R12.2 Extension and Development Tips and Tricks

John Peters
JRPJR, Inc.
john.peters@jrpjr.com

Introduction

This paper has come out of my years of experience as an Oracle Consultant and Developer. I have come up with some creative solutions to EBS development challenges. As a consultant I have observed many ways companies have handled EBS development, some good, some bad. This paper is just a first attempt to skim off some of the key points I would like to make. This collection of about 30 topics is just the top of the pile of things I would like to share. I tried to keep some of this high level, otherwise this would be hundreds of pages long, while providing references to drill down on topics in more detail if you desire.

I have organized these tips in the paper into the following high level groupings:
Development Standards (2 tips)
OnLine Patching, EBR (6 tips)
Concurrent Programs (4 tips)
Interfaces (3 tips)
PL/SQL Code (7 tips)
Don't Hard Code Configure Instead (5 tips)
Use New Technology (3 tips)

I hope you find this useful. I would love to hear feedback so feel free to drop me an email at john.peters@jrpjr.com.

Thanks and Enjoy!

Development Standards

I cannot emphasize the importance of development standards. Standards provide a common framework for extensions and customizations. This commonality makes support and maintenance far easier. When a new set of eyes comes in it is far easier to grasp what is happening if all of the extensions and customizations look similar.

1) Oracle EBS Development Standards

With R12.2 and Edition Base Redefintion (EBR) and OnLine Patching, it has become an absolute requirement to follow Oracle's EBS Development standards documented in the following manuals:

Oracle® E-Business Suite Developer's Guide Release 12.2
(Part No. E22961-13), September 2015
Oracle® Workflow Developer's Guide Release 12.2
(Part No. E22011-11), September 2015
Oracle® E-Business Suite Integrated SOA Gateway Developer's Guide Release 12.2
(Part No. E20927-10), September 2015
Oracle® E-Business Suite Desktop Integration Framework Developer's Guide Release 12.2
(Part No. E22005-10), September 2015

I will not go into Oracle's Development standards in detail, the following two references provide a summary level view of this topic.

Oracle also has a great white paper that helps to summarize the development standards that are now required with R12.2.

1) Deploying Customizations in Oracle E-Business Suite Release 12.2 (MOS Doc ID 1577661.1)

I have also summarized some of these concepts in a prior presentation from a different perspective please take a look at this presentation for additional details.

2) R12.2 Development and Deployment of Customizations
http://jrpjr.com/paper_archive/collab15_R12_2_dev.pdf

2) Your EBS Development Standards

In addition, to Oracle's Development Standards your organization needs another layer of development standards. Topics that are key to cover are:

Documention of the Extension or Customization
What was the original gap?
Businesses change and without this original insight it is difficult to reevaluate extensions/customizations to determine if they are really still needed.

Who requested the gap to be filled?
Having a name or department goes a long way toward helping reevaluate the extensions/customizations.

High level design
What are the major moving parts to the extension/customization. Tables, code, programs, value sets, dff's, etc. It is often tough to find all those components after the fact.

On-going maintenance
When you are developing the extension/customization you have an idea where people are possibly going to trip up and forget to maintain things. Just jot them down in a spec so you have them for reference later on. As you test the code you will see how it handles errors and how to correct them. Again quickly make note of this so it can be referenced later no. These just have to be one liners and you can write them up in more detail later on if you get a chance.

How to migrate it
I now get in the habit of taking screen shots of the setups as I do them the first time. Sure there are some things I end up throwing away and deleting out of the spec. But you will be surprised that the majority of the initial decisions will stick. And if they are documented, someone else can migrate them from DEV to TEST to verify the documentation.

Object Naming Conventions
Of course we need naming conventions or DB objects (tables, views, pl/sql, triggers, sequences, etc). I prefer a convention of <custom product_code>_<area>_, so something like XXCUST_CONCUR_ for an interface to a 3rd party expense reporting system., of XXCUST_WOM_ for an interface from an external web store to OM.

But we also need these conventions for File System Objects. Install scripts, executables, reports, forms, etc should follow the same naming conventions as the DB objects. Also migration files used to install AOL objects into the database should follow the same convention and be stored in the $XXCUST_TOP/install/ directory. The same holds true for data fixes keep the same naming prefix for all

related files. That way when they are out of context, in a JIRA ticket or other location you know what they are related to.

