================
DD/MM/AAAA      XYZ     ..............................
 */
```

Packages

```
/*
Nombre del programa: <nombre del proceso>
Objetivo: <descripción del objetivo del procedimiento>

Notas: <Aclaraciones y supuestos que se deban hacer sobre el paquete>
Autor: <iniciales del diseñador 1[,...]>

Historial:  <se  deberá  colocar  fecha,  iniciales  del  autor  y
descripción breve de las
modificaciones realizadas, comenzando por la versión original>

Fecha           Autor   Descripción
=================================================
DD/MM/AAAA      XYZ     ..............................
 */
```

Muchas herramientas IDE de codificación permiten definir plantillas iniciales de Stored Procedures. Si se acostumbra utilizar o se define alguna de estas herramientas como estándar dentro de un proyecto es recomendable customizar estas plantillas para que contengan el template de documentación de cabecera y compartirlas con todo el equipo de desarrollo.

4.020 Indentación

4.020.1 Estilo sentencia SELECT

a) Regla

Las palabras claves SELECT, INTO, FROM, WHERE, GROUP BY, HAVING y ORDER BY van en la línea 0 relativo al código que se está escribiendo.

Las columnas del SELECT, deben especificarse de a una por reglón con un nivel más de tabulación con respecto a las palabras claves que la preceden. Las siguientes columnas deben ubicarse en el reglón siguiente nivelados tabularmente con el nombre de la primera columna.

Las columnas de ORDER BY y expresiones de HAVING deben ir en el reglón siguiente con un nivel más de tabulación con respecto a las palabras claves que la preceden.

La palabra clave FROM va seguida del nombre de la tabla (o vista) con un nivel más de tabulación. Las siguientes tablas deben ubicarse en el reglón siguiente nivelados tabularmente con el nombre de la primera tabla.

La palabra clave WHERE va seguida de la primera condición con un nivel más de tabulación. Las condiciones siguientes deben comenzar con la sentencia AND (u OR) a la misma altura que la sentencia WHERE y las condiciones al mismo novel de tabulación que la condición anterior. No se deben especificar más de un conector lógico por reglón. Si se necesitara una condición compleja se con combinación de condiciones se deben representar con paréntesis y un nivel más de tabulación.

Las subqueryes deben estar tabuladas y seguir las mismas reglas de estilo. Deben estar entre paréntesis

Para las condiciones que impliquen una subquery (EXISTS / IN), deben abrir paréntesis y ubicar la subquery con el mismo estilo anterior pero tabulados un nivel mas que el inicio de la expresión de condición.

Las sentencias de operaciones de conjuntos UNION, UNION ALL y MINUS se escriben en un reglón aparte en la misma línea que la sentencia SELECT al igual que el paréntesis de las subqueries.

La función DECODE debe ser tratada al igual que una columna, especificando en el mismo reglón solamente la expresión a evaluar. Las expresiones con la que se compara y el valor a asignar deben ubicarse en los reglones siguientes con dos niveles más de tabulación, el valor evaluado y el asignado deben estar separados por un Tab. El valor default debe ir en el último reglón separado de los demás al mismo nivel que las expresiones a comparar.

<tab> es realmente un TAB y no espacios en blanco contiguos. La longitud de un
TAB es equivalente a 8 espacios en blanco, pero 8 espacios en blanco NO equivalen al caracter de representación de un TAB.

Se recomienda setear el uso de TABs en las herramientas de desarrollo deshabilitando el uso de espacios en blanco contiguos.

b) Ejemplo

Select 1:

```
SELECT <Tab> columna1,
               <Tab> columna2,
...
FROM    <Tab> tabla1,
               <Tab> tabla2,
...
WHERE <Tab>  expresion
AND       <Tab> (
               <Tab>  <Tab> expresion
```

```
                <Tab>  <Tab> OR
                <Tab>  <Tab> expresion
                <Tab>)
AND EXISTS (
                <Tab> SELECT
                <Tab> FROM

                <Tab> WHERE
                <Tab>)
ORDER BY
                <Tab> columna1,
                <Tab> columna2,
```

…

Select 2:

```
SELECT <Tab> columna1,
                <Tab> columna2,
...
FROM    <Tab> tabla1,
                <Tab> tabla2,
...
WHERE <Tab> expresion
GROUP BY
<Tab>                columna1,
<Tab>                columna2,
…
HAVING
              <Tab> expresion1
AND      <Tab> expresion2
….
```

Select 3:

```
(
SELECT <Tab> columna1,
...
FROM    <Tab> tabla1
...
WHERE <Tab> expresion1
)
UNION
(
SELECT <Tab> columna1,
...
FROM    <Tab> tabla1,
                <Tab> tabla2,
...
WHERE <Tab> expresion1
)
```

Select 4:

```
SELECT <Tab> DECODE (columna1,
                <Tab>  <Tab> valor_evaluado1, <Tab> valor_asumido1,
                <Tab>  <Tab> valor_evaluado2, <Tab> valor_asumido2,
                <Tab>  <Tab> valor_evaluado3, <Tab> valor_asumido3,
                <Tab>  <Tab> valor_default)
...
FROM    <Tab> tabla1
```

4.020.2 Estilo sentencia INSERT

c) Regla

Las palabras claves INSERT y VALUES van en la línea 0 relativo al código que se está escribiendo.

La palabra clave INSERT va seguida de la palabra clave INTO y el nombre de la tabla o vista sobre la cual se lleva a cabo la operación.

Se deben especificar los nombres de las columnas en la sentencia. Las columnas se deben escribir a partir de la línea siguiente a la de la sentencia INSERT con un nivel más de tabulación con respecto a la sentencia. Se especifica una columna por renglón.

Luego de la sentencia VALUES se especifica un valor de asignación por cada reglón, alineados en un nivel más de tabulación.

Las sentencias INSERT que toman como fuente de valores el resultado de una subquery deben especificar los paréntesis y la subquery en un nivel más de tabulación con respecto a la palabra clave INSERT. La subquery debe seguir las reglas de estilo de las sentencias SELECT.

d) Ejemplo

Insert 1:

```
INSERT INTO tabla (
<Tab>     columna1,
<Tab>     columna2,
…
<Tab>     columna_n
<Tab>     )
VALUES(
<Tab>     valor_columna1,
<Tab>     valor_columna2,
…
<Tab>     valor_columna_n
<Tab>     );
```

Insert 2:

```
INSERT INTO tabla (
<Tab>     columna1,
<Tab>     columna2,
…
<Tab>     columna_n
```

```
    <Tab>      )
    <Tab>      (
<Tab>      SELECT <Tab> columna1,
<Tab>                   <Tab> columna2,
<Tab>      ...
<Tab>                   <Tab> columna_n,
<Tab>      FROM    <Tab> tabla2
<Tab>      ...
<Tab>      WHERE <Tab> expresion1
<Tab>      );
```

4.020.3 Estilo sentencia UPDATE

a) Regla

Las palabras claves UPDATE, SET y WHERE van en la línea 0 relativo al código que se está escribiendo.

La palabra clave UPDATE va seguida del nombre de la tabla o vista sobre la cual se lleva a cabo la operación con un nivel más de tabulación.

Luego de la palabra clave SET se especificar la primera columna a actualizar con un nivel más de tabulación. Si existieran mas columnas a actualizar se deben especificarse de a una por reglón alineadas con la primera columna a actualizar.

Las condiciones que se especifican en la sección de la palabra WHERE siguen las mismas reglas que en la sentencia SELECT.

Si se utilizara una subquery en la asignación de un valor, debe estar tabulada y seguir las mismas reglas de estilo que en la sentencia SELECT, además debe estar entre paréntesis.

b) Ejemplo

Update 1:

```
UPDATE <Tab> tabla
SET         <Tab> columna1 = valor1,
                <Tab> columna2 = valor2,
…
WHERE   <Tab> expresion1
AND         <Tab> expresion2
```

Update 2:

```
UPDATE <Tab>  tabla1
SET         <Tab>  columna1 = (
<Tab>          SELECT <Tab> columna,
<Tab>          FROM    <Tab> tabla2
<Tab>          WHERE <Tab> expresion2
<Tab>          )
```

4.020.4 Estilo sentencia DELETE

a) Regla

Las palabras claves DELETE y WHERE van en la línea 0 relativo al código que se está escribiendo.

La palabra clave DELETE va seguida del nombre de la tabla o vista sobre la cual se lleva a cabo la operación con un nivel más de tabulación.

Las condiciones que se especifican en la sección de la palabra WHERE siguen las mismas reglas que en la sentencia SELECT.

b) Ejemplo

```
DELETE  <Tab> tabla1
WHERE   <Tab> expresion1
AND         <Tab> expresion2
```

4.020.5 Estilo bloques generales PL/SQL

c) a) Regla

Dentro de un bloque común de PL/SQL las palabras claves DECLARE (cuando se trate de un bloque anónimo), BEGIN y END van en la línea 0 relativo al código que se está escribiendo.

Las sentencias e identificadores deben alinearse con un nivel de tabulación más con respecto al bloque que determinan estas palabras claves. Esta regla se aplica de la misma manera para bloques anidados

Si el bloque poseyera una sección de manejo de excepciones (EXCEPTION) el mismo debe ubicarse en la misma línea que el BEGIN y el END del bloque al que pertenece.

Las palabras claves WHEN de la sección de manejo de excepciones se escriben en un nivel más de tabulación. Seguido a la palabra WHEN con un nivel más de tabulación se ubica el identificador de excepción. En el renglón siguiente alineado con el WHEN se escribe la palabra clave THEN. Las sentencias a ejecutarse para la excepción se comienzan a escribir en el renglón siguiente con un nivel más de tabulación que el THEN.

c) Ejemplo

```
DECLARE
    <Tab>  v_variable1 tipo_dato;
    <Tab>  v_variable2 tipo_dato;

BEGIN

    <Tab> BEGIN
    <Tab>       <Tab> sentencia1;
    <Tab>       <Tab> sentencia2;
    <Tab> EXCEPTION
    <Tab> WHEN exception1
    <Tab> THEN
```

```
   <Tab>     <Tab> sentencia3;
      <Tab>        <Tab> sentencia4;
      <Tab>        <Tab> RAISE;
      <Tab> END;
…

EXCEPTION
WHEN exception2
THEN
      <Tab>        sentencia1;
END;
```

4.020.6 Estilo comando IF THEN ELSE ELSIF

a) Regla

Las palabras claves IF, THEN, ELSE, ELSIF y ENDIF van en la línea 0 relativo al código que se está escribiendo.

Las expresiones que deben evaluar más de una condición deben encerrarse entre paréntesis y separase por renglón. En caso que se utilicen operadores lógicos se deben alinear las expresiones a evaluar a la misma altura que los operadores.

En caso de que se aniden sentencias IF se deben respetar las reglas anteriores sumando un nivel más de tabulación con respecto a la estructura que la contiene.

b) Ejemplo

IF-THEN-ELSE 1:

```
IF expresion1

THEN

     <Tab>        sentencia1;

ELSE

     <Tab>        sentencia2;

END IF;
```

IF-THEN-ELSE 2:

```
IF            <Tab>  expresion1

AND       <Tab> (

               <Tab>  <Tab> expresion2

               <Tab>  <Tab> OR
```

```
                <Tab>  <Tab> expresion3
                <Tab>)
AND       <Tab> expresion4
THEN
     <Tab>       sentencia1;
ELSE
     <Tab>       sentencia2;
END IF;
```

IF- ELSIF 3:

```
IF expresion1
THEN
     <Tab>       sentencia1;
ELSIF expresion2
THEN
     <Tab>       sentencia2;
ELSIF expresion3
THEN
     <Tab>       sentencia3;
….
ELSE
     <Tab>       sentenciaN;
END IF;
```

IF- ELSE 4:

```
IF expresion1
THEN
     <Tab>       IF expresion2
     <Tab>      THEN
```

```
      <Tab>       <Tab>    sentencia1;

      <Tab>       ELSE

      <Tab>       <Tab>    sentencia1;

      <Tab>       END IF;

ELSE

      <Tab>       sentencia2;

