ORACLE development rules

1. Foreword

Each rule is followed by a number (in brackets) that will not change in time. This number is used as a rule ID. Current maximum rule number is 140.

1.1. Rules validity

From 28.06.2010 (rules added after this date have it's own validity that is individually mentioned)

Rules are mandatory for newly created code or DB objects.

When the current code is modified, then we try to change it in order to meet the requirements as much as we can. Time needed for this adjustment should not be longer than 10% of time estimated for the change of current functionality.

1.2. Types of rules

 readability - following this type of rules increases code readability and uniformity (elimination of individual coding styles)

 effectivity - the rule helps to increase the code efficiency

 maintainability - following this rule has a positive effect on the time needed for change of code

 functionality - rule prevents code from malfunction

 operation - application support is easy to be monitored in the production environment

1.3. Vocabulary

Always, all - really always and all

Have not, don't use - no exceptions allowed

Protect/prevent - we are trying to avoid this, but there can be some exceptions. There must a very good reason for breaking the rule

We try - we want to make it this way if it is useful.

2. Terminology

2.1. General rules

2.1.1. Names of all objects are in English (1)

2.1.2. Names of all objects start with letters (2)

2.1.3. We always use exact and meaningful names (3)

We avoid using too generic names.

Good exaple	Bad example
instalsched.instalment	exact.pkg_exact
	amis

2.1.4. We always keep the same name for one thing (4)

Good example	Bad example
hardcontrol	pers,person

2.1.5. We are trying to avoid using the shortcuts if name is not too long (5)

- Shortcut is allowed to have 6 letters as a maximum
- Used shortcut must be unique and listed in the following table:

Full name	Shortcut
communication	comm
message server	msgsrv
statement	stment

Good example	Bad example
hardcontrol	pers
	err

2.1.6. Compound object names (127) valid from 17.8.2010

When object name consists from more than one word, then we have to separate the individual words by using the underscore.
In some cases using of number that replaces the whole word is allowed (2-to, 4-for).
We never use Camel style.

Good example	Bad example
prod.product_flag	checkandclose
encl.credit4encl	

2.2. PL/SQL naming conventions (6)

Object type	Name template (prefix/suffix)	Example
Global variable	g_	
Local variable	v_	
Cursor	cur_	
Record	_rec	
Array/table	_tab	
Object/type	t_	
Input parameter	p_	
Output parameter	p_	
Input/Output parameter	p_	
Exception	e_	
Constant	c_	

2.3. SQL naming conventions (7)

Object type	name template(prefix/suffix)	Convention	Example	Comment
Column		Use singular for the column content description. Don't use information enclosed in table name (usually entity name).	CODE ID_TEMPLATE STATUS	Details are listed in special chapter.
Column for DROP	X_<column name>		X_STATUS	The column that we want to drop stays for the whole release time with "X_" prefix, it will drop afterwards.

Trigger	TG_<table name>_XYZ	X is used for the following values: S - statement trigger R - row level trigger C - compound trigger O - instead of Y is used for the following values: A - after trigger B - before trigger Z is combination of letters I, U, D dependi ng on the fact which event has run the trigger - INSERT, UPDATE,DELETE.	TG_CREDIT_SAD TG_CREDIT_RBIU	We are trying to avoid their usage (mentioned in a separate chapter).
Queue	Q_			
Queue table	QT_			
Foreign key	FK_<table name from>_<table name to>		FK_INSTALMENT_CRE DIT	
Function	<verb>_<entity>	Verb that defines the functionality activity and the entity is the returned object name, even and entity for name of returned object or column.	calculate_debt get_id_emp	Used only in packages (details are listed in special chapter) Examples of verbs: • get – when the data from tables only load and are put further (all functions return something so we try not to overuse this word) • calculate – calculation of value • is - test functions of the entity attribute/status • can - test either access rights or procedure's input parameters
Index	ABI_<tablename>_<c olumn>	A presents the following values: G - global L - local nothing - others B is one of: N - nonunique U - unique F - function	GUI_TRANSACTION_I D_FEE LNI_TRANSACTION_I D_CREDIT NI_CREDIT_EVID_SRV UI_PERSON_IDENT FI_COMM_NOTSEND	When index is used to enforce constraint (primary or unique) then index have same name as constraint.
Package	PKG_<area> IFC_<schema/area> CONSTANTS EXCEPTIONS	Don't repeat the schema name in package name if not necessary.	pkg_export pkg_print pkg_evaluation	Details are listed in special chapter.
Type	T_<entity>			
Primary ley	PK_<tablename>		PK_CREDIT PK_PERSON	Index used for primary key should be named in the same way as PK.