Lastly make sure all of your scripts follow a similar structure, prompts should be similar so when a DBA or other user reviews them or runs them they know exactly what you are asking for.

Source Code Control
I have had mixed feelings on this over the years. I originally felt that the best source for the code was PROD. But with the advent of SOX controls PROD is often unavailable to developers. I can't see the source code in PROD or the filesystem and DBA's are often too busy to get you copies of whats in PROD quickly. In addition, with R12.2 we now have two sets of code, the Run and Patch editions, so getting something reliably from PROD becomes even more of challenging.

Check out some of the OpenSource options for code control:
- GIT/GITHub (A Merge based solution)
- Subversion (A Merge or Lock based solution)

I have even seen Microsoft Visual Source Safe (or more recently Team Foundation Server Basic) work just fine. Even SharePoint works fine, with some simple check in/check out capability as well as historical version tracking.

OnLine Patching, EBR

3) Register You Custom Application

It is now absolutely essential that you register your custom application in EBS due to Online Patching and EBR. It won't work if you don't. Of course this will include your custom top and environment variables to get to it.

Come up with a short name prefixed by 'XX' and use this throughout your custom work. This makes searching for custom objects far easier. I have clients where development has not don't this and we end up searching for things with 'XXCUST%', 'CUST%', and every other prefix known to mankind.

Everything should following this naming convention DB Objects, Filesystem Objects, Concurrent Programs, etc. A good option is a stock trading symbol (1-4 characters) or a company abbreviation.

When registering the custom application keep Short Name and Product Code the same. Why EBS Development ended up with AP and SQLAP is beyond me. Keep it simple on yourself.

I have seen companies use the one Custom Application prefix convention, and others that went with a set of Module Centric Custom Applications. I prefer the single Custom Application approach, especially when it is hard to draw a line between a customization that spans multiple modules, like OM and AR. I will sometimes give in and create additional Custom Applications if you have some really large customization or extension that is key to your business. But you want to watch out for that because it just provides another alternate naming convention you have search for and be familiar with.

The following query should be able to find you custom applications.

```
select t.APPLICATION_NAME,
       t.DESCRIPTION,
       a.APPLICATION_SHORT_NAME,
       a.PRODUCT_CODE,
       a.BASEPATH
  from FND_APPLICATION_TL t,
       FND_APPLICATION a
```

```
where a.APPLICATION_ID = t.APPLICATION_ID
  and a.APPLICATION_SHORT_NAME like 'XX%';
```

4) Know Edition Based Redefinition

EBS Online patching is built on top of EBR. You should fully understand how EBR works if you are attempting any R12.2 development. It is absolutely critical when trying understand the interplay between the Run and Patch editions.
The following are some simple concepts to keep in mind (there are some more detailed ones that result from these, but these are the basis for EBR).

Editionable DB Objects are:
- Synonyms
- Views
- All PL/SQL Object Types (Function, Package Spec/Body, Procedure, Trigger, Type and Type Body)

Everything else in the DB is non-Editionable.

Then based on those there are some Simple Rules
- A Non-Editioned Object cannot depend on an Editioned Object
- An Editioned Object (View) cannot be involved in a Foreign Key Constraint
- An Abstract Data Type cannot be both editioned and evolved

Edition
Basically this is an attribute assigned to the DB session and DB objects. In EBS you have two: Run and Patch. Remember that this only affects:: Views, Synonyms, PL/SQL

Editioning View
This provides a logical representation of the physical data model. Physical table storage may be different. What you think a column is named is just what the editioning view is presenting to you. There may be obsolete columns with similar names in the physical table storage.

Cross Edition Triggers
This is how we keep data in sync during an online patch cycle. Forward Cross Edition Trigger (FCET) is used to update the Patch data. Reverse Cross Edition Trigger (RCET) to update the Run data. RCET's are very rarely required so you can probably forget about them.