END IF;
```

4.020.7 Estilo comando LOOP

a) Regla

Las palabras claves LOOP y END LOOP van en la línea 0 relativo al código que se está escribiendo.

La palabra clave EXIT se escribe en el renglón siguiente al LOOP con un nivel más de tabulación. La condición que debe evaluarse para salir del LOOP se escribe con un nivel más de tabulación en el renglón siguiente al de la palabra clave WHEN. Si la expresión es compleja deben encerrarse entre paréntesis y separase por renglón. En caso que se utilicen operadores lógicos se deben alinear las expresiones a evaluar a la misma altura que los operadores.

En caso de que se aniden estructuras LOOP se deben respetar las reglas anteriores sumando un nivel más de tabulación con respecto a la estructura que la contiene.

b) Ejemplo

```
LOOP

     <Tab>     EXIT

     <Tab>     WHEN <Tab>  expresion1

     <Tab>     sentencia1;

     <Tab>     sentencia2;

END LOOP;
```

4.020.8 Estilo comando WHILE LOOP

a) Regla

Las palabras claves WHILE, LOOP y END LOOP van en la línea 0 relativo al código que se está escribiendo.

Seguido a la palabra clave WHILE con un nivel más de tabulación se escribe la condición que debe evaluarse para salir del WHILE. Si la expresión es compleja deben encerrarse entre paréntesis y separase por renglón. En caso que se utilicen operadores lógicos se deben alinear las expresiones a evaluar a la misma altura que los operadores.

En caso de que se aniden estructuras WHILE LOOP se deben respetar las reglas anteriores sumando un nivel más de tabulación con respecto a la estructura que la contiene.

b) Ejemplo

```
WHILE
              <Tab> (
              <Tab> expresion1
              <Tab> AND
              <Tab> Expression 2
              <Tab> )
LOOP
              <Tab> EXIT
              <Tab> WHEN <Tab>  expresion1
              <Tab> sentencia1;
              <Tab> sentencia2;
END LOOP;
```

4.020.9 Estilo comando FOR LOOP

c) a) Regla

Las palabras claves FOR, LOOP y END LOOP van en la línea 0 relativo al código que se está escribiendo.

Seguido a la palabra clave FOR con un nivel más de tabulación se escribe la condición que debe evaluarse para salir del FOR LOOP. Si la expresión es compleja deben encerrarse entre paréntesis y separase por renglón. En caso que se utilicen operadores lógicos se deben alinear las expresiones a evaluar a la misma altura que los operadores.

En caso de que se aniden estructuras FOR LOOP se deben respetar las reglas anteriores sumando un nivel más de tabulación con respecto a la estructura que la contiene.

c) Ejemplo

```
FOR       <Tab> expresion1

LOOP

              <Tab> sentencia1;

              <Tab> sentencia2;

END LOOP;
```

4.020.10 Estilo Especificación de Stored Procedure

a) Regla

En la primera línea se especifica la palabra clave PROCEDURE o FUNCTION, según corresponda y el nombre del stored procedure. Si el mismo posee parámetros se agrega el paréntesis después del identificador.

Los parámetros se especifican en el renglón siguiente, uno por renglón. Cerrando el paréntesis en un renglón siguiente, si se hubiera abierto previamente.

En caso de tratarse de una función se debe especificar la palabra RETURN alineado con la palabra FUNCTION, seguido de la declaración de la variable de retorno.

La palabra clave IS se escribe alineada a la palabra clave PROCEDURE o FUNCTION.

El cuerpo del stored procedure debe cumplir con las reglas de bloques PL/SQL.

b) Ejemplo

Procedure

```
PROCEDURE prc_calcular_ventas_prod_empl(

              <Tab>         p_id_empleado                              IN
srh_empleados.id_empleado%TYPE,

              <Tab>           p_cod_producto                           IN
com_productos.cod_producto%TYPE,

              <Tab> p_fecha_desde    IN      DATE,

              <Tab> p_fecha_hasta    IN      DATE,

              <Tab>              p_monto_total_venta                   OUT
com_resumen_ventas.monto_total%TYPE,

              <Tab>      p_cant_total_venta                            OUT
com_resumen_ventas.cant_total%TYPE,

              <Tab>)
```

```
IS

BEGIN

            <Tab> Sentencia1;

            <Tab> Sentencia2;

            …

EXCEPTION

            <Tab> WHEN excepcion1

            <Tab> THEN


            <Tab>     <Tab> Sentencia3


END prc_calcular_ventas_prod_empl;
```

Function

```
FUNCTION fn_identificador_empleado(

            <Tab>           p_cod_empleado                      IN
srh_empleados.cod_empleado%TYPE

            <Tab>)

RETURN srh_empleados.id_empleado%TYPE

IS

BEGIN

            <Tab> Sentencia1;

            <Tab> Sentencia2;

            …

            <Tab> RETURN valor1;

END fn_identificador_empleado;
```

Si es posible se recomienda alinear las sentencias IN / OUT y las declaración de los tipos de datos de los parámetros. Esto ayuda a la fácil lectura del código. Sin embargo no es obligatorio dado que la diferencia de longitudes de los nombres de los parámetros puede

hacer que la identación deje la declaración de los tipos de datos muy alejados de los nombres de los parámetros haciendo la lectura más dificultosa.

4.030 Estructuras de datos

4.030.1 Definir los tipos de datos según su uso

a) Regla

Definir de forma correcta los tipos de datos con un uso computacional.

b) Motivación

Utilizar los tipos de datos correctos.

c) Recomendaciones

NUMBER: Evita desperdiciar memoria.
CHAR: Evitar CHAR a menos que sea específicamente necesario.
VARCHAR: Evitar VARCHAR en favor de VARCHAR2.
VARCHAR2: Evitar hard-code una longitud máxima.
INTEGER: usar PLS_INTEGER cuando sea posible.

4.030.2 Usar %TYPE y %ROWTYPE

a) Regla

Definir los tipos de datos de las variables con el uso de %TYPE y %ROWTYPE.

b) Motivación

El código se adapta automáticamente a las modificaciones estructurales del modelo de datos. Si la variable del código tiene el mismo tipo de dato que una columna de la tabla, utilizar %TYPE. Si un registro tiene la misma estructura que una fila de la tabla utilizar %ROWTYPE para declararlo.

c) Ejemplo

No es correcto

```
DECLARE

   l_title VARCHAR2(100);
```

Es correcto

```
DECLARE

   l_title book.title%TYPE;
```

d) Beneficios

El código se adapta automáticamente a las modificaciones estructurales del modelo de datos.

Las declaraciones se auto documentan.

e) Desafío

Se necesita saber los nombres de las columnas de las tablas.

4.030.3 No fijar longitud de los VARCHAR2

a) Regla

No fijar el valor de la longitud máxima de los VARCHAR2.

b) Motivación

El problema se basa en que normalmente este tipo de datos sufre muchos cambios.
Por ejemplo su máximo permitido cambio de 4000 bytes en Oracle 11g.

c) Ejemplo

En vez de:

```
DECLARE

   big_string VARCHAR2(4000);
```

Crear un tipo especial:

```
CREATE OR REPLACE app_types

IS

   SUBTYPE dbmax_vc2 IS VARCHAR2(4000);
```

d) Beneficios

Se evita que se levanten excepciones del tipo VALUE ERROR a lo largo de la ejecución.

4.030.4 Centralizar definiciones de tipos

a) Regla

Centralizar las definiciones de TYPE en la especificación de un package.

b) Motivación

Estandarizar tipos usados (TYPE), los cuales pueden ser usados en múltiples programas.

c) Ejemplo

```
CREATE OR REPLACE PACKAGE  colltype

IS

   TYPE boolean_ntab IS TABLE OF BOOLEAN;

   TYPE boolean_itab IS TABLE OF BOOLEAN

         INDEX BY BINARY_INTEGER;

   TYPE date_ntab IS TABLE OF DATE;

   TYPE date_itab IS TABLE OF DATE

         INDEX BY BINARY_INTEGER;

   ...

END colltype;
```

d) Beneficios

Se escribirá el código mas rápido, minimizando los bugs al utilizar tipos pre-definidos. Centralizar el mantenimiento de tipos.

e) Desafíos

Los desarrolladores deben ser entrenados para identificar los tipos pre-definidos o agregar nuevos tipos en los packages existentes.

Manejo de Constantes

4.030.5 Utilizar constantes

a) Regla

Usar declaraciones de constantes para aquellos datos que no sufran cambios. Es decir, usar la palabra clave CONSTANT.

b) Motivación

Legibilidad y corrección del código.

c) Ejemplo

```
DECLARE

    c_date CONSTANT DATE := TRUNC(SYSDATE);

    c_category CONSTANT book.category%TYPE;
```

```
BEGIN
    checkouts.analize(c_date, c_category);
    ...
    FOR rec IN (SELECT *
                        FROM book
                        WHERE category=c_category)
    LOOP
    ...
    END LOOP;
```

d) Beneficios

El código, con este tipo de declaraciones queda auto documentado; expresando explícitamente que este dato no debe ni pueden ser modificados.

Un desarrollador no podrá de esta manera cometer un error modificando el valor de este tipo de estructuras de datos.

4.030.6 Crear packages de constantes

a) Regla

Las constantes deben ser recuperadas con funciones empaquetadas que retornen un valor constante. Las constantes pueden ser valores configurables obtenidos de una tabla de configuración o bien constantes propiamente dichas. En el segundo caso, la constante debe estar definida en el BODY del PACKAGE.

b) Motivación

Evitar hardcodear literales.

Permite independizarse de la forma en que fue definida la constante dada, que puede tratarse tanto de una configuración del sistema como de una constante propiamente dicha. Además, puede pasar que ante una modificación del sistema, el valor deba ser obtenido de otra fuente.

Evita compilar la especificación del PACKAGE ante un cambio del valor de la constante.

Permite que todas las aplicaciones accedan a la constante.

No permite que los usuarios del PACKAGE conozcan el valor de la constante, evitando vicios de programación.

Permite validar configuraciones una sola vez, evitando overhead de validación y posibles errores.

Permite cambiar configuraciones de sistemas fácilmente a través de tablas de configuraciones.

c) Ejemplo

Este es el package donde esta la lógica de negocios:

```
PACKAGE pck_app_emp IS

    PROCEDURE xxxx ();


END pck_emp;

-- Este es el package donde estan las constantes del package emp_pck

PACKAGE pck_app_emp_const IS

    PROCEDURE const1 ();

END pck_emp;
```

-- Este es el package donde están las constantes del package emp_pck

```
PACKAGE BODY pck_app_emp_const IS

    v_const1 NUMBER;

    FUNCTION const1 () RETURN NUMBER IS

    BEGIN

        IF v_const1 IS NULL THEN

            v_const1 := to_number(pck_app_cfg.get('CONST1');

        END IF;

        RETURN v_const1;

    END;

END pck_emp;
```

De esta forma se haría uso de las constants dentro de los packages que contienen la lógica de negocios.

```
PACKAGE BODY pck_app_emp IS

  PROCEDURE xxxx () IS

  BEGIN

     --
```

```
      IF pck_app_emp_const.const1 > x THEN
      --
   END;
END pck_emp;
```

d) Beneficios

Incrementar la legibilidad y el mantenimiento del código.

Se establece un único lugar para que los desarrolladores puedan agregar nuevas constantes para ocultar literales.

Manejo de Variables

4.030.7 Inicialización de variables

a) Regla

Realizar inicializaciones complejas de variables en la sección ejecutable, nunca en la sección de declaraciones.

b) Motivación

Si en la sección de declaración se inicializa una variable erróneamente, este error se propaga a lo largo de la ejecución, sin forma de manejarlo. Además dificulta el debug de este problema.

c) Ejemplo

Código Peligroso:

```
CREATE OR REPLACE PROCEDURE prc_find_bestsellers
IS
 l_last_title_book.title%TYPE := last_search(SYSDATE);
 l_min_count INTEGER(3) := bestseller.limits (bestseller.low);
BEGIN
```

Alternativa más segura:

```
CREATE OR REPLACE PROCEDURE prc_find_bestsellers
IS

    l_last_title book.title%TYPE;
    l_min_count INTEGER(3);
```

```
    PROCEDURE INIT IS
    BEGIN
         l_last_title := last_search (SYSDATE);
         l_min_count := bestseller.limits (bestseller.low);
    EXCEPTION
         --capturar y manejar los errores dentro del programa
    END
BEGIN
    init;
```

d) Beneficios

Incrementa la legibilidad.

Mejora la performance en ciertos casos pues dependiendo del flujo de control del programa el código de inicialización no siempre es necesario.

Evita propagar excepciones. Las excepciones dentro de la sección de declaración se propagan automáticamente; no pueden ser capturadas dentro de la subrutina.

4.030.8 Ocultar expresiones complejas

a) Regla

Ocultar las funciones complejas con variables booleanas o funciones.

b) Motivación

Ocultar expresiones complejas incrementa la legibilidad.

c) Ejemplo

Este es un ejemplo de lo que no hay que hacer:

```
IF total_sal BETWEEN 10000 AND 50000
    AND emp_status (emp_rec.empno) = 'N'
    AND ( MONTHS_BETWEEN (emp_rec.hiredate, SYSDATE ) > 10)
THEN
    give_raise (emp_rec.empno);
END IF;
```

Mejor usar una variable local:

```
DECLARE
    eligible_for_raise BOOLEAN;
BEGIN
    eligible_for_raise :=
  total_sal BETWEEN 10000 AND 50000
  AND emp_status (emp_rec.empno) = 'N'
  AND (MONTHS_BETWEEN (emp_rec.hiredate, SYSDATE) > 10);
     --
    IF eligible_for_raise THEN
  give_raise (emp_rec.empno);
    END IF;
END;
```

Mejor aún, usar una función...