Procedure				We use procedures in packages only (details listed in special chapter). Don't use standalone procedures.
Sequence	SQ_<tablename>		SQ_CREDIT SQ_PERSON	
Table	<tablename>	The entity description should be in singular.		
Temporary table	TMP_<tablename>		TMP_TRANSLATOR_GOODS_TYPE TMP_TYPE_INSTALMENT	
Table prepared to DROP	X_<tablename>		X_INSTALMENT	The table that we want to drop stays with "X_" prefix for the whole release time and it is dropped afterwards.
Historization table	H_<tablename>		H_CREDIT	
Unique key	UK_<tablename>_<column>			Index used to enforce unique key has the same name as unique key (UK).
Check constraint	CH_<tablename>_<column>			When constraint uses more then one column and it is not possible to compose a constraint name, then we use the simple numbering instead of column names and describe this in detail in the documentation.
View	VJ_<viewname>			Complex view
View 1:1 to table	VI_<tablename>		VI_CARD VI_CREDIT	
Materialized view	MV_			
DB Link		DB link name is equal to linked DB name.	CIF.HOMECREDIT.NET	

2.3.1. We never name the object in same words as schema (8)

ORACLE uses the DB schema name in a very similar way as the key name, the same names for schema and object might therefore cause some problem. ORACLE doesn't prevent from using the same name for object and schema but it is not recommended.
Example: When we use the same name for table as for schema where table is created, then ORACLE has a trouble with name recognition and it is not possible to use the full notation of schema package procedure in PL/SQL.

2.4. Scheduler objects' naming conventions (121) valid from 17.8.2010

Object type	name template (prefix/suffix)	Convention	Example	Comment

job_class	JOB_CLASS_aabb_xxx_zzz	aabb - abbreviation of system and destination (HO - Homer, VN -Vietnam) aa: HO - Homer HS - HomeSiS LP - LAP xxx - class type: APP - application OPR - operation zzz - logical name	JOB_CLASS_HOVN_APP_DEFAULT	New job classes can be created only after this has been agreed with operations department.
job	JOB_xxx	xxx - logical job name	JOB_EOM	
schedule	SCH_xxx	xxx - logical schedule name	SCH_HOLIDAYS	
window	WND_xxx	xxx - logical window name	WND_NIGHT_MAINT	We don't use windows for application jobs.
chain	CHN_xxx	xxx - logical chain name	CHN_EXACT	
rule	RULE_xxx_zzz	xxx - logical chain name the rule belongs to zzz - logical rule name	RULE_EXACT_START RULE_EXACT_END RULE_EOM_START_SLAVES	
step	ST_xxx	xxx - logical step name	ST_FIND_CREDITS	
program	PRG_xxx	xxx - logical program name	PRG_LOG_START	

3. SQL & PL/SQL

3.1. Genral rules

3.1.1. We have no declaration of unused variables, procedures, functions (9)

3.1.2. We have no unused code fragments (10)

- The part of code that is not/can´t be used, is recommended to delete
- We don't comment the old code fragments, just delete it (they are stored in SVN)

3.1.3. Don't hardcode literals into code (in PL/SQL) (11)

- Define literals in one place: appropriate package constants in a particular schema (see packages) – this rule is valid primarily for PL/SQL.
- Keep in mind that by using constant in SQL you define a bind variable for SQL statement and this can affect SQL generated execution plan

3.1.4. Always format PL/SQL code using PL/SQL Developer Beautifier (111)

Formatted code increases readability not only for its author. Code is formatted within PL/SQL Developer using the file beautifierX.br for given PL/SQL Developer version.
This file is stored in SVN:https://subversion.homecredit.net/repos/tools/db/plsql_developer/beautifier

When some parts of code can't be formatted automatically, then we have to do it manually so as the code readability is kept. We use the tags that will prevent the text from being automatically formatted according to beatuifierX.br file.

Example:

```
...
  -- NoFormat Start
   SELECT ...
     INTO ...
     FROM ...
    WHERE parameter  = ( SELECT ..
                         FROM ..
                        WHERE ..
                          ...
                      )
  -- NoFormat End
```

Comment: Beatuifier can handle model clause only from version 9.

3.2. Tables

3.2.1. We do not have ROWID and UROWID stored in tables (13)

DB can´t guarantee the consistency of link to ROWID-row in table.

3.2.2. We don´t have the object type columns in tables (14)

The tolerated exception is storing the objects without dates in action code list.

3.2.3. All tables that refer to strong entities contain columns for logging of row changes (15)

This refers to the following columns:

Sloupec	Datový typ	Popis
MODIF_TIME	supp.IFC_SUPP.t_date	Time of last modification
INS_TIME	supp.IFC_SUPP.t_date	Time when record was created
ID_EMP	supp.IFC_SUPP.t_primary_key	User ID who made last modification
ID_EMP_INS	supp.IFC_SUPP.t_primary_key	User ID who created record

There is an exception for pure relational tables.

3.2.4. We don't create foreign key to table with user accounts(140) valid from 10.11.2014

For performance reason we don't create foreign keys on columns with information who created or modified record to table with user accounts.

For HOMER columns are id_emp_ins and id_emp and table is adm.employee.

For HOSEL columns are sloupce created_by and updated_by and table is tabulka user_detail.