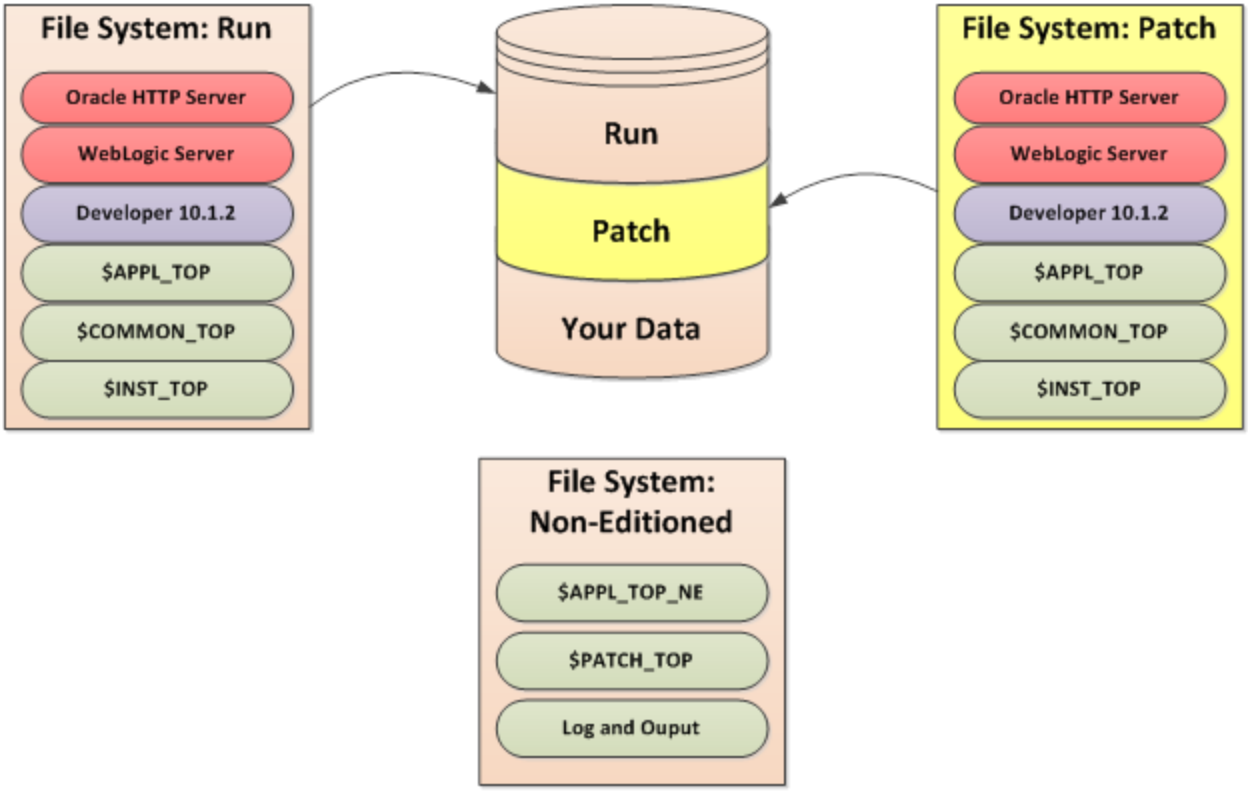
If you can completely grasp these concepts you are a long way towards understanding EBR and what you need to do when it comes to development.

5) Understand EBS Editions

EBR is an Oracle DB concept. EBS implements editions in a more simplistic approach of just Run and Patch. Keep in mind that each edition has a full filesystem, see the diagram below.

File system structure:
Has an Editioned set of directories: Run and Patch
Has a Non-Editioned set of directories, for things like request output

Database Objects
Has an Editioned set of object: Run and Patch
Your Data is not editioned (with the exception of seed data which is edition striped)



How to identify your Edition?
In the OS:

```
echo $FILE_EDITION
```

In a DB session:

```
select ad_zd.GET_EDITION_TYPE,
       ad_zd.GET_EDITION
  from dual;
```

How to set your Edition:
In the OS:

```
source /oracle/ebs122/EBSapps.env run
or
source /oracle/ebs122/EBSapps.env patch
echo $RUN_BASE          - the run filesystem
echo $PATCH_BASE  - the patch filesystem
```