```
IF eligible_for_raise (totsal, emprec) THEN
    give_raise (empr_rec.empno);
END IF;
```

d) Beneficios

De esta manera hará que el código sea mucho más fácil de leer y entender por cualquier persona.

Facilita la detección de errores.

e) Desafíos

Verificar que los cambios hechos no introduzcan bugs en la aplicación.

4.030.9 No recargar el uso de estructuras de datos

a) Regla

Declarar y manipular estructuras de datos separadas para distintos requerimientos.

b) Motivación

Evitar de introducir estructuras de datos confusas y bugs.

c) Ejemplo

En este ejemplo la variable intval se está utilizando para dos propósitos diferentes. Deberían haberse declarado dos variables; una para cada propósito. Por ejemplo:

```
DECLARE
 intval INTEGER;
BEGIN
 intval:= list_of_books.COUNT;
 IF intval > 0 THEN
  intval := list_of_books (list_of_books.FIRST).page_count;
  analize_book (intval);
 END IF;
END;
```

Hacer lo siguiente:

```
DECLARE
 … otras declaraciones …
 l_book_count INTEGER;
 l_page_count INTEGER;
BEGIN
 l_book_count := list_of_books.COUNT;
 IF l_book_count > 0
 THEN
  l_page_count := list_of_books (list_of_books.FIRST).page_count;
  analize_book(l_page_count);
 END IF;
END;
```

d) Beneficios

Hace más fácil entender qué es lo que hace el código.

Se puede realizar un cambio en el uso de una variable sin afectar otras áreas del código.

4.030.10 Limpiar las estructuras de datos

a) Regla

Limpiar las estructuras de datos una vez que el programa haya terminado.

b) Motivación

Existen muchos escenarios en donde es absolutamente crucial realizar la limpieza de las estructuras de datos con métodos propios.

c) Ejemplo

```
CREATE OR REPLACE PROCEDURE prc_busy_busy

IS

   fileid UTL_FILE.file_type;

   dyncur       PLS_INTEGER;

BEGIN

         dyncur := DBMS_SQL.open_cursor;

         OPEN book_pkg.all_books_by ('FEUERSTEIN');

         fileid                  :=                  UTL_FILE.fopen
('/apps/library','bestsellers.txt','R')

EXCEPTION

   WHEN NO_DATA_FOUND

   THEN

         err.log;

         RAISE;

END;

CREATE OR REPLACE PROCEDURE prc_busy_busy

IS

   fileid UTL_FILE.file_type;

   dyncur PLS_INTEGER;


   PROCEDURE cleanup IS

   BEGIN

         IF book_pkg.all_books_by%ISOPEN
```

```
          THEN
                 CLOSE book_pkg.all_books_by;
          END IF;
          DBMS_SQL.CLOSE_CURSOR (dyncur);
          UTL_FILE.fclose (fileid);
   END;
BEGIN
   dyncur := DBMS_SQL.open_cursor;
   OPEN book_pkg.all_cursor_by ('FEUERSTEIN');
   fileid:= UTL_FILE.fopen ('/apps/library','bestsellers.txt','R');
   ...
   cleanup;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
          err.log;
          cleanup;
          RAISE;
END;
```

d) Beneficios

Se reduce la probabilidad que los programas utilicen memoria de más y que afecten a otros programas.

Futuros desarrolladores pueden fácilmente agregar nuevas operaciones de limpieza en un lugar determinado.

e) Desafíos

Definir un formato estándar para sus programas, incluyendo procedimientos de inicialización y limpieza.

4.030.11 Evitar conversiones de tipos de datos implícitas

a) Regla

Tener cuidado y evitar conversiones de datos implícitas.

b) Motivación

El comportamiento de las conversiones puede no ser intuitivo. Las reglas de conversión no están bajo el control del desarrollador.

c) Ejemplo

No es correcto:

```
DECLARE
     my_birthday DATE := '09-SEP-58';
```

Es correcto:

```
DECLARE
     my_birthday DATE := TO_DATE ('09-SEP-58','DD-MM-RR');
```

d) Beneficios

El comportamiento del código es más predecible y consistente, puesto que no depende de recursos externos al código.

4.030.12 No usar variables globales

a) Regla

Sólo usar globales en packages si no existe otro modo de resolverlo y usarlas exclusivamente en el body para casos en que se necesite persistencia a nivel de sesión.

b) Motivación

Evitar dependencias ocultas y efectos laterales.

c) Ejemplo

Función con una dependencia oculta sobre una variable global

```
CREATE OR REPLACE FUNCTION fn_overdue_fine (
    isbn_in IN book.isbn%TYPE)
    RETURN NUMBER
IS
    l_days_overdue NUMBER;
BEGIN
    l_days_overdue := pkg_overdue.days_overdue (isbn_in, SYSDATE);
```

```
    RETURN ( l_days_overdue * pkg_overdue.g_daily_fine);

END;
```

Se puede reemplazar la dependencia agregando un parámetro

```
CREATE OR REPLACE FUNCTION fn_overdue_fine (

   isbn_in IN book.isbn%TYPE,

   daily_fine_in IN NUMBER)

   RETURN NUMBER

IS

   l_days_overdue NUMBER;

BEGIN

   l_days_overdue := pkg_overdue.days_overdue (isbn_id, SYSDATE);

   RETURN (l_days_overdue * daily_fine_in);

END;
```

d) Beneficios

Reduce la interdependencia entre programas.

Modificar un programa sin la preocupación de afectar a otros programas.

e) Desafíos

Es necesario recodificar los programas para reemplazar las referencias globales por parámetros.

Modularidad

4.030.13 Programar modularmente

a) Regla

Los procedimientos y funciones PL/SQL deben ser codificados en forma modular y encapsuladas evitando duplicaciones de código. En general una rutina no debería extenderse en más de una página; rutinas extensas indican alta probabilidad de necesidades de refactorización.

b) Motivación

Se promueve el reuso y se logran mejoras en el mantenimiento de código.

4.030.14 Utilizar templates para packages y funciones empaquetadas

a) Regla

Deben utilizarse los templates definidos por El MEM.

b) Beneficios

Contienen el esqueleto de toda la documentación requerida por El MEM.
Promueven los estándares.

4.030.15 Reglas de negocios definidas en funciones

a) Regla

Encapsular y definir reglas de negocios y formulas en los encabezados de funciones.

b) Motivación

Como las estructuras de datos e interfaces de usuarios, las reglas de negocios cambian frecuentemente. De esta forma se genera código que se adapta fácilmente a estos cambios.

c) Beneficios

Se puede actualizar el código de una regla de negocio o fórmula más rápidamente.

El código es mucho más fácil de entender.

d) Desafíos

Esto significará tener una disciplina y una planificación avanzada. Antes de comenzar a construir su aplicación, crear un conjunto de packages que definan las reglas de negocios y fórmulas para las distintas áreas de funcionalidad.

4.030.16 No usar procedures o funciones stand-alone

a) Regla

Dentro de la base de datos no se debe usar procedures o funciones stand-alone.
Deben usarse los packages. Esta regla es aplicable a código almacenado solamente.

b) Motivación

Los packages encapsulan lógica relacionada y estado, haciendo los componentes más fáciles de entender y mantener. Por otro lado, los objetos de base de datos dependientes del package no deberán recompilarse si se cambia la implementación, mientras que modificar un stored procedure hará que los objetos dependientes se tengan que recompilar. Salvo que se deba cambiar la interfaz, el mantenimiento del package solamente modificará el body. Por último, en Oracle es no es posible ocultar el código fuente con packages (pues se puede wrappear el body pero no la especificación).

c) Beneficios

Reuso, minimizar efectos de recopilación de stored procedures.

4.030.17 Encapsular estructuras de datos y funcionalidad relacionada en un solo paquete

a) Regla

Agrupar las Estructuras de Datos y funcionalidad relacionada en packages de la siguiente forma:

Un package con api por tabla importante o más usada.
Otro package con las queries.
Otro package con la lógica.

b) Motivación

Crear contenedores intuitivos para cada área de la funcionalidad.

c) Ejemplo

Construyendo un sistema de gestión de bibliotecas identifico rápidamente las áreas funcionales siguientes:

Información de lector: mantener la tabla subyacente. Establecer las reglas sobre quién es lector válido o habilitado.

Información del libro: mantener la tabla subyacente. Definir la validación de un ISBN, etc.
Historia de préstamos: mantener las tablas subyacentes para realizar el seguimiento sobre quienes tomaron prestado un libro, qué libro es y cuándo lo pidieron.

Multas por atraso: Colección de fórmulas y de reglas del negocio para procesar cargas por atrasos.

Reservaciones especiales: mantener la tabla subyacente, y unificar todas las reglas que gobiernan que un lector pueda reservar un libro.

Información del editor: mantenimiento de las tabla subyacente.

Ahora creo los paquetes separados para cada subconjunto de datos o especificación funcional.

d) Beneficios

Facilita mucho más la construcción y el mantenimiento de programas

Se podrán explotar las características en si mismas que brindan los packages (ej. información persistente, overloading, sección de inicialización, etc.)

e) Desafíos

No esta definido formalmente como agrupar las tablas de descripciones o si convendría generar package independientes para cada una. En ciertos casos convendrá encapsularlas dentro de packages de la entidad fuerte relacionada mientras que en otros podrá definirse un package exclusivo.

4.030.18 Poner la lógica del negocio en packages con interfaces bien definidas

a) Regla

Incluir en packages las interfaces de negocio bien definidas y la manipulación funcional.

Lo importante de esta regla es “usar packages”, no programar en el cliente ni en stored procedures stand-alone. Hay una regla aparte para no usar stored procedures.

b) Motivación

Exponer la información y las reglas de negocios en una forma ordenada, a través de la especificación del package.

c) Ejemplo

Construir un utilitario de cálculo de tiempos.