3.2.5. All tables contain ID column as a primary key generated from sequence (16)

There is one sequence for each table used for ID generation with the exception of:

- external tables
- pure relational tables that contain the foreign keys only and a primary key contains all table columns
- tables that are an extension of primary table and therefore the foreign key to primary table is a primary key of extension table at the same time

3.2.6. All tables are created in correct tablespace (17)

The following tablespaces are available: SMALL_DATA, MIDDLE_DATA, LARGE_DATA
Empty table is created in SMALL_DATA, when table is being filled during its creation, we will try to estimate its correct storage according to the amount of inserted data. The movement itself, in case of table increase, is an operational matter.

Comment: Please use "dbadmin.em_segments_tablespace" view (prefered_tablespace_name column) to find out the recommended tablespace

3.2.7. All indexes are created in the correct tablespace (18)

The following tablespaces are available: SMALL_IDX, MIDDLE_IDX, LARGE_IDX
The index for empty table is created in SMALL_IDX, when table is being filled during its creation, we will try to estimate its correct storage according to the amount of inserted data. The movement itself, in case of table increase, is an operational matter.

Comment: Please use "dbadmin.em_segments_tablespace" view (prefered_tablespace_name column) to find out the recommended tablespace

3.2.8. Each foreign key has an appropriate index created (19)

If this index is not created, then the change in parent table will cause the locking of whole child table.

3.2.9. Tables and columns have the comments (129) valid from 26.6.2012

All tables and columns are commented (COMMENT ON TABLE, COMMENT ON COLUMN). We recommend to write the comments in English, only in CS clone Czech can be used.

This rule is not valid for the structures generated by ORACLE (AQ tables, MV logs).

We always use English for product's systems (HOSEL, EOM,..).

3.2.10. Whenever character variable/column is defined, then length semantics must be used (136) valid from 29.7.2013

While creating a "string" column in DB table or PL/SQL variable, then we have to declare the length semantics explicitly. In most cases we will use CHAR semantics. We can't depend on DB settings. This parameter can be changed on session level any time.

Example:

```
ALTER TABLE ADD name VARCHAR2(30 CHAR);
```

Details in documentation

3.3. Variables and types

3.3.1. We don´t use variables and parameters defined by free type if they correspond with the column (20)

We use anchored definition whenever possible, for example for a table column or a predefined type.

Good example	Bad example
v_evid_srv contract.vi_credit.evid_srv%type;	v_evid_srv VARCHAR2(10);

3.3.2. We don't have variables initialized as NULL (21)

Variables are initialized as NULL automatically.

3.3.3. We try not to have variables initialized in declaration section using the functions (22)

The malfunctions occurred within the initialization or proceeding the function can't be handled by procedure exception handler.

3.3.4. We don't use double quotation marks (") in object name (23)

Double quotation marks allow us to use lower/upper case letters and special characters in object name. However, we don´t want to use any of this, so there is no need to use quotation marks.

Good example	Bad example
v_debt_and_peny	"v_dEbT+PenY"

3.3.5. We don't use exaggerately short names for variables (24)

Reducing the number of letters to minimum might cause unreadability of the code.

Good example	Bad example
v_debt_and_peny	v_1

There are exception for cycle index variables (FOR i in 1..v_count)

3.3.6. We don't use data NUMBER type without the accuracy definition (25)

1/ we save memory by defining the accuracy (default is 38 or system maximal precision)
2/ we increase readability

3.3.7. We preffer PLS_INTEGER instead of NUMBER for operation with natural numbers (26)

PLS_INTEGER range -2.147.483.648 - 2.147.483.647 on 32bitsystem. PLS_INTEGER consumes less memory than NUMBER. PLS_INTEGER is nearly 3 times faster for arithmetic operations. In ORACLE 11g we can use SIMPLE_INTEGER that is faster than PLS_INTEGER, but it can't contain NULL value.

3.3.8. We are avoiding CHAR data type usage (27)

Variables of CHAR data type are always right padded by spaces to fixed length which can be confusing

3.3.9. We always use explicit conversion functions when converting one data type to another (TO_DATE, TO_CHAR, ...) (28)

NLS server set up is not guaranteed.

3.3.10. We don't use VARCHAR data type(29)

We always use VARCHAR2 data type.

3.3.11. We don't assign empty string instead of NULL (30)

Whenever we want to use NULL, it is good to use it as NULL and not as '', though it has the same meaning. We don't know if this stays the same in next versions of ORACLE.

3.3.12. We use BOOLEAN data type for YES/NO variables (31)

This is valid for PL/SQL only.

Good example	Bad example
DECLARE v_bigger BOOLEAN; ... BEGIN v_bigger := NVL(v_newFile < v_oldFile, FALSE); ...	DECLARE v_bigger NUMBER(1); ... BEGIN IF v_newFile < v_oldFile THEN v_bigger := 1; ELSE v_bigger := 0; END IF; ...