In a DB session:

```
ad_zd.SET_EDITION('RUN')
or
ad_zd.SET_EDITION('PATCH')
```

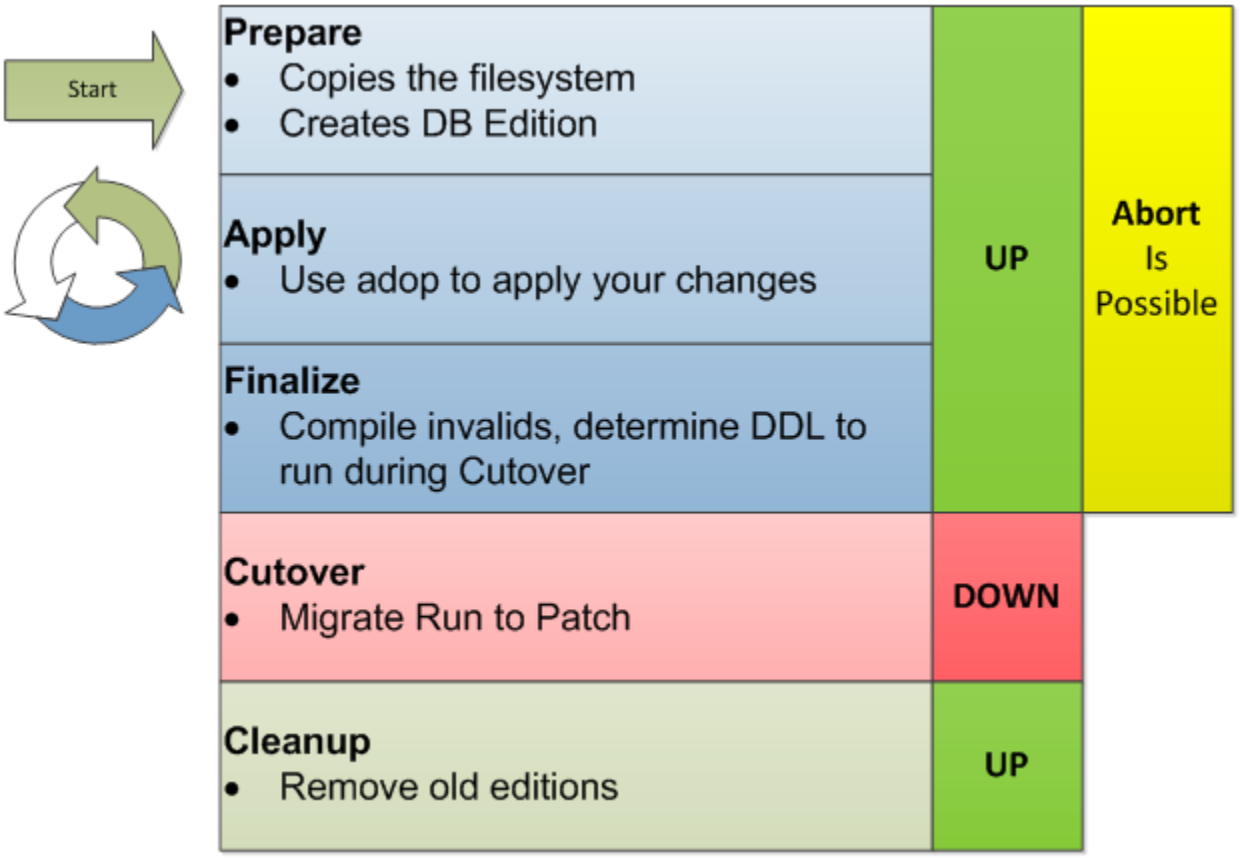
6) Develop in Run Edition

Remember that we always develop in the Run Edition of the DEV instance. One easy way to remember this is that only the Run Edition has a UI. So if you are not in the Run Edition there would be no way to perform the initial development related configuration and setup of the EBS environment.

Be very aware of open patch cycles in your DEV instance. DBA's and Developers have to be in constant communication or development will be overwritten by a patch cycle in DEV. This is absolutely critical, I have learned this the hard way myself. Look at number 7 below to see how you can identify if a patch cycle is open.

7) Understand EBS OnLine Patching

In particular understand the EBS Patch Cycle. What happens during each step in the Patch Cycle. When can a patch be aborted when can it no longer be aborted. During the cutover phase you can migrate things without the users being online. You don't have to an expert DBA on online patching, just be familiar with it.