```
DECLARE

   l_start_time PLS_INTEGER;

   l_stop_time PLS_INTEGER;

BEGIN

   l_start := DBMS_UTILITY.GET_TIME;

   overdue.calculate_fines;   -- Proceso

   l_end := DBMS_UTILITY.GET_TIME;

   pl ( 'Calculated fines in ' || ' ( l_end - l_start ) / 100 || '
seconds' ) ;

END;
```

Dos temas acerca de este código:

Involucra mucho código para calcular el tiempo transcurrido.b
La formula está expuesta, ¿qué pasa si la fórmula cambia?
Solución alternativa: construir un package

```
CREATE OR REPLACE PACKAGE pkg_tmr

IS

   PROCEDURE capture;

   PROCEDURE show_elapsed;

END tmr;

/
```

```
CREATE OR REPLACE PACKAGE BODY pkg_tmr

IS

   c_bignum INTEGER := power ( 2, 32 );

   last_timing NUMBER := NULL;

PROCEDURE capture IS

   BEGIN

         last_timing := DBMS_UTILITY.GET_TIME;

   END capture;


   PROCEDURE show_elapsed  IS

   BEGIN

         pl ( MOD (DBMS_UTILITY.GET_TIME - last_timing + c_bignum,
c_bignum ) );

   END show_elapsed;

END pkg_tmr
```

Ahora se puede calcular el tiempo transcurrido de la siguiente manera:

```
BEGIN

   tmr.capture;

   overdue.calculate_fines;

   tmr.show_elapsed

END;
```

d) Beneficios

Con el uso de packages para ocultar la complejidad, se emplea naturalmente refinamiento por etapas.

El código resultante es fácil de entender, usar y mantener.

Con el ocultamiento de formulas, estas pueden ser fijas o bien ser modificadas en el tiempo

4.030.19 Construir especificación de paquetes antes que el Package body

a) Regla

Definir primero la especificación con los headers de los procedures y después completar los bodies.

b) Motivación

Desarrollar una disciplina de "especificación primero".

c) Beneficios

El software resultante se comporta mucho mejor y es fácilmente modificable.

Se pierde menos tiempo solucionando errores en la etapa de desarrollo.

d) Desafíos

Se podría crear un package body que contenga trozos de subprogramas, los cuales tengan un mínimo de código necesario como para que el package compile

4.030.20 Uso de packages para información persistente

a) Regla

Usar el cuerpo del package como almacenamiento de información persistente como cache y optimizador del proceso de manejo de datos.

b) Motivación

La declaración de datos en package, no en funciones y procedimientos individuales, permite que estos datos sean persistentes a lo largo de una sesión.

c) Ejemplo

Consideremos la función USER que retorna el usuario actual conectado:

```
FUNCTION USER RETURN VARCHAR2 IS

C VARCHAR2(225);

BEGIN

 SELECT USER INTO C

 FROM SYS.DUAL;

 RETURN C;

END;
```

Veremos que la declaración de una variable dentro del package se transformará en un cache almacenando el nombre del usuario:

```
CREATE OR REPLACE PACKAGE BODY pkg_paquete

IS

name VARCHAR2(225);


FUNCTION USER RETURN VARCHAR2 IS

BEGIN

   IF name is null THEN

      SELECT USER INTO name

      FROM SYS.DUAL;

   END IF;

   RETURN name;

END user;

END pkg_paquete;
```

Ahora podremos referenciar el valor de USER evitando múltiples llamadas a la función...

```
FOR user_rec IN user_cur

LOOP

 IF user_rec.user_name = paquete.user

 THEN
```

...

d) Beneficios

Incrementar la performance en las aplicaciones evitando accesos innecesarios y minimizándolos a través de la SGA

e) Desafíos

Cada sesión tiene su propia copia de datos del package, hay que planificar de forma correcta qué volumen de datos y cuántos usuarios los utilizarán como cache.

4.030.21 Llamada a procedimientos con parámetros con nombre

a) Regla

En la llamada a procedimientos los parámetros deben pasarse por nombre.

b) Motivación

Esto tiene la ventaja de que el código es más fácil de interpretar, ya que no hace falta referirse a la función y/o procedimiento llamado para conocer el significado de los parámetros.

c) Ejemplo

```
Sumar (
      valor1 => 1,
      valor2 => 2)
```

d) Beneficios

Permite cambiar el orden de los parámetros.

Permite obviar parámetros, si es necesario, que tienen valores por default.

e) Desafíos

El código se hace más extenso.

4.030.22 Efectos Laterales

a) Regla

Evitar los efectos laterales en los programas.

b) Motivación

Diseñar los programas de la manera que estos tengan un propósito claro y bien definido.

c) Ejemplo

Visualizar información sobre todos los libros dentro de determinado rango…

```
CREATE OR REPLACE PROCEDURE display_book_info (
     Start_in IN DATE, end_in IN DATE )
IS
   BEGIN
      CURSOR cur_book IS
          SELECT *
          FROM books
          WHERE date_published BETWEEN  start_in AND end_in;
BEGIN
    FOR book_rec IN book_cur
```

```
    LOOP
        pl ( book_rec.title  || ' by ' || book_rec.author ) ;

        usage_analysis.update_borrow_history ( book_rec ) ;
    END LOOP;
END;
```

Beneficios

El código podrá se usado con gran confianza, ya que hace lo que dice que hace.

4.030.23 Sentencia RETURN en funciones

a) Regla

Limitar las funciones a contener sólo una sentencia RETURN en la sección de ejecución.

b) Ejemplo

Función con múltiples sentencias RETURN

```
CREATE OR REPLACE FUNCTION fn_status_desc ( cd_in VARCHAR2 )
RETURN VARCHAR2
IS
BEGIN
    IF cd_in = 'C'
       THEN RETURN 'CLOSED';
    ELSIF cd_in = 'O'
       THEN RETURN 'OPEN';
    ELSE RETURN 'INACTIVE'
    END IF;
END;
```

Función con una sola sentencia RETURN

```
CREATE OR REPLACE FUNCTION fn_status_desc ( cd_in VARCHAR2 )
IS
  BEGIN
     IF cd_in = stdtypes_c_closed_abberv
       THEN retval := stdtypes.c_closed;
     ELSIF cd_in = stdtypes_c_open_abberv
       THEN retval := stdtypes.c_open;
     ELSE retval := stdtypes.c_inactive;
     END IF;
     RETURN retval;

END;
```

c) Beneficios

Una función con una sola sentencia RETURN es más fácil de debuguear y seguir, de este modo no hay que preocuparse por múltiples salidas de la función.

4.030.24 Evitar parámetros [IN] out en funciones

a) Regla

No usar parámetros para retornar valores en funciones.

b) Motivación

El objetivo principal de una función es retornar un solo valor, si se retorna más de un valor con parámetros IN OUT se está oscureciendo el propósito de la función.

c) Ejemplo

Función que retorna varios pedazos de información

```
FUNCTION fn_book_info (
 isbn_in IN book.isbn%TYPE,
 author_out OUT book.author%TYPE,
 page_count_out OUT book.page_count%TYPE )
RETURN book.title%TYPE;
```

La función se usaría así:

```
BEGIN

   l_title := book_info ( l_isbn, l_author, l_page_count );

Reescribiendo la función para que retorne un registro:

FUNCTION fn_book_info  ( isbn_in book.isbn%TYPE )

RETURN book%ROWTYPE;

El código resultante es más claro:

one_book_rec book%ROWTYPE;

BEGIN

   one_book_rec := book_info ( l_isbn );
```

d) Beneficios

La función puede ser mucho más reusada.

Las funciones con parámetros OUT no podrán ser usadas en SQL.

e) Desafíos

En el caso de retornar información de estado, considere utilizar un procedure en vez de una función.

4.030.25 Evitar implementación de módulos innecesarios

a) Regla

Evitar el crecimiento del volumen de código innecesario en un package o de módulos que son simples de implementar pero que nunca se usaran.

b) Motivación

La implementación de módulos que nunca serán utilizados complica la utilización de aquellos que si son útiles. Construir sólo los necesarios.

c) Beneficios

No se pierde tiempo en construir código que nunca será usado.

Desarrolladores pueden mas fácilmente enterarse de la funcionalidad que actualmente necesitan.

d) Desafíos

Cada sesión tiene su propia copia de datos del package, hay que planificar de forma correcta qué volumen de datos y cuántos usuarios los utilizarán como cache.

4.030.26 Simplificar el uso de módulos con OVERLOADING

a) Regla

Overloading (Sobrecarga) significa que la misma rutina la defino con distinta cantidad y tipos de parámetros (se denomina distintas firmas). Debe simplificarse y promocionarse el uso de módulos con overloading para extender las opciones de llamadas.

b) Motivación

Transferir la "necesidad de saber" del usuario, referido a cómo se usa la funcionalidad, al package en si mismo.

c) Ejemplo

Para evitar el uso de distintos nombres para los procedures DBMS_PUT_LINE, se puede sobrecargar y que cada uno implemente las diferencias:

```
PROCEDURE DBMS_PUT_LINE ( A VARCHAR2 );

PROCEDURE DBMS_PUT_LINE ( A NUMBER );

PROCEDURE DBMS_PUT_LINE ( A DATE );
```

d) Beneficios

Usando sobrecarga apropiadamente, los desarrolladores interpretan el código en forma más simple y rápida.

e) Desafíos

No escribir código de una forma abstracta, pensar cómo el código necesita ser usado. Siempre estar listos a usar sobrecarga en respuesta al feedback del usuario.

4.030.27 Consolidar métodos de OVERLOADING

a) Regla

En caso de usar sobrecarga que la implementación esté consolidada. Esto quiere decir, que todos los procedimientos sobrecargados deben terminar llamando a un solo procedimiento que contiene la lógica real, previamente haciendo las conversiones de tipo necesarios o agregando funcionalidad.

b) Motivación

Evitar la construcción de código difícil de mantener y extender.

c) Ejemplo

Ciertas formas de realizar un procedimiento de insert:

Pasando un valor para cada columna

Pasando un registro, conteniendo un valor para cada columna

Pasando una colección de múltiples registros de datos

Una implementación utilizando sobrecarga podría ser la de especificar métodos ins con los diferentes parámetros posibles, y estos llaman al mismo procedimiento (interno) internal_ins que es el que realiza efectivamente el INSERT.

```
-- Package specification

CREATE OR REPLACE PACKAGE pkg_te_book

IS

   TYPE book_tt IS TABLE OF book%ROWTYPE;


   PROCEDURE prc_ins (

   isbn_in IN book.isbn%TYPE DEFAULT NULL,

   title_in IN book.title%TYPE DEFAULT NULL,

   sumary_in IN book.summary%TYPE DEFAULT NULL,

   author_in IN book.author%TYPE DEFAULT NULL,

   date_published_in IN book.date_published%TYPE DEFAULT NULL,

   page_count_in IN book.page_count%TYPE DEFAULT NULL,

   isbn_in_out IN OUT book.isbn%TYPE );


        --record based insert—

   PROCEDURE prc_ins ( rec_in IN book%ROWTYPE );

   --collection based insert –

   PROCEDURE prc_ins ( coll_in IN book_tt );

END pkg_te_book;
```

```
--Package body
```

```
CREATE OR REPLACE PACKAGE BODY pkg_te_book

IS

   PROCEDURE prc_internal_ins (

   isbn_in IN book.isbn%TYPE DEFAULT NULL,

   title_in IN book.title%TYPE DEFAULT NULL,

   summary_in IN book.summary%TYPE DEFAULT NULL,

   author_in IN book.author%TYPE DEFAULT NULL,

   date_published_in IN book.date_published%TYPE DEFAULT NULL,

   page_count_in IN book.page_count%TYPE DEFAULT NULL )

   BEGIN

   Validate_constraints;

   INSERT INTO book

   ( isbn, summary, author, date_published, page_count )

   VALUES

   ( isbn_in, title_in, summary_in, author_in, date_published_in,
page_count_in );

EXCEPTION

WHEN OTHERS THEN

   err.log;

END prc_internal_ins;
```

```
PROCEDURE prc_ins (

   isbn_in IN book.isbn%TYPE DEFAULT NULL,

   title_in IN book.title%TYPE DEFAULT NULL,

   sumary_in IN book.summary%TYPE DEFAULT NULL,

   author_in IN book.author%TYPE DEFAULT NULL,

   date_published_in IN book.date_published%TYPE DEFAULT NULL,

   page_count_in IN book.page_count%TYPE DEFAULT NULL,

   isbn_inout IN OUT book.isbn%TYPE )


IS

   v_kpy INTEGER := new_isbn_number;

BEGIN

   internal_ins   (   v_pky,   title_in,   summary_in,   author_in,
date_published_in,
page_count_in );

   isbn_inout := v_pky;

END;

PROCEDURE prc_ins ( rec_in IN book%ROWTYPE ) IS

BEGIN

   internal_ins ( rec_in.isbn,

         rec_in.title,

         rec_in.summary,

         rec_in.author,

         rec_in.date_published,

         rec_in.page_count

   ) ;

END;
```

```
PROCEDURE prc_ins ( coll_in IN book_tt ) IS

Indx PLS_INTEGER := coll_in.FIRST;

BEGIN

   LOOP

         EXIT WHEN indx IS NULL

         ...