3.3.13. We use NOCOPY hint for huge IN OUT or OUT parameters (32)

ORACLE handles variables by value, ie. it allocates space on stack for the parameters. When entering the called procedure/function, IN and IN OUT content will copy to stack. After finishing the procedure, the IN/IN OUT parameters process a copy of stack values in variables. This is a very expensive operation for huge variables (long strings, collections, LOBs) both from memory and time consuming point of view. NOCOPY hint is implicit for IN parameters from ORACLE 9i, so it is not necessary to write there is hint NOCOPY implicit for IN parameters so it not necessary to write them explicitly.
Be careful, this is only a hint so you can't be 100% sure that the variable will be handled by reference and not by value (see ORACLE documentation for defined exceptions).
Comment:If NOCOPY hint is listed, please keep in mind that in case it is not mentioned, IN OUT/OUT parameters are copied only after successful procedure fulfillment and are therefore consistent.If the exception that quits the procedure is raised, nothing is copied. This doesn´t apply to the case when the variables are handled by reference, the variables are then changed directly in a called procedure and if they finish by exception, IN OUT/OUT parameters values might be inconsistent and a calling code must handle this situation correctly.

3.4. DML and SQL

3.4.1. Always specify the columns in INSERTS (33)

SQL statement is then immune to table structure changes.

3.4.2. Always use table alias in queries (34)

If possible, use the same alias for the same table within the whole package/schema.

3.4.3. Always use ANSI-Join syntax (35)

We are then able to differ join condition from restriction condition.

3.4.4. We try to read cursors into the records, not to the list of variables (36)

Structure change can be solved much more efficiently when record is used.

Good example	Bad example
DECLARE CURSOR cur_user IS SELECT id, firstname, lastname FROM user; v_user_rec cur_user%ROWTYPE; BEGIN OPEN cur_user; LOOP FETCH cur_user INTO v_user_rec; EXIT WHEN cur_user%NOTFOUND -- do something END LOOP; CLOSE cur_user; END;	DECLARE CURSOR cur_user IS SELECT id, firstname, lastname FROM user; v_id_user USER.ID%TYPE; v_firstname USER.FIRSTNAME%TYPE; v_lastname USER.LASTNAME%TYPE; BEGIN OPEN cur_user; LOOP FETCH cur_user INTO v_id_user, v_fistname, v_lastname; EXIT WHEN cur_user%NOTFOUND END LOOP; CLOSE cur_user; END;

3.4.5. Bind variables in SQL in OLTP (131) valid from 31.10.2012

SQL used in OTLP system always use bind variables. In PL/SQL it is for free because of SQL integration. Be careful when using dynamic SQL where SQL is concatenated in runtime which might lead to direct use of variable value.
Easy hint:
whenever using execute immediate, then it must be with using the key word
whenever using dbms_sql, then I have to call dbms_sql.bind_variable

3.4.6. Don't use PL/SQL functions in SQL (133) valid from 31.10.2012

We try to avoid using PL/SQL function in SQL statements. It is because CBO is not able to rewrite PL/SQL function to pure SQL and include it to main SQL statement.
Functions that read lots of data can be potentially very problematic.
Deterministic functions are almost without any troubles, because there are working with input parameters only and don't contain any SQL.

3.5. Flow control structures

3.5.1. We always use %NOTFOUND instead of NOT %FOUND when control reading from cursor (37)

Sentences without any negation enhance the code readability.

3.5.2. We always close locally opened cursors (38)

Opened cursors consume space in SGA. Number of concurrently opened cursors is restricted per session which might come out in the exception.

3.5.3. Cursor opened in procedure must be closed in procedure exception handler (39)

3.5.4. We don't have any additional SQL nor PL/SQL code parts between SQL operations and follow-up implicit cursor test (40)

Embedded code can proceed the operation that has the effect on %ROWCOUNT or %FOUND values.

Good example	Bad example
`DELETE FROM ADM.EMPLOYEE e` `WHERE e.ID = v_id_emp;` `IF SQL%ROWCOUNT > 1` `THEN` `  ROLLBACK;` `END IF;`	`DELETE FROM ADM.EMPLOYEE e` `WHERE e.ID = v_id_emp;` `delete_user_history(v_id_emp);` `IF SQL%ROWCOUNT > 1` `THEN` `  ROLLBACK;` `END IF;`

3.5.5. We always use CASE instead of IF with many ELSIF (41)

CASE is better readable and new choices can be easily added.

Good example	Bad example
`CASE v_color` `WHEN c_color_red THEN ...` `WHEN c_color_blue THEN ...` `WHEN c_color_green THEN ...` `END;`	`IF v_color = c_color_red THEN ...` `ELSEIF v_color = c_color_blue THEN ...` `ELSEIF v_color = c_color_green THEN ...` `END IF;`

3.5.6. We preffer using CASE instead of DECODE (42)

DECODE is an old function which was replaced by more readable CASE. CASE is even available in PL/SQL.

3.5.7. We always use COALESCE instead of NVL when second NVL parameter is function call or SQL statement (43)

NVL function always evaluates both parameters which can be very inefficient.

3.5.8. We always use CASE instead of NVL2 when the 2nd or 3rd NVL2 parameter is function call or SQL statement (44)

NVL2 function always evaluates both parameters which can be very inefficient.

3.5.9. We don´t have unreferenced index in FOR cycles (45)

Whenever we don't use FOR index value in loop body then we are nearly 100% sure that our loop code is wrong or useless.