How to check the status of patching, where in the cycle are you?
In the OS:

```
adop –status
```

In a DB session:

```
sqlplus apps @ADZDSHOWED.sql
```

This is especially important to verify if a patch cycle is in process in the development instance so you don't lose you work. Remember that Patch becomes Run and Patch is refreshed with a copy of the patched Run. So if you put your development in Run like you are supposed to, but it during an online patch cycle, it will be deleted and replaced with a copy of the patched Run edition.

8) Use EBS Tools to Deploy Changes

This one is still hard for even me to follow. I am so use to documenting my setups/configuration and just having someone rekey them. When deploying your changes in an Online Patching environment you won't have a UI. Remember that the UI is always hooked to the Run edition. There is no way to access the Patch edition from a screen or form. The only choice is to use the EBS tools to migrate changes since these connect at the DB level and don't require a UI. A partial list of the key tools to become familiar with are:

FNDLOAD
MOS Note: Tips and Examples Using FNDLOAD
(Doc ID 735338.1)

XMLImporter/XMLExporter
MOS Note: How to use XMLImporter/XMLExporter to import/export personalization (Doc ID 344204.1)

WFLOAD
MOS Note: How To Use WFLOAD To Download, Upload, Upgrade, Force Upload A Workflow To Database? (Doc ID 1569004.1)

XDOLoader
MOS Note: How To Use XDOLoader to Manage, Download and Upload Files? (Doc ID 469585.1)

I won't explain each of them here to keep this paper manageable in size. But take a look at the notes referenced above for more details on each.

If you don't use these tools you won't have a UI available for your migrations, so you will have to perform the steps after the patch cutover, when users can possibly be in the system and concurrent requests can start to run again. This is a bad practice to follow.

Concurrent Programs

9) Names

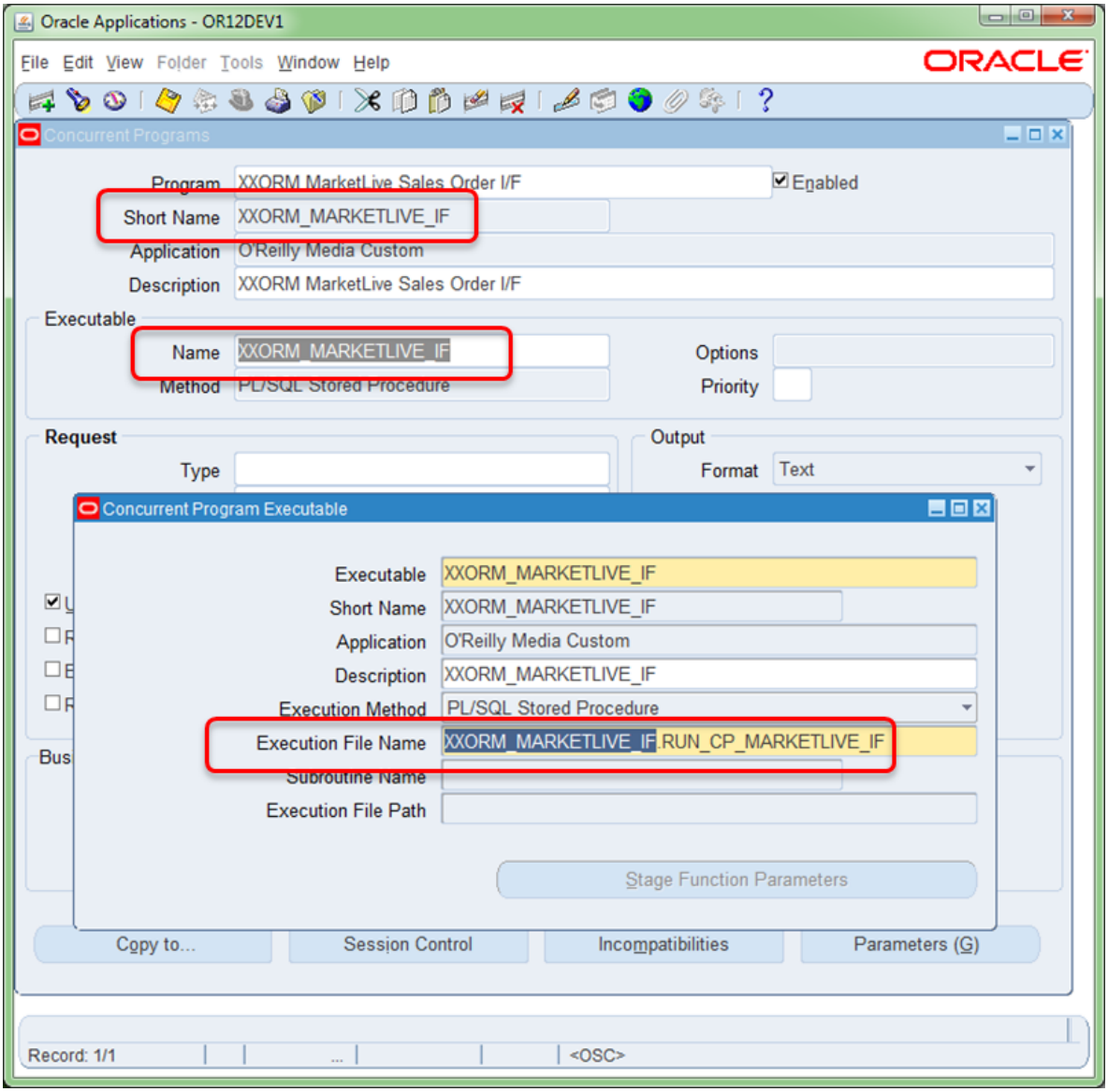
Follow a convention for the User Concurrent Program Name. I prefer to use the custom application APPLICATION_SHORT_NAME. If not use a standard prefix so the custom programs are easy to find. You should also register the custom programs to the custom application.