         --use the record based version

         ins ( coll_in ( indx );

         indx := coll_in.NEXT ( indx );

   END LOOP;

END prc_ins;

END pkg_te_book;
```

d) Beneficios

Al hacer los cambios en un solo lugar, los cambios afectan a todos lo procesos sobrecargados.

e) Desafíos

Desarrollar esta disciplina requiere tomarse el tiempo de identificar áreas con funcionalidades comunes y aislar éstas en sus propios programas.

4.040 Estructuras de control

4.040.1 Uso del ELSIF

a) Regla

Usar el constructor ELSIF con cláusulas mutuamente exclusivas.

b) Motivación

Si se escribe una lógica donde una cláusula es TRUE y las demás se evalúan con algo distinto a TRUE usar el constructor ELSIF.

c) Ejemplo

En vez de escribir

```
PROCEDURE prc_process_lineitem (line_in IN INTEGER)
IS
BEGIN
```

```
   IF line_in = 1 THEN
       process_line1;
   END IF;
   IF line_in = 2 THEN
       process_line2;
   END IF;
   ...
   IF line_in = 2045 THEN
       process_line2045;
   END IF;
END;
```

Se debería reescribir de esta manera

```
PROCEDURE prc_process_lineitem (line_in IN INTEGER)
IS
BEGIN
 IF line_in = 1 THEN
     process_line1;
 ELSIF line_in = 2 THEN
     process_line2;
      ...
 ELSIF line_in = 2045 THEN
     process_line2045;
 END IF;
END;
```

d) Beneficios

La estructura expresa la realidad de la lógica de negocio: si una condición es TRUE, ninguna otra puede ser TRUE.

Este constructor ofrece la más eficiente implementación para procesar cláusulas mutuamente exclusivas. Cuando una cláusula es evaluada TRUE, las demás son ignoradas.

4.040.2 Uso del IF - ELSIF

a) Regla

Usar IF…ELSIF sólo para testear una sola y simple condición.

b) Motivación

Escribir un código que exprese de la mejor forma la complejidad de la vida real en la aplicación.

c) Beneficios

El código resultante es fácil de leer y mantener.

Dividiendo una expresión en pequeñas piezas incrementa el buen mantenimiento, y cuando la lógica cambia, sólo se debe modificar una sola cláusula IF sin afectar las demás.

d) Desafíos

Evitar múltiple niveles de sentencias IF anidadas, ya que disminuye la legibilidad.

4.040.3 Sentencias IF y expresiones booleanas

a) Regla

Reemplazar y simplificar sentencias IF con expresiones booleanas. Además, no usar IF boolvar = true then.

b) Motivación

Evitar escribir código con sentencias condicionales difíciles de comprender. En cambio utilizar variables booleanas con nombres nemotécnicos para expresar la semántica de la expresión.

c) Ejemplo

Considerar la siguiente sentencia condicional:

```
IF hiredate < SYSDATE
THEN
    date_in_past := TRUE;
ELSE
    date_in_past := FALSE
END IF;
```

La anterior sentencia se puede reemplazar por:

date_in_past := (hiredate < SYSDATE);

d) Beneficios

El código es más legible y expresivo.

4.040.4 Evitar sentencias EXIT y RETURN en bloques LOOP

a) Regla

Nunca incluir sentencias EXIT o RETURN dentro de bucles WHILE y FOR.

b) Motivación

Estas estructuras de iteración ya tienen su propio método para determinar cuando salir del bucle. Si EXIT o RETURN son incluidos, el código es mucho más difícil de debuguear y seguir.

c) Beneficios

El código es más fácil de entender y debuguear.

4.040.5 Utilizar FOR y WHILE

a) Regla

Utilizar solamente FOR y WHILE. Usar un sólo EXIT en LOOP simples de REPEAT UNTIL.

b) Motivación

PL/SQL no tiene la estructura de control REPEAT UNTIL. Sin embargo en algunas ocasiones, esta estructura de control es útil. En estos casos, deberá emularse REPEAT UNTIL mediante un loop incondicional y un EXIT WHEN… único al final del loop.

Para el resto de los casos debe utilizarse FOR o WHILE.

c) Beneficios

Un solo EXIT es especialmente importante en cuerpos de LOOP complejos y largos; esto permite que el código sea más fácil de seguir y debuguear.

d) Desafíos

Exige que el código sea reestructurado de una forma correcta, dependiendo como este escrito el código original.

4.040.6 Indices de bucles FOR

a) Regla

Nunca declarar las variables de índices del bucle FOR.

b) Motivación

El motor de PL/SQL declara implícitamente estas variables. La declaración explicita de las mismas puede causar confusión.

c) Beneficios

Evita escribir código extra.

Simplifica el mantenimiento y comprensión del código.

4.040.7 Iteración de colecciones

a) Regla

Iterar sobre colecciones utilizando FIRST, LAST y NEXT dentro de bucles.

b) Motivación

Si se trata de explorar una colección con un bucle FOR y la colección es escasa (no densa), el bucle FOR trata de acceder a una fila indefinida y levanta una excepción

NO_DATA_FOUND. En este caso utilizar FIRST, NEXT, LAST and PRIOR para explorar la colección.

c) Beneficios

La exploración de la colección se realizará con menos excepciones

Es un método mucho más eficiente para explorar una colección.

4.040.8 Expresiones estáticas y bucles

a) Regla

No incluir expresiones estáticas dentro de un bucle.

b) Motivación

Un error común es incluir el cálculo de expresiones estáticas dentro de un bucle, esto da como consecuencia que esta expresión se calcule para cada iteración.

c) Beneficios

El código no realiza trabajo innecesario y de esta manera es más eficiente.

4.040.9 No utilizar GOTO

a) Regla

Evitar de incluir sentencias GOTO.

b) Motivación

La inclusión de esta sentencia es una característica de la programación no estructurada, y da como consecuencia que el código sea difícil de analizar y debuguear.

c) Beneficios

Siempre se puede obtener el mismo efecto con programación estructurada y además es más fácil de entender el uso de condiciones y bucles lógicos.

4.050 Triggers

4.050.1 Tamaños de los triggers

a) Regla

Minimizar el tamaño de la sección ejecutable de los triggers.

b) Ejemplo

Trigger que expone una regla de negocio que debería ser oculta

```
CREATE OR REPLACE TRIGGER trg_check_emplyee_age

BEFORE UPDATE OR INSERT ON  t_employee
BEGIN
 IF ADDMONTHS ( SYSDATE, -216 ) < NEW.hire_date
    THEN
        RAISE_APPLICATION_ERROR  (  -20706,  'employees  must  be   18
years old to work ');
 END IF;
END;
```

Una mejor implementación sería

```
CREATE OR REPLACE TRIGGER trg_check_employee_age
BEFORE UPDATE OR INSERT ON t_employee
BEGIN
 IF employee_rules.emp_too_toung ( :NEW.hire_date )
    THEN
        err_pkg.raise    (    employee_rules.c_err_num_emp_too_young,
:NEW.employee_id );
 END IF;
END;
```

c) Beneficios

Mantener el código de los triggers pequeños provee un código modular que es más simple de mantener y debuguear.

Reduce la posibilidad de incluir reglas de negocio redundantes.

d) Desafíos

Al nivel de filas de los triggers, se puede pasar como parámetro los pseudo registros :NEW y :OLD.

4.050.2 Reglas de negocios definidas en triggers

a) Regla

Validar reglas de negocio complejas mediante DML triggers.

b) Motivación

Triggers permiten evaluar lógica compleja, como el estado de una transacción; lo cual es más complicado realizarlo con una simple constraint.

c) Ejemplo

Si la transacción de cuenta fue aprobada, ésta no puede ser actualizada:

```
CREATE TRIGGER trg_cannot_change_approved
BEFORE UPDATE ON t_account_transaction
FOR EACH ROW
BEGIN
 IF :OLD.approved_yn = constants.yes
 THEN err_pkg.raise(account_rules.c_no_change_after_approval);
 END IF;
END;
```

Transacciones de cuentas no pueden ser creadas con estado de aprobado:

```
CREATE TRIGGER trg_cannot_create_aproved
BEFORE INSERT ON t_account_transaction
FOR EACH ROW
BEGIN
  IF :NEW.approved_yn = 'Y'
  THEN
     err_pkg.raise(account_rules.c_no_preapproval);
  END IF;
END;
```

d) Beneficios

La planificación y diseño cuidadoso permite siempre que las más complejas reglas de negocio sean validadas en DML triggers.

Las interfaces de usuarios no requieren que estas validaciones sean creadas.

4.050.3 Valores de columnas derivadas

a) Regla

Insertar datos en columnas de valores derivados con triggers.

b) Motivación

Algunas aplicaciones necesitan almacenar información extra en el momento que un registro es insertado, modificado o eliminado. Esto puede o no ser realizado por la misma aplicación.

c) Ejemplo

La fecha de modificación de un registro es almacenada dentro del mismo registro:

```
CREATE OR REPLACE TRIGGER trg_set_update_fields
BEFORE UPDATE ON t_account_transaction
FOR EACH ROW
BEGIN
 IF :NEW.updateed_date is NULL
 THEN
   :NEW.updated_date := SYSDATE;
 END IF;
END;
```

d) Beneficios

Se asegura que los campos sean insertados ya que todos lo registros son procesados por el trigger.

e) Desafíos

Si existen columnas que reciben valores por medio de triggers, están no deberían ser incluidas en sentencias DML. Están tendrían más sentido sólo para construir vistas de tablas bases que ocultan los valores de columnas derivadas.

4.060 Manejo de cursores

4.060.1 Cursores

a) Regla Estándar de Programación PL/SQL

Parametrizar los cursores evitando la utilización de bind variables globales y hardcoding de condiciones de filtrado.

b) Motivación

Facilitar la reutilización de los cursores dentro del código, como dentro la memoria de instancia de la base de datos, reduciendo los requerimientos de mantenimiento.

c) Ejemplo

En lugar de:

```
DECLARE

          CURSOR cur_departamentos
          IS
          SELECT       id_departamento,
                        codigo_departamento,
                        nombre_departamento
          FROM         srh_departamentos
```

```
        WHERE codigo_departamento = 'FINANCIAS';
BEGIN
        OPEN cur_departamentos
```

Mover el cursor a un paquete y especificar parámetros:

```
 CREATE OR REPLACE PACKAGE pkg_srh_departamentos
   IS
        -- cursor que permite recuperar el departamento que
corresponde al
        -- código de departamento especificado.
        -- si no se especifica ningun codigo recupera todos los
departamentos

        CURSOR cur_departamentos(
               p_c_cod_departamento
srh_departamentos.codigo_departamento%TYPE
               )
        IS
        SELECT       id_departamento,
                     codigo_departamento,
                     nombre_departamento
        FROM         srh_departamentos
        WHERE (
                      p_c_cod_departamento IS NULL
                      OR
                            (
                            p_c_cod_departamento IS NOT NULL
                            AND
                            codigo_departamento                  =
p_c_codigo_departamento
                            )
                      );
```

Y abrirlo como este ejemplo:

```
 BEGIN
        -- apertura para recuperar solo el departamento con el
código solicitado
        OPEN srh_departamentos_pkg.cur_departamentos(
               p_c_cod_departamento => p_cod_departamento
               );

        …..

        -- apertura para recuperar todos los departamentos
        OPEN srh_departamentos_pkg.cur_departamentos (
               p_c_cod_departamento => NULL

               );
```

Es conveniente declarar cursores generales en los paquetes para que puedan ser reutilizados no solo dentro del contexto del paquete sino fuera de los mismos. De esta

manera, además promover la reutilización es posible centralizar la declaración de cursores de uso general y facilitar el mantenimiento de los mismos.

4.060.2 Usar registros para fetch de cursores

a) Regla

En cursores, hacer el fetch sobre un registro (definido con %rowtype) y no sobre listas de variables.

b) Motivación

Es peligroso explicitar las variables y tipos retornados por un cursor en variables individuales, porque ante un cambio del cursor (por ejemplo, agregado de una columna) el código fallará.
Usando registros, si el cursor cambia, puede recompilarse el código y automáticamente se adapta a la nueva definición del cursor.

c) Ejemplo

Suponer la siguiente declaración de un cursor en un package:

```
PACKAGE pkg_book
IS
    CURSOR books_by_category (category_in book.category%TYPE);
    IS
        SELECT title, author FROM book
        WHERE category = category_in;
END pkg_book;
```

En vez de usar variables individuales:

```
DECLARE
    l_title book.title%TYPE;
    l_author book.author%TYPE;
BEGIN
    OPEN book_pkg.books_by_category('SCIFI');
    FETCH book_pkg.books_by_category INTO l_title, l_author;
    …
```

Escribir el código usando variables de cursores:

```
DECLARE
    scifi_rec book_pkg.books_by_category%ROWTYPE;
BEGIN
    OPEN book_pkg.books_by_category ('SCIFI');
    FETCH book_pkg.books_by_category INTO scifi_rec;
    …
```

d) Beneficios

El código adapta automáticamente a los cambios en la estructura del cursor

Se escribe menos código, no es necesario definir variables individuales

4.060.3 Utilizar cursor FOR-LOOP para procesar cursores

a) Regla

Usar FOR para manejar cursores incondicionales.

b) Motivación

Hacer un código más legible y evitar de escribir código extra.

c) Ejemplo

Visualizar la cantidad total de libros de “FEUERSTEIN, STEVEN” vendidos:

```
DECLARE
CURSOR sef_books_cur IS
SELECT title, total_count FROM book_sales
WHERE author = 'FEUERSTEIN, STEVEN';
BEGIN
FOR rec IN sef_books_cur LOOP
pl ( rec.title || ': ' || rec.total_count || ' copies' );
END LOOP;
END;
```

d) Beneficios

Salvar el esfuerzo de codificar la apertura, búsqueda (fetch) y cierre de los cursores. Código resultante más legible.

e) Desafíos

Luego del END LOOP no puede saberse nada acerca del cursor o del proceso realizado. Si fuera necesario información adicional (como filas procesadas) debe mantenerse mediante variables adicionales actualizadas en el bloque LOOP.

Si el desarrollador no es cuidadoso, el código dentro del LOOP puede ser muy extenso.

4.060.4 No usar cursor FOR-LOOP para fetch de una fila

a) Regla

No usar nunca cursores que retornen una única fila con un FOR loop.

b) Motivación

Un cursor FOR-LOOP es menos eficiente que un SELECT-INTO o un OPEN-FETCH-CLOSE cuando se trata de recorrer sólo una fila.

c) Ejemplo

```
CREATE OR REPLACE FUNCTION fn_book_title ( isbn_in IN book.isbn%TYPE
)
RETURN t_book.title%TYPE
IS
CURSOR title_cur IS
SELECT title INTO l_title

         FROM  t_book
                WHERE isbn = isbn_in;
   l_rec  title_cur%ROWTYPE;
```

En vez de:

```
BEGIN
   FOR rec IN title_cur LOOP
         l_rec := rec;
   END LOOP;

   RETURN l_rec.title;
END;
 Usar SELECT INTO o cursor explícito:
BEGIN
   OPEN title_cur;
   FETCH title_cur INTO l_rec;
   CLOSE title_cur;
   RETURN l_rec.title;
END;
```

d) Beneficios

El código de esta forma satisface el requerimiento de la forma más directa y clara.
Un cursor FOR-LOOP es menos eficiente que un SELECT-INTO o un cursor explícito para retornar una fila

4.060.5 Especificar columnas a actualizar en SELECT FOR UPDATE

a) Regla

Especificar las columnas a actualizar en SELECT FOR UPDATE.

b) Motivación

Lock de las filas a ser actualizadas. Evitar que otra sesión cambie las filas afectadas por el cursor.

c) Ejemplo

-- Actualizar el sabor de helado preferido por la familia PEREZ

```
DECLARE
CURSOR change_prefs_cur IS
                SELECT PER.name, PREF.name flavor
                FROM person PER, preference PREF
WHERE PER.name = PREF.person_name
AND PREF.type = ' HELADO'
FOR UPDATE OF PREF.name;
BEGIN
     FOR rec IN change_prefs_cur
     LOOP
          IF rec.name LIKE 'PEREZ' THEN
              UPDATE preference SET name = 'CHOCOLATE'
              WHERE CURRENT OF change_prefs_cur;
          END IF;
     END LOOP;
END;
```

d) Beneficios

Mantener la mínima cantidad de locks sobre una tabla
Auto documentar el comportamiento del código

4.060.6 Parametrizar cursores explícitos

a) Regla

Parametrizar los cursores explícitos (evitar hardcode en los where).

b) Motivación

Posibilita que el cursor sea mas fácilmente reusado en diferentes circunstancias y programas, mayormente cuando se emplean definiciones de cursores en paquetes.

c) Ejemplo

En lugar de:

```
DECLARE
   CURSOR r_and_d_cur IS
         SELECT last_name FROM employee
         WHERE department_id = 10;
BEGIN
   OPEN r_and_d_cur;
```

Mover el cursor a un paquete:

```
 CREATE OR REPLACE PACKAGE pkg_dept_info
   IS
         CURSOR name_cur (dept IN INTEGER) IS
                SELECT last_name FROM employee
                WHERE department_id = dept;
```

Y abrirlo como este ejemplo:

```
BEGIN
        OPEN pkg_dept_info.name_cur (10);
```

O mejor aún, evitar harcode del literal:

```
DECLARE
   r_and_d_dept CONSTANT PLS_INTEGER :=10;
BEGIN
   OPEN pkg_dept_info.name_cur ( r_and_d_dept );
  ...
```

d) Beneficios

Mantener un nivel alto de reusabilidad en las aplicaciones, reduciendo los requerimientos de mantenimiento.

Mejoras en la aplicación, ya que los parámetros de los cursores son tratados como binded variables y no necesita parsearlo repetidas veces.

4.060.7 Uso del ROWCOUNT

a) Regla

Muchas veces es necesario verificar la cantidad de registros modificados por una DML. En estos casos, debe utilizarse SQL%ROWCOUNT.

b) Motivación

Asegurarse que el DML se haya ejecutado apropiadamente. Ya que en un UPDATE o DELETE no levanta excepción si ninguna fila es afectada.

c) Ejemplo

```
BEGIN
 UPDATE t_book
 SET author = 'PEREZ, PEDRO'
 WHERE author = 'PEREZ, JUAN';
 IF SQL%ROWCOUNT < 8 THEN
  ROLLBACK;
 END IF;
END;
```

d) Beneficios

Los programas son verificados y están mejor habilitados para manejar problemas más eficientemente.

4.060.8 Definir cursores de múltiples filas en packages

a) Regla

Definir cursores de múltiple filas en packages. Además de empaquetar las queries, los cursores deben estar documentados en la especificación del package.

b) Motivación

Compartir cursores definidos a través de distintos programas

c) Ejemplo

Definicion del Package...

```
PACKAGE pkg_book
IS
CURSOR allbooks  IS
         SELECT * FROM books;
CURSOR books_in_category (category_in IN book.category%TYPE) IS
SELECT * FROM books
WHERE category = category_in;

Uso del Package...
BEGIN
OPEN pkg_book.books_by_category('THRILLER');
LOOP
FETCH pkg_book.books_by_category INTO thiller_rec;
…
END LOOP;
CLOSE pkg_book.books_by_category;
END;
```

d) Beneficios

Sólo se debe escribir la query en un solo lugar.

Reusabilidad del código.

e) Desafíos

Cerrar cursores explícitamente. Los cursores de paquetes son persistentes, y permanecen abiertos hasta cerrarse explícitamente o hasta la desconexión de la sesión.

Esto es diferente a cursores definidos localmente que se cierran al finalizar el bloque actual.

Se debe definir con el equipo de desarrollo un proceso para encontrar el código necesario y a su vez modificar la definición de estos cursores

4.070 SQL dentro de PL/SQL

4.070.1 Autonomous Transactions

a) Regla

Utilizar Autonomous Transactions para aislar los efectos de COMMITs y ROLLBACKs.

b) Motivación

Realizar y salvar cambios en la base sin afectar a la transacción principal o "llamadora".

c) Ejemplo

Para convertir un bloque PL/SQL como Autonomous Transaction sólo se debe incluir la instrucción remarcada en la sección de declaración:

```
CREATE OR REPLACE  PROCEDURE prc_log_error (
                code IN INTEGER, msg IN VARCHAR2)
        AS
                PRAGMA AUTONOMOUS_TRANSACTION;
        BEGIN
                INSERT INTO error_log
                (errcode, errtext, created_on, created_by);)
                VALUES
                (code, msg, SYSDATE, USER);
                COMMIT;
        EXCEPTION
                WHEN OTHERS THEN ROLLBACK;
        END;
```

d) Beneficios

Escribir y salvar mensajes en tablas de log en la base de datos sin afectar la transacción principal.

Ejecutar funciones PL/SQL que cambian la base de datos.

Escribir componentes PL/SQL que se comporten satisfactoriamente en ambientes distribuidos.

4.070.2 Encapsular consultas de una fila en funciones

a) Regla

Colocar las consultas que devuelven sólo una fila en funciones, y luego llamar a dicha función para retornar la información.

b) Motivación

Evitar harcode de estas queries en los bloques de código

c) Ejemplo

En lugar de escribir:

```
BEGIN
   SELECT title INTO l_title
   FROM book
   WHERE isbn = isbn_id
   …
```

Crear una función:

```
 PACKAGE pkg_te_book
IS
   FUNCTION fn_title (isbn_IN book.isbn%TYPE)
         RETURN book.titel%TYPE;
   …
```

Y ahora la aplicación será similar a:

```
BEGIN
   l_title := pkg_te_book.fn_title (isbn_id);
   …
```

d) Beneficios

Mayor legibilidad del código resultante.

e) Desafíos

Entrenar y disciplinar al equipo de desarrollo para adherir al estándar de encapsulamiento. Se deberá asignar un correcto tamaño de SGA para el manejo de grandes volúmenes de código.

4.070.3 Ocultar el uso de la tabla DUAL

a) Regla

Ocultar el uso de la tabla dual a través de la definición de funciones.

b) Motivación

El uso de la tabla DUAL es una alternativa posible y valedera dentro de muchas más. Encapsular este tipo de alternativas en funciones o procedimientos, de tal manera que si se puede cambiar la implementación por otra alternativa al uso de esta tabla, sea transparente a las aplicaciones.

c) Ejemplo

En vez de escribir:

```
DECLARE
my_id INTEGER;
BEGIN
SELECT patient_seq.NEXTVAL INTO my_id
```

```
FROM dual;
```

Crear una función:

```
CREATE OR REPLACE FUNCTION fn_next_patient_id
   RETURN patient.patient_id%TYPE
IS
   retval patient.patient_id%TYPE;
BEGIN
SELECT patient_seq.NEXTVAL INTO retval
FROM dual;
RETURN retval;
END;
```

Ahora la aplicación se vería de esta manera:

```
DECLARE
my_id INTEGER;
BEGIN
my_id := fn_next_patient_id;
…
```

d) Beneficios

Mayor legibilidad del código.

Cambios en las implementaciones de las funciones no afectan la funcionalidad.

4.070.4 Evitar innecesario uso de COUNT

a) Regla

Usar el COUNT exclusivamente cuando la cantidad actual de ocurrencias es requerida. No usar COUNT para consultar si existen registros que cumplen determinados criterios.

b) Motivación

Evitar trabajo innecesario.

c) Ejemplo

Escribir un programa que retorne TRUE si existe como mínimo un libro en una categoría determinada:

```
CREATE OR REPLACE FUNCTION fn_atleastone (
 category_in IN book.category%TYPE)
IS

   numbooks INTEGER;

BEGIN

   SELECT COUNT(*)
   INTO numbooks
   FROM books
   WHERE category = category_in;

   RETURN (numbooks > 0)
END;
```

Una mejor solución sería:

```
CREATE   OR   REPLACE   FUNCTION   fn_atleastone   (category_in   IN
book.category%TYPE)
IS  retval BOOLEAN;
CURSOR category_cur IS
   SELECT 1  FROM book
   WHERE category = category_in;

BEGIN
   OPEN category_cur;
   FECTH category_cur INTO category_rec;
   retval := category_cur%FOUND;
   CLOSE category_cur;
   RETURN retval;
END;
```

e) Beneficios

Óptima Performance

Buena legibilidad del código

Una traducción más exacta del requisito

f) Desafíos

El desarrollador debe escribir un poco más de código.

4.070.5 Referenciar atributos de cursores inmediatamente después de la operación SQL

a) Regla

Referenciar atributos de cursores inmediatamente después de la operación SQL. Las instrucciones DML son ejecutadas por cursores implícitos en PL/SQL. Los atributos de

cursores reflejan lo realizado en la última operación implícita. Debería mantenerse al mínimo la cantidad de código entre las operaciones DML y la referencia a algún atributo del cursor.

b) Motivación

Obtener información sobre los resultados de la más reciente operación implícita realizada.

c) Ejemplo