3.5.10. We don´t have fixed low and high values in FOR cycles (46)

Constants are changing all the time. So it is better to obtain them then to have them fixed in code. There is low value exception which is 1 in most cases, especially when we are looping trough some collection.

3.6. Exceptions

The exception is a situation in runtime whenever the error, that is unexpected and obstructs the smooth application running, occurs. Since the exceptions are not expected to occur, we must always find out what was the reason and how we can fix it. For example by input parameters validation, table row data validation and so on.

3.6.1. Raising exception outside procedure is allowed only using logs.ifc_logs.error (47)

Never use RAISE_APPLICATION_ERROR.

3.6.2. Public exceptions are defined in package SCHEMA.EXCEPTIONS (48)

This package is automaticly generated during release deployment and can't be stored in SVN. For package generation are dedicated procedures in logs.ifc_logs.

3.6.3. . EXCEPTION WHEN OTHERS exception must be reraised by using logs.ifc_logs.error as a new exception (49)

This exception must be reraised not masked.

3.6.4. Whenever we are using logs.ifc_logs.error we set parameter p_parameters at least with procedure input parameters (50)

If the exception is already logged, then its meaningful value without any input parameters is insufficient.

3.6.5. Don't use EXCEPTION WHEN OTHERS whenever exact excaption can be used (NO_DATA_FOUND, DUP_VAL_ON_INDEX, TOO_MANY_ROWS) (51)

3.6.6. EXCEPTION WHEN OTHERS must be listed as last exception in exception handler (52)

3.6.7. EXCEPTION WHEN OTHERS must be handled in every procedure/function (53)

This rule guarantees that all exceptions will be logged.

3.6.8. Never handle exception by its number (54)

Good example	Bad example
`` ... EXCEPTION WHEN no_data_found THEN ... WHEN OTHERS THEN ... END; ```	``` ... EXCEPTION WHEN OTHERS THEN IF SQLCODE = 100 THEN ... END IF; ... END; ```

3.7. Dynamic SQL

3.7.1. We are always using string variable for dynamic SQL (55)

We can simply debug or log the proceeded SQL statement.

3.8. Stored objects

3.8.1. We always use name notation whenever calling some procedure/function (56)

The following adding of new parameters is much more safer. Using parameter name increases the code readability.
Comment: From ORACLE 11g we can us this notation in SQL too.

Good example	Bad example
param.IFC_PARAM.get_regname(p_number => 75, p_value_code => 'a')	param.IFC_PARAM.get_regname(75,'a')

3.8.2. We are trying to add block name after END keyword (57)

Exception: packages

3.8.3. We always prefer using parameters instead of global variable (58)

Exception: large cached collections, internal status packages variables when whole package acts as a class. Procedure/function is then not a standalone entity and its usage is complicated.

3.8.4. We are trying not to have unrefferenced objects (59)

3.8.5. We have no unused parameters (60)

3.8.6. We are trying to reduce procedure length to one page as a maximum (61)

3.8.7. We are trying to reduce procedure length to one page as a maximum (62)

This is not a dogma, it is just a clue that package needs to be separated in more smaller packages.

3.8.8. Package variable are declared in package body not in package header (63)

We create get&set methods for internal variables so as we have full control over setting values to variables.

3.8.9. All procedures and functions are in packages (64)

3.8.10. We don't use RETURN in procedures (65)

Every procedure has just one trail line and that is the last one with except of exception handler section.

3.8.11. We don´t have more than one RETURN statements in functions (66)

3.8.12. All functions have RETURN as the last statement (67)

3.8.13. We don´t have functions with OUT parameters (68)

3.8.14. We don't return NULL from function with BOOLEAN return data type (69)

Be carefull when comparing NULL value:

```
DECLARE
   v_test_result BOOLEAN;
BEGIN
   v_test_result := 10 < NULL;
   IF v_test_result IS NULL THEN
      dbms_output.put_line('Result is NULL');
   END IF;
END;
```

3.8.15. . If the function is used to test the possibilities to proceed the action, then the correct control is indicated by returning 0 (70)

3.8.16. Try to hold number of procedures/functions parameters on a reasonable amount (135) valid from 1.6.2013

For good readability try to hold number of procedure parameters up to 10. When it is necessary to use more than 10 parameters, then we do so by using RECORD data type variables. Parameters are grouped within the record on logical level. Whenever some of parameters repeat, for example the address (once is permanenet residence, the other time a correspondence address), it worths considering whether to use collection as a parameter (the collection that would contain such parameters).

3.9. Logging

3.9.1. We log all important steps in the code (71)

There are four logging levels:

DEBUG	Detailed information used for debugging purposes (procedure parameters, variable values, IF branche). This logging level is not used by default.
INFO	Interesting runtime events (startup, start/end of process).
WARN	Information about unusual events (deprecated functionality is used, runtime situation which is not intended or unexpected but not fault)
ERROR	Unexpected runtime error that must be resolved. This level raise exception simultaneously.