This way users clearly know what is custom and what is not. Nothing like having an end user put up an SR to Oracle Support on a custom program.

Users know generally what module the custom program is related to, usually a two letter abbreviation after the short name helps to group things to make it a little easier to find a program in an LOV. Example:

'XXORM INV%' – Inventory custom
'XXORM GL%' – General Ledger custom

The Program Short Name should begin with the same custom prefix as all of your other objects. This helps keep things together. Keep the Program Short Name, Executable, and Execution Filename in sync. There are a few rare exceptions where this is not possible, but the majority of time you should be able to keep them the same.



10) Output File

The output file is for the end user. Make sure it is as clear as possible and not cluttered up with extraneous debug related information.

Have a consistent header that includes the:
- Parameters user submitted with
- Request ID (fnd_global.CONC_REQUEST_ID)
- Database Name (v$database.NAME)

Have a consistent footer that includes the:
- Start Time, End Time, Calculated Run Time (min)
- Summarized Counts at end

The reason for this is, user have a way of saving Output files and then pulling them out months later and raising questions. If you don't have these details you won't know where the user ran it, or how.

11) Log File

The log file is for IT to help with debugging. Have a debug parameter to keep log files small unless in the middle of debugging. Writing gigabytes of log file messages will slow down a programs execution noticeably. Also when you end up with a huge log file you can't download it reliably from the Apps Tier. You also become an OS System Administrator's worst nightmare because you tend to fill up filesystem.

An option is to consider implementing a log table in the DB to hold messages. It is faster to search for rows in a log table that a million row text file.