```
DECLARE
   PROCEDURE prc_show_max_count
   IS

         l_total_pages PLS_INTEGER;
   BEGIN
         SELECT MAX (page_count) INTO l_total_pages
         FROM book
         WHERE title LIKE '%PL/SQL%';
         DBMS_OUTPUT.PUT_LINE(l_total_pages);
   END;
BEGIN
         UPDATE book SET page_count = page_count / 2
         WHERE title LIKE '%PL/SQL%';
         prc_show_max_count;
         DBMS_OUTPUT.PUT_LINE(' pages adjusted in '|| SQL%ROWCOUNT
|| ' books');
END;
```

En este ejemplo, entre el UPDATE y la referencia a SQL%ROWCOUNT, se ejecuta el cursor implícito del SELECT MAX. El resultado de esto dependerá del cursor del SELECT y no del UPDATE.

4.070.6 Utilizar cláusula RETURNING

a) Regla

Usar RETURNING para retornar información de filas que se modifican.

b) Motivación

Disminuye la cantidad de código generado.

c) Ejemplo

Suponer que se usa una secuencia para generar la PK de una tabla, y luego se necesita ese número para un posterior proceso:

```
INSERT INTO patient
(patient_id, last_name, first_name)
VALUES
(patient_seq.NEXTVAL, 'FEUERSTEIN', 'STEVEN')
RETURNING patient_id INTO l_patient_id;
```

d) Beneficios

Mejora la performance de las aplicaciones.

Reduce el volumen del código.

4.070.7 Usar cláusula BULK COLLECT

a) Regla

Usar BULK COLLECT para mejorar la performance de queries con múltiples filas.

b) Motivación

Necesidad de retornar gran cantidad de filas de la base de datos. Esto retira las filas en un solo pedido al motor de la base.

c) Ejemplo

```
CREATE OR REPLACE PROCEDURE prc_process_employee ( deptno_in
dept.deptno%TYPE)

RETURN emplist_t
IS
   TYPE numTab IS TABLE OF emp.empno%TYPE;
   TYPE chartTab IS TABLE OF emp.ename%TYPE;
   TYPE dateTab IS TABLE OF emp.hiredate%TYPE;
   enos numTab;
   names charTab;
   hdates dateTab;
BEGIN
   SELECT empno, ename, hiredate
   BULK COLLECT INTO enos, enames, hdates
   FROM emp
   WHERE deptno = deptno_in;
   …
END;
```

Si se utiliza un cursor explícito:

```
BEGIN
OPEN emp_cur INTO emp_rec;
   FETCH emp_cur BULK COLLECT INTO enos, enames, hdtaes;
```

d) Beneficios

Mejora la performance de las aplicaciones.

e) Desafíos

Se debe declarar una colección por cada columna de la lista del SELECT.

Se debe ser cuidadoso al usar esto, ya que puede quedarse sin memoria debido al alto número de filas retornadas.

4.070.8 Encapsular sentencias DML en llamadas a procedures

a) Regla

Encapsular los INSERTs, UPDATEs y DELETEs en procedimientos.

b) Motivación

Mayor legibilidad y consistencia en el manejo de errores.

c) Ejemplo

En vez de escribir un INSERT como sigue:

```
INSERT INTO book
 (isbn, title, author)
VALUES (...)
```

Usar un procedure:

```
prc_add_book (...);
```

O un procedimiento de un package:

```
pkg_te_book.prc_ins (...)
```

d) Beneficios

La aplicación se ejecuta mas rápido, por reutilizar el mismo insert, realizando menos parseo y reduce la demanda de memoria SGA.

La aplicación maneja de manera consistente los errores relacionados con DML "

e) Desafíos

Se necesita generar más código procedural.

Se podría necesitar crear procedimientos múltiples de UPDATE.

4.070.9 Usar Bind Variables en SQL Dinámico

a) Regla

No concatenar los valores de variables en SQLs dinámicos, utilizar BIND VARIABLES. De esta manera Oracle parsea la versión generica del SQL, la cual puede ser ejecutada una y otra vez sin importar el valor actual de la variable.

b) Motivación

Evitar el parseo en sucesivas ejecuciones dinámicas con diferentes valores.

c) Ejemplo

Actualizar cualquier columna numérica en la especificada tabla, basados en el nombre pasado como parámetro:

```
CREATE OR REPLACE PROCEDURE prc_updnuval (
   tab_in IN VARCHAR2,
   namecol_in IN VARCHAR2,
   numcol_in IN VARCHAR2,
   name_in IN VARCHAR2,
   val_in IN NUMBER )
   IS
cur PLS_INTEGER;
Fdbk PLS_INTEGER;
BEGIN
   cur := open_and_parse (
         ' UPDATE ' || tab_in ||
         ' SET '          || numcol_in ||
         ' WHERE '  || namecol_in || ' LIKE : name ' ) ;
   DBMS_SQL.BIND_VARIABLE ( cur, ' val ' , val_in );
   DBMS_SQL.BIND_VARIABLE ( cur, ' name ' , name_in );
   fdbk := DBMS_SQL.EXECUTE ( cur );
   DBMS_SQL.CLOSE_CURSOR ( cur );
END;
```

d) Beneficios

La SGA requiere menos memoria para los cursores de los SQL Dinámicos.

La performance de la aplicación se incrementa al reducirse el parseo.

Escribiendo SQL Dinámico se conseguirá una forma más fácil y menos propensa a errores.

4.070.10 Formatear SQL dinámicos

a) Regla

Formatear los strings de los SQL dinámicos para que sean más fáciles de leer y mantener.

b) Ejemplo

Alternativas para formatear el mismo SQL…

```
   v_sql :=
                  ' DECLARE '
   ||                    ' CURSOR curs_get_orders IS '
   ||                          ' SELECT * FROM ord_order; '
   ||             ' BEGIN '
   ||                    ' FOR v_order_rec IN curs_get_orders LOOP '
   ||                          ' process_order ( v_order_rec.order_id
) '
   ||                    ' END LOOP; '
   ||             ' END; ' ;

   v_sql :=
                  ' DECLARE
                        CURSOR curs_get_orders IS
                               SELECT * FROM ord_order;
                   BEGIN
                          FOR v_order_rec IN curs_get_orders LOOP
                               process_order ( v_order_rec.order_id )
;
                          END LOOP;
                   END '  ;
```

c) Beneficios

Leer y mantener el código más fácilmente.

d) Desafíos

Es extremadamente importante convenir un estándar con los desarrolladores para formatear los SQL dinámicos

4.070.11 Optimización basada en costos

a) Regla

Usar optimización basada en costos para SQL o DMLs. Para esto la base de datos no debe estar configurada con OPTIMIZER_MODE=RULE y las tablas deben estar analizadas. Por otro lado, en cuanto a la programación, no está permitido alterar las sesiones a reglas ni usar hints de RULE.

b) Motivación

El optimizador basado en costos genera mejores planes de ejecución que el de reglas. A partir de Oracle 8i el optimizador basado en costos no tiene tantos problemas como en versiones anteriores. El optimizador basado en reglas no está siendo actualizado por Oracle, por lo tanto muchas formas de acceder a los datos no están disponibles por reglas (partition prunning, hash join, bitmap indexes, etc).

4.070.12 No utilizar hints en SQL

a) Regla

No utilizar hint dentro de definiciones de sentencias SQL.

b) Motivación

Usar hints es una forma de hardcoding (en este caso del algoritmo de ejecución de una consulta). Requieren mantenimiento de código. Un plan de ejecución puede ser bueno en un momento y malo en el futuro debido a cambios en la cantidad y calidad de los datos subyacentes. Por ultimo, la feature de Oracle 9i u 8i "stored outlines" permitirá modificar a nivel de base de datos los planes de ejecución de posibles queries con mala performance si fuera necesario.

4.070.13 Evitar el uso de Sorts

a) Regla

Evitar las operaciones de Sorts siempre que sea posible. Las operaciones que producen Sorts son las siguientes:

Cláusulas ORDER BY o GROUP BY.

Cláusula DISTINCT.

Operadores INTERSECT, UNION, MINUS.

Join Sort-Merge.

Ejecución del comando ANALYZE.

b) Motivación

Las operaciones de Sorts consumen una excesiva cantidad de recursos al eliminarlos se mejora la performance de las aplicaciones.

c) Alternativas

Algunas alternativas para evitar sorts innecesarios:

Usar UNION ALL en lugar de UNION. Esto evita la eliminación de duplicados.

Acceder a tablas por índices. De esta manera el optimizador realizara nested loop join en lugar de sort-merge-join.

Crear índices que contengan los campos utilizados frecuentemente en cláusulas ORDER BY y/o GROUP BY. El optimizador Oracle utilizara el índice en lugar de hacer una operación de Sorts, ya que el índice se encuentra ordenado.

Para tomar estadísticas, usar ESTIMATE en lugar de COMPUTE. Usar DBMS_STATS para tomar estadísticas solo de ciertas columnas, de una tabla determinada.

4.070.14 Uso de Exists vs IN

a) Regla

Usar EXIST en lugar de IN para subquerys, si el predicado de selección esta en el Query principal.

Usar IN en lugar de EXISTS para subquerys, si el predicado de selección esta en el subquery

b) Ejemplo

Uso de IN

```
SELECT *
  FROM tabla1 t1
 WHERE t1.id IN
   (SELECT t2.id
      FROM tabla2 t2
     WHERE t2.name LIKE 'LOPEZ%')
```

Uso de EXISTS