3.9.2. In every procedure/function body fully qualified procedure name and input parameters are logged on DEBUG level (72)

Logging code is generated by this package https://subversion.homecredit.net/repos/tools/db/code_generators/debug/debug_log_code_generator.pck

3.9.3. For logging purposes use logs.ifc_logs only (73)

3.9.4. In old fasion functionality using errnum/errmsg/v_ok when fault is indicated these variables are logged on WARN level (74)

3.10. Versioning

3.10.1. All source codes and deploy scripts that create the application are versioned in Subversion (77)

Subversion usage rules are defined on this page Subversion pravidel.

3.10.2. Versioning runs in schema (directory) / object (file) structure (78)

3.10.3. When one clone is used for more then one country then country specific implementation is added between schema (directory) and object (file) as a directory with name equal to country code (79)

3.10.4. PL/SQL source code is encoded in ANSI, deployment scripts in UTF8-BOM (80)

3.10.5. View are always created with CREATE OR REPLACE FORCE VIEW (81)

Since we don´t solve the deployment order within the process itself, the view would not succed in deployment if all view referred objects did not exist in a valid state. There is always a list of columns in view definition. Using * is forbiden.

3.10.6. We don't have any empty rows in view definition (82)

Deployment is done using PL/SQL Developer's command window or SQL*Plus and empty row generate a problem (SQL is not processed correctly).

3.10.7. Every file with source code contains row with slash (/) as the last line (83)

Slash (/) is a mandatory part of PL/SQL code. When all source scripts end with slash, then there is no need to add them during the deployment script generation process and this prevents from doubled PL/SQL code deployment.

3.10.8. Object types (130) valid from 26.6.2012

In all cases API and objects bodies will be stored separately. The schema is as follows: [Schema] [Settings] Destination.pck Obj.bdy – object body [obj] Obj.spc – object API (specification)

By words – object points are stored on the same level as other files for given schema and it is therefore included in deployment script. Specification is used only for changes tracking in subdirectory that is not being deployed. Whenever it is necessary to deploy object specification (both for new and current one) in release, it is done by putting required SQL statement to release scripts. Please see the more detailed information here.

3.11. Materialized views

3.11.1. We use MV only if necessary (84)

MV is usually used for long running question optimization where big tables merging is prepared in advance. Query rewrite is used in this case. Or when we need replicate the data between two DBs.

3.11.2. MV are created with a regard on its refresh (using both Fast and Complete method) so as it always runs on request and not automatically (85)

We will avoid unnecessary load of the system, especially in cases when it is useless.

```
CREATE MATERIALIZED VIEW dm.mv_ft_cm_contract_ad
  TABLESPACE DW_DATA_64M
  NOCACHE LOGGING PARALLEL (DEGREE DEFAULT INSTANCES DEFAULT) BUILD DEFERRED
  USING INDEX TABLESPACE DW_IDX_32M
  REFRESH FORCE ON DEMAND WITH PRIMARY KEY AS
SELECT ...
```

3.11.3. MV always refreshes so as the synchronization with primary table runs through a primary key and not through ROWID (86)

The problem with table moving where a complete table refresh would be requested is therefore eliminated.


```
CREATE MATERIALIZED VIEW dm.mv_ft_cm_contract_ad
  TABLESPACE DW_DATA_64M
  NOCACHE LOGGING PARALLEL (DEGREE DEFAULT INSTANCES DEFAULT) BUILD DEFERRED
  USING INDEX TABLESPACE DW_IDX_32M
  REFRESH FORCE ON DEMAND WITH PRIMARY KEY AS
SELECT ...
```


3.12. Hints

3.12.1. We try to avoid using the hints (87)

3.12.2. Never use undocumented hints (88)

Hint functionality is not guaranteed in new ORACLE versions. The hint can´t be used in general. Each SQL statement with hint must be dealed individually for each given DB and it's version.

3.12.3. We never use hint parallel without specification of parallel degree (132) valid from 31.10.2012

Whenever the hint parallel is used, then the parallelism degree must be defined.

3.13. Triggers

3.13.1. Don't use triggers except of special cases (89)

Triggers existence can be found only in ORACLE data dictionary or by using development tools. You are not able to find trigger existence from reading PL/SQL code. When ORACLE is performing DML statement, it must check trigger presence and evaluates the conditions for its running which requires some system resources. Until ORACLE11g you can't control execution order of triggers with the same type.

3.13.2. We use triggers for historisation data changes (90)

Whole historicisation logic is concentrated in one place and it is run every time the data are changed regardless the client who provides the change including the data manipulation proceeded within the service.

3.13.3. Trigger is always writen following PL/SQL code development rules (91)

Trigger is de facto anonymous PL/SQL block, which means that the same rules need to be applied.

3.14. Check constraints

3.14.1. We using appropriate check constraints (92)

The data validation in DB is done by using the check constraint and we have therefore ensured that the data will be consistent even in case that any of application control fails or in case the data are processed directly. Check constraints validation consumes some system resources. It slowdowns INSERT and UPDATE operations. Be careful on number and complexity of check constraint you are using on a table.CBO (Cost Based Optimizer) is able to use some of check constraints and generate more effective execution plan.

3.15. Advanced Queue

3.15.1. We always use defined object type as AQ payload (93)

Using object type gives us full control over queue content. We can simply modify data content in message by adding/removing object type attribute.

3.15.2. We always write/read from/to a queue by using our own procedures (94)

Usage of dbms_aq for reading/writing to queue requires definition of some auxiliary variables. When we encapsulate dbms_aq we are increasing code readability. Procedure dequeue riase an exception when reading from empty queue and handle this exception in encapsulation procedure is an easy task. Queue emptyness can be indicated by additional out parameter. By encapsulating procedures dbms_aq.enqueue a dbms_aq.dequeue we concentrate the own reading/writting logic to one place and any possible changes are then made in one place. The places of encapsulated procedures are not affected.

3.16. Jobs

3.16.1. We always log information about the job run (95)

It is necessary from operation point of view that all jobs log the information about their run. Monitoring system is then able to generate the particular warning messages based on this information, for example to contact Emergency if the job marked as critical ends up with fault.

3.16.2. Job session must be always indicated (96)

The monitoring tools are always able to detect if any session consumes a lot of system resources, has locked a lot of records etc. If job XY which provides C action is indicated directly in session, it is then easy to solve the possible problems.

3.16.3. We always indicate job progress/processing (97)

Whenever job is processing some pile of records in a loop then it is necessary to indicate the processing procedure so as it is possible to measure the speed, estimate the time of job finish etc.

3.16.4. We always control the transaction within the job explicitly (98)

Job must process the transaction control (commit proceeding), it must not count on automatic implicit commit within the session end.

3.16.5. We perform the commit for processing x records within the job (99)

Since commit presents the record in redolog, it is advisable to process it within the job, if containing a cycle, commit after x cycle transit. Comment: The exact value of X depends always on the job and the amount changes proceeded within one cycle transit.

3.16.6. We try to make job re-runable (100)

Job can fail any time for any reason. If job doesn´t proceed within one transaction, then some records have been processed. After the deleting the the job malfunction cause, it is important to re-run it again so as it is able to proceed the remaining records. If it is not able to do that, then the processing of remaining records is more complicated.

3.16.7. Every job must have filled in the header for JOBMONITOR (101)

For JOBMONITOR application that is used for monitoring the jobs, it is necessary to fill in the header. Otherwise the JOBMONITOR is not able to fully monitor the job. Please see the more detailed description of JOBMONITOR on operation department WIKI.

3.16.8. We always have the pkg_job package procedure calling of the particular schema as the only job body (102)

When the own job body contains the procedure calling only, then the all logic is concentrated in one place and is easily handled.

3.16.9. Never raise exception out of job body (103)

ORACLE (dbms_job) jobs are repeated in case that they end up by fault automatically repeated up to 16 times. If the 16th repeat fails as well, then the broken status is set up in job. The interval for repeating depends on ORACLE and we have therefore no control over the time when the jobs will be run.

3.16.10. We always time the job with regard on DB application time (104)

Dbms_job timing depends on OS time and varies from application time. For example it is 2 hours difference in RU destination and 4-5 hours in KZ destination depending on summer/winter time period in the Czech republic. We solve this in Scheduler according to rule 123.

3.16.11. Job are created in Scheduler only (128) valid from 26.6.2012

Application jobs are created in Scheduler only because it gives us better possibility to time and control the jobs.

3.17. Scheduler

3.17.1. Application job in Scheduler must use given job class (122) valid from 17.8.2010

Until ORACLE 11gR2 Scheduler jobs can't be stopped without using services connected to job classes.
Application job's job class has name defined by naming convention.

3.17.2. Start_date of job must be defined with timezone in format region/city (123) valid from 17.8.2010

When time zone of start date is defined in format region/city then Scheduler is able correctly follow daylightsaving time changes.

3.17.3. We use timing with PL/SQL for jobs with a short run period (124) valid from 17.8.2010

For this type of job is relevant period between two job runs than exact time when job is run. Using PL/SQL formula eliminate repeating job run after outage for missed runs when schedule_limit attribute is not set.

3.17.4. It is forbidden to use window for application job (125) valid from 17.8.2010

Window is dedicated to be used for maintenance jobs where job is automatically ended when window ends.

3.17.5. We try to have schedule_limit set up in all jobs (126) valid from 17.8.2010

When parameter_schedule_limit is not set up, then the jobs is automatically run in case of missing the regular run due to outage. This run is eliminated by correct set up which will make the start up after outage more smooth. Schedule_limit value is needed to set so as it specifies a maximum possible run delay where the delayed run is still running with regard on the time of this process.

Example: If we know that the job is run every day at 22 :00 and the job run time is 6 hours and the requirement is the job should run by 6.00 a.m., then the schedule_limit value is 2 hours. I.e. if the run at 22.00 is not proceeded due to outage and this will end at 24.00, then the job starts running, otherwise it is rescheduled to the following run (the run is skipped).

3.18. RAC – Real Application Cluster

3.18.1. We always define cache parameter for frequently used sequences (105)

If the value in RAC environment is set up to nocache or low value, then the frequent synchronization over RAC is performed. That means that more nodes cause longer time to synchronization and getting nextval from sequence takes longer time. In is therefore recommended to use higher values of cache parameters for very active sequences (1000 and more).

3.18.2. We don't use queries for dynamic views (106)

Dynamic v$xxx views contain data for specific instances (node). When application logic is build up on those data then it is necessary to use the gv$xxx views in RAC environment. Those views are union all v$xxx from all nodes in RAC. Selection from gv$xxxx views are not very fast (it is distributed query).
Comment: Dynamic views can be used when necessary and where there is no other simple way.

3.18.3. We don't use pipes in RAC (107)

Pipe is available only within the instance so it can't be used for the synchronization across the RAC.

3.19. Comments

3.19.1. We always comment the source code (108)

The goal is to describe the functionality of particular code fragments.
We write comments in a following way:

```
-- simple description of application logic
 --  , description is continuing on next row
 IF p_pocet > 0
 THEN
 ...
```

, another way as:

```
/**
   */
```

is forbidden to use for the reason of unified recording.

3.19.2. Every procedure, function or package have the document header created by PLSQLDOC tags (109)

```
CREATE OR REPLACE PROCEDURE  nazev_procedury AS
 -- Simlpe description of procedure
 -- %version  $Id$
 -- %author        author's name
 -- %param         parametr's name and it's description
 -- %raise         exception
    -- %deprecated    code marked as deprecated is not used in new code
 ...
 BEGIN
 ...
```

3.19.3. Comments are written in Czech without diacritics and always above the description part of source code (110)

3.19.4. We don´t have nestedcomments (12)

3.20. Packages

3.20.1. Procedures/functions in ifc_ package can only be called to foreign schema (112)

Ifc_ packages are designed as a schema public API. There can be more then one ifc_xxx packages per schema.

3.20.2. There is no logic in ifc_xxx packages (113)

3.20.3. All the constants, literals and related functions are declared in CONSTANTS packages (114)

There is only one such package per schema.

3.20.4. User defined exceptions are defined only in EXCEPTIONS packages (115)

Package exceptions are not versioned in SVN, except of schema exceptions.

3.20.5. Calling another package within the same schema is direct, without using trought ifc_xxx package (116) (116)

3.20.6. Procedures and functions inside the package are called with short notation (117)

3.20.7. Procedures from another package in the same schema are always called with full notation (schema.package.procedure) (118)

3.20.8. The transaction is not controlled in PL/SQL code called by client (119)

The transaction is controlled by the application (client) itself. This does not refer to autonomous transactions and jobs.

3.21. Grants

3.21.1. Grants between the schemas are granted for ifc_xxx packages and vi_xxx views only (120)

There is an exception for role HCI_ACCESS_ORM. This role has the rights directly to table.

Application with large portion of PL/SQL (old HOMER)
Applications have their own schemas (appuser) that refer to the above mentioned.

Application with minimal portion of PL/SQl code (BSL)
Application have one application schema (schema A). This schema owns tables and second schema (schema B) taht is used to access DB from application server. Schema A then grants DML rights on owned tables to schema B.

3.21.2. Grants and views (139) valid from 17.9.2014

When view in schema A is referencing object oustside schema A then it is necessary to grant required priviled with clause WITH GRANT OPTION.

Note: THis is necessary due to a changes made in ORACLE 11.2.0.4 more details you found there Upgrade 11.2.0.4

3.22. DB Links

3.22.1. We don't use public DB link (134) valid from 1.6.2013

For security reason it is forbidden to use public DB links. We use private DB links only.

3.23. Access to DB

3.23.1. Access to DB from 3rd languages (137) valid from 27.1.2014

Application with large portion of PL/SQL (old HOMER)
We need have control what DB objects are used outside DB. For that reason we create separate DB schemas known as interface. We have following schemas: WS - for webservice WEB - for web client APP_VB - for VB6 client Access to others schemas from JAVA/VB6 is forbidden. DB procedures and functions are published through ifc_ packages. There is a separate package for aech source DB schema. Example: When webservice need to use some function from card.ifc_card then this function is published to ws.ifc_card. Tables are accessed using 1:1 views. As for PL/SQL code same rule is applied. When webservice need to read data from table card.payment_card then view card.vi_payment_card 1:1 to card.payment_card is created and then view ws.vi_payment_card 1:1 to card.vi_payment_card is created too. Notice: Current state when JAVA access some DB objects directly and not throught interface schema will be removed.

Application with minimal portion of PL/SQL code (BSL)
Application have one application DB schema (schema A). this schema own DB objects and second DB schema (schema B) is used as interface. TO this schema application sever is logged on. Tables with archivation atrribute (column) have view that is builed with condition that filter archved reows out (archived=0). In this case Dbs chema B containt sysnonym pointing to this view. Db schema B contain synonym to rest of schema A: sequence tables without archivation atrribute (column archived) packages, standalone procedures and functions

3.24. Synonyms

3.24.1. Synonym usage (138) valid from 13.6.2014

Synonym is simple but powerfull feature and it's wise usage is not forbiden.

Typical usage:

- backward compatibility
- preparation for table move from schema A to schema B

Usage of PUBLIC synonym is not recommended.