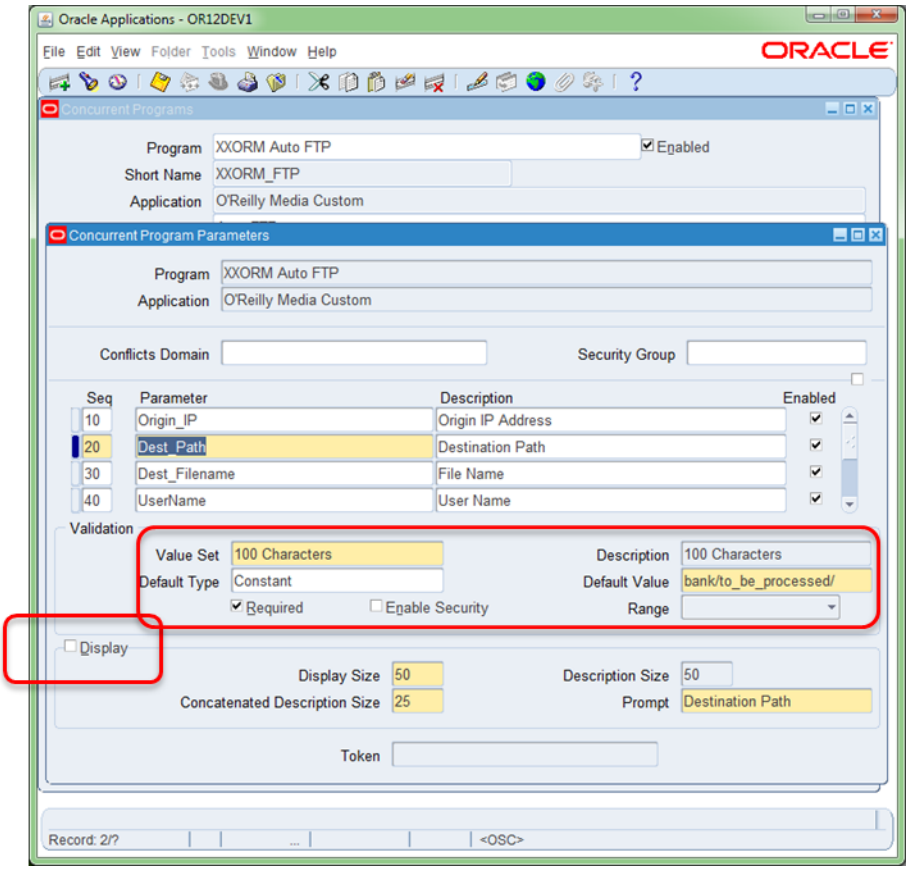
Always include a consistent header with the:
- Parameters User Submitted With
- Request ID (fnd_global.CONC_REQUEST_ID)
- Database Name (v$database.NAME)

And a consistent footer with the:
- Start Time, End Time, Calculated Run Time (min)

This is often all you will have access to since Output Files are no longer visible to all users of a responsibility in our current SOX restricted world.

12) Parameters

Don't hard code values in the program code, include them in hidden Concurrent Program Parameters with default constant values. Notice in the example below how I unchecked the Display option and set a constant default value. Better yet store them in a flexible data structure (more on that later).



Interfaces

13) Have DEV, TEST, PROD Support

Always consider support for a DEV, TEST, PROD instance of EBS and corresponding external system. This might seem really obvious to have DEV, TEST support for external system interfaces but it is surprising how many times it does not happen or is overlooked during the rush to implement an external system interface.

Once you have a PROD and non-PROD external system, you should automatically know the instance in your code and not ask the user. Again this seems obvious but it is really easy to implement. Your PROD instances usually have a DB Name that does not change. All of your non-PROD instances have names that are always changing and being added to and removed. So just check for the existence of the PROD name else assume non-PROD.
In the OS:

```
$TWO_TASK, or query from DB
```

In a DB seesion:

```
v$database.NAME
```

Then based on the instance you are running in have a user editable, flexible data structure that holds the required details (more on that later). This can hold remote system names, directories, passwords or key filenames, etc. Basically anything that is variable data should be stored in a flexible data structure and not hard coded.

14) Interface Top Directory

If this is a file based interface, use a standard Interface Top Directory structure for all of your interfaces with subdirectories like:

To_Be_Processed
Done
Error

Put this under the instance path, possibly under $APPLCSF.
$APPLCSF/interfaces/<if_name>/<processing>
Where:
Interfaces = constant string
<if_name> = the interface name or abbreviation
<processing> = the subdirectories mentioned above

This standard structure makes supporting these interfaces far easier in the future.

15) Consider Parsing File in PL/SQL

Consider reading and parsing your file in PL/SQL rather than using SQL*Loader. This is much simpler and you have more robust error/exception handling in PL/SQL than in SQL*Loader. And this skips the convoluted SQL*Loader control file syntax. This also keeps you out of having to write shell scripts to call and manage SQL*Loader.

In order to read the file from within the DB you will need to mount the Apps Tier interface directory structure on the DB Tier.

You will use the PL/SQL utl_file package to read the file. Don't use the deprecated utl_file_dir mechanism, instead use the new DB Directories, to reference the physical locations. These can be quickly updated during clones by the DBA using a script to handle differences between PROD and non-PROD instances.

I would not use the Oracle DB feature External Tables that allows you to read flat files as if they are database tables. The error handling mechanisms are similar to SQL*loader and I find them to be more difficult than parsing the file in PL/SQL. But check it out for yourself, you might prefer them.

PL/