```
SELECT *
  FROM tabla1 t1
 WHERE EXISTS
     (SELECT 1
        FROM tabla2 t2
       WHERE t1.id = t2.id)
   AND t1.name LIKE 'LOPEZ%'
```

c) Beneficios

Mejorar performance de SQLs cuando se deben utilizar Subquerys.

4.080 Seguridad

4.080.1 Asignación de privilegios mediante roles

a) Regla

Manejar el uso de priviliegios mediante roles evitando los permisos directos.

b) Motivación

De esta manera se evita que en la creación o eliminación de usuarios se olvide asignar o revocar un permiso con los potenciales problemas de seguridad que acarrea esta situación.

c) Ejemplo

Imaginar una situación con 100 objetos y 100 usuarios, suponiendo que todos tienen acceso a todos los objetos por cada usuario que se cree/elimine se deben crear/eliminar 100 permisos, lo mismo para cada objeto que se cree/elimine.

d) Beneficios

Reduce drásticamente la cantidad de revocaciones y asignaciones de permisos.

Elimina potenciales problemas de seguridad a la hora de crear/eliminar usuarios.

4.080.2 Asignar roles de acuerdo a niveles de autorización para cada aplicación

a) Regla

Manejar el uso de privilegios mediante roles evitando accesos indebidos.

b) Motivación

Al hacer uso de los roles, los usuarios tienen accesos a la minima cantidad de objetos de la base de datos necesarios para desarrollar sus tareas en la organizacion, por tanto al crear un rol para cada nivel de autorización se logran reducir los accesos indebidos.

c) Ejemplo

Suponiendo que se tiene solo dos areas en una empresa, una de contabilidad y otra operativa, se podra crear un rol contable con permisos en las aplicaciones de balances y otro operativo con permisos en las aplicaciones de stock, evitando de esta manera que un usuario operativo tenga acceso a los balances contables.

4.080.3 Crear un paquete de acceso a cada tabla

a) Regla

Por cada tabla de acceso frecuente, crear un paquete para las operaciones en la misma.

b) Motivación

El acceso a las tablas deberá hacerse mediante un paquete que maneje el acceso a la misma cuando se necesite acceder a la misma. Por tanto las aplicaciones que necesiten acceder a la tabla deberan hacer uso de un paquete específico diseñado para insertar, borrar y buscar en dicha tabla.

4.080.4 Nunca asignar privilegios de acceso a tablas

a) Regla

Nunca se debe asignar ni privilegios ni roles a tablas.

b) Motivación

El acceso a tablas se debe hacer solo mediante el uso de paquetes, procedimientos y/o funciones. Por tanto serán estos últimos los que permitirán los accesos a las tablas.

c) Beneficios

Se restringe el acceso a tablas mediante procedimientos, paquetes y funciones.

Se separa el modelo lógico del modelo de datos.

4.080.5 Usar el usuario del esquema para desarrollo

a) Regla

Solo el usuario dueño del esquema deberá ser utilizado para desarrollo de aplicaciones.

b) Motivación

Solo un usuario, el dueño del objeto es el que tendrá todo tipo de permisos sobre el mismo. Este usuario deberá ser utilizado por el equipo de desarrollo de la aplicación y NUNCA por usuarios finales. La necesidad de este usuario radica en que para compilar y/o crear objetos es necesario tener permisos sobre esquemas.

c) Beneficios

Se evita por completo que los usuarios finales accedan a objetos restringidos por sus roles.

Recordar que la aplicación se ejecutara con los privilegios del dueño.

4.080.6 Minimizar el número de privilegios por defecto

a) Regla

Minimizar el uso de privilegios para los usuarios creados por defecto.

b) Motivación

La existencia de privilegios innecesarios para todos los usuarios creados provoca potenciales accesos indebidos. Los privilegios que el usuario creado necesite deberán ser asignados mediante un rol que satisfaga sus requerimientos de accesos.

4.090 Escalabilidad

4.090.1 Acceso masivo a datos

a) Regla

Siempre se debe evitar el acceso masivo a datos en aquellas instancias Oracle que están destinadas a la transacción unitaria y a la operación del negocio, ya sea a través de aplicaciones desarrolladas o productos de acceso a datos.

b) Motivación

Normalmente una base de datos está catalogada como transaccional o histórica, y en el primer caso la SGA está configurada de tal manera que de prioridad a la ejecución de procedimientos cortos, muy volátiles y de bajo impacto. Este no es el caso de una base de datos histórica, y combinar ejecuciones de un tipo en una base de datos configurada para algo distinto tiene graves consecuencias en el rendimiento de la instancia.

c) Desafío

Mover los datos históricos o de análisis a instancias de base de datos adecuadas para el trabajo con volumen de datos masivos.

4.0100 Trazabilidad

4.0100.1 Realizar o utilizar un plan de trazabilidad para cada organización y/o sistema.

a) Regla

Siempre debe existir un plan de trazabilidad para cada organización.

b) Motivación

La existencia y apego a un plan de trazabilidad de la organización evita que la trazabilidad sea inconsistente, redundante, irrelevante, excesiva y/o deficiente.

Lo primero que se debe definir en dicho plan es el objetivo y el alcance de la trazabilidad. En dicho plan también deben figurar las normas sobre que artefactos, usuarios, aplicaciones y sistemas deben trazarse. Además detalla que datos serán relevantes en cada uno, detalla las estructuras de almacenamiento de la traza así como el detalle de su implementación.

Además se debe detallar los componentes usados para trazar como sean paquetes, módulos de aplicación, auditoria automatizada, etc.

La especificación de las estrategias de trazabilidad debe ser detallada en el plan de trazabilidad.

c) Desafíos

Encontrar la cantidad de información adecuada para trazar es una tarea bastante crítica, puesto que si se traza demasiado puede perjudicarse el rendimiento. Y si se traza de menos, el plan no alcanzara el objetivo.

4.0100.2 Usar esquema monousuario y multisesión para desarrollo de aplicaciones

a) Regla

Utilizar una sesión por cada usuario de aplicación.

b) Motivación

La utilización de una sesión (conexión) de base de datos por cada usuario de aplicación, permite acceder a los beneficios del contexto único por usuario de negocio, por lo que cada usuario es identificado de manera única por el gestor de base de datos.

Esta regla tiene como objetivo que todas las aplicaciones utilicen una conexión a la base de datos para cada usuario de las aplicaciones.

c) Ejemplo

Por tanto cada si se tienen varios usuarios a nivel de aplicación como son:

Usuario_1, Usuario_2, Usuario_3, …., Usuario_N.

Y a su vez la aplicación se conecta a un único usuario de la base de datos:
UsuarioBD_A
Entonces para todas las actividades de cada usuario de aplicación se deberá hacer una conexión a UsuarioBD_A.

d) Beneficios

Se permite que cada usuario pueda usar el contenido de las variables de paquete.

Se evita que otros usuarios accedan al contenido de las variables de sesión de otro paquete.

Permite usar triggers para realizar la trazabilidad identificando unívocamente cada usuario.

Permite obtener datos extras acerca del usuario mediante la sesión, como son el nombre del ordenador, la red por la cual se accedió, la IP, etc.

Permite usar paquetes, con procedimientos y funciones para trazar el desarrollo de las distintas invocaciones.

Permite usar parte importante de los mecanismos automatizados de oracle para la auditoria.

e) Desventajas

Al no usar un esquema multiusuario se pierde parte de la información que brinda la opción de auditoria de ORACLE.

f) Desafíos

Se debe tener en cuenta un plan para efectuar la trazabilidad a fin de no afectar en forma significativa el rendimiento.

Es recomendable que las aplicaciones utilicen un manejador de acceso a la base de datos.

4.0100.3 Usar un paquete especial para la trazabilidad

a) Regla

Utilizar un paquete especialmente diseñado para la trazabilidad.

b) Motivación

Centralizando el proceso de trazado, se evita que este sea inconsistente, redundante, irrelevante, excesivo y/o deficiente, puesto que todos la trazabilidad se deberán hacer mediante interfaces bien definidas.

Ayuda a especificar la estrategia de trazabilidad mediante el uso de interfaces.

Dicho paquete deberá contar con estructuras (variables de paquete) para que puedan ser usados en cada sesión.

Permite limitar y controlar el acceso y la escritura de la información de trazabilidad.

4.0100.4 Sincronizar la sesión de aplicación con la sesión de base de datos

a) Regla

Se debe sincronizar siempre que lo amerite, la información de sesión de la aplicación con la información con la información de la sesión de la base de datos.

b) Motivación

Esta regla se refiere a la que información y con que frecuencia debe realizarse la sincronización de la información de usuario, transacción, etc, con las variables de paquete de la sesión de la base de datos.

Ese proceso debe realizarse con la frecuencia especificada en el plan, por ejemplo cada vez que el usuario inicie una sesión, así como cada vez que se inicie una nueva transacción a nivel de aplicación, etc.

Esta tarea es necesaria para poder sincronizar la traza de la base de datos con las operaciones de aplicación.

También ayuda a asociar la sesión de base de datos con la sesión de aplicación para las trazas realizadas por el motor de base de datos en forma automática (Oracle Audit Trails).

c) Beneficios

Permite sincronizar la traza de la base de datos con las transacciones de aplicación.

Permite usar el enorme potencial de la auditoria de Oracle para la el beneficio de la organización.

d) Desafíos

Dado que al gestor de Oracle le resulta imposible reconocer automáticamente la información de aplicación, es necesario que dicha información sea sincronizada a nivel de aplicación.

Es recomendable que las aplicaciones utilicen un manejador de acceso a la base de datos encargado de dicha actividad.

4.0110 Documentación de PLSQL

En esta sección se detallan las reglas de formato de documentación de objetos PLSQL.

Para documentar objetos PLSQL se deben seguir algunas reglas que permitan utilizar una herramienta de parseo y generación automática de documentación en HTML llamada NaturalDocs. Para documentar un objeto PLSQL se deben seguir las normas siguientes:

4.0110.1 Objetos a documentar

Los objetos a documentar son packages, procedimientos, funciones, parámetros de entrada y salida y tipos globales de un package. Para cada uno de estos objetos se debe documentar la funcionalidad que cumplen (packages, procedimientos, funciones) o una descripción sobre qué representan (parámetros de entrada y salida, tipos globales).

4.0110.2 Objetos a documentar

La documentación referida a descripciones de objetos o sus funcionalidades debe hacerse en el encabezado (spec) del package a documentar. Documentación adicional, por ejemplo, indicaciones en el desarrollo del código fuente, deben hacerse en el cuerpo (body) del package

4.0110.3 Documentación en código fuente

La documentación se debe realizar en el mismo código fuente y siempre antes del objeto a documentar. No se debe documentar código que se encuentre, físicamente en el archivo, en una posición superior a la documentación.

Por ejemplo:

Documentación objeto 1
Código fuente Objeto 1

Documentación objeto 2
Código fuente objeto 2

Documentación objeto 3
Código fuente objeto 3

4.0110.4 Keywords permitidos

Para documentar se deben utilizar sólo los siguientes keywords:

- Package: se utiliza para describir el objetivo del pakage, funcionalidades que entrega y sistemas que utilizan el package
- Function: se utiliza para describir la funcionalidad de una función
- Procedure: se utiliza para describir la funcionalidad de un procedimiento
- Parameters: se utiliza para describir los parámetros de entrada o salida de los procedimientos o funciones
- Types: utilizado para describir los parámetros o tipos globales de un package
- Section: utilizado para describir ejemplos de ejecución de procedimientos o funciones del Packaged
- Keyword para documentar código fuente: se utiliza para documentar código fuente. Se puede utilizar en los ejemplos de ejecución
- <LINK> : utilizado para hacer referencias a otros objetos documentados

4.0110.5 Comienzo y término de la documentación

Toda documentación que se realice en el encabezado de un package debe comenzar en una línea, con los símbolos /* y terminar, en otra línea con los símbolos */.

En las líneas de inicio y término de documentación no debe ir otro carácter de algún tipo. Estos símbolos deben ubicarse en la primera columna del archivo

```
/*



*/
```

4.0110.6 Formato de documentación

Cada línea de documentación debe comenzar con un espacio seguido por el símbolo

*.

```
/*
 *
 *
 *
 *
 *
*/
```

4.0110.7 Formato de keywords

Los keywords deben seguir el siguiente formato

```
/* Keyword:
```

Los primeros caracteres deben ser /*

Luego debe haber un espacio seguido del keyword con la primera letra, y sólo la primera letra en mayúscula.

Los keywords deben escribirse con dos puntos para finalizar, es decir, keyword:.

El único keyword que tiene una nomenclatura distinta es el keyword Link.

Por ejemplo:

```
/* Package:
```

```
/* Procedure:
```

```
/* Function:
```

```
/* Types:
```

4.0110.8 Formato de nombres de objetos asociados a keywords

Los objetos identificados como keywords deben nombrarse en la misma línea donde se define el keyword, separado por un espacio entre el nombre y los dos puntos del keyword

```
/* Keyword: Nombre
```

Por ejemplo:

```
/* Package: PKG_LIQUIDACION_PARTICIPES
```

```
/* Procedure: prc_Listado_Modalidades_Cobro
```

```
/* Type: t_cur_ref
```

4.0110.9 Espacios entre keywords

Dos keywords deben ir siempre separados por, al menos, una línea.

```
/* Keyword1:
 *
 * Keyword2:
```

4.0110.10 Formato de nombres de objetos asociados a keywords

Los objetos identificados como keywords deben nombrarse en la misma línea donde se define el keyword, separado por un espacio entre el nombre y los dos puntos del keyword

```
/* Keyword: Nombre
```

Por ejemplo:

```
/* Package: PKG_LIQUIDACION_PARTICIPES_PKG
```

```
/* Procedure: prc_Listado_Modalidades_Cobro
```

```
/* Type: t_cur_ref
```

4.0110.11 Formato de descripciones de objetos asociados a keywords

La descripción de los objetos identificados como keywords deben realizarse en la siguiente línea de la declaración del keyword y del nombre del keyword. La descripción comienza una tabulación después del símbolo * de la línea. Si se necesita escribir en más de una línea, se debe escribir en la misma columna donde se comenzó la primera línea

```
/* Keyword: Nombre
 * Descripción linea 1
 * Continuación descripción linea 2
```

Por ejemplo:

```
/* Package: PCK_LIQUIDACION_PARTICIPES
 * Servicio encargado de entregar el proceso de liquidación para
 * APV y APVC esto en base a una tabla paramétrica
```

4.0110.12 Formato de keyword parameters

EL keyword parameters tiene un tratamiento especial. Se usa para describir parámetros de entrada y salida de procedimientos y funciones.

El keyword parameters no incluye descripción del keyword

La descripción de cada parámetro debe realizarse en líneas separadas entre sí y separada de la línea donde se declara el keyword parameters

El nombre del parámetro comienza una tabulación después del símbolo * de la línea

Antes de enunciar cada parámetro se debe incluir un signo menos (-). Luego se debe dejar un espacio entre el símbolo menos (-) y el nombre del parámetro.

La descripción de cada parámetro se debe hacer en la misma fila donde se enunció el parámetro. Entre el nombre y la descripción del parámetro se debe dejar un espacio, seguido por un signo menos (-) y seguido por otro espacio.

La primera letra de la primera palabra de la descripción debe ser mayúscula

```
/* Parameters:
 * - parametro1 - Descripción parámetro 1
 * - parametro2 - Descripción parámetro 2
 * - parametro3 - Descripción parámetro 3
```

4.0110.13 Formato de keyword parameters

4.0110.14 Formato de links

Si se desea agregar links a otros objetos, se debe ocupar el siguiente formato:

```
<LinkOtraPagina>
```

No se debe hacer referencias a URL. Sólo se debe incluir el nombre del objeto documentado a referenciar. En caso que el objeto sea de otro package, la referencia se debe realizar de la siguiente forma:

```
<NombrePackage.NombreObjeto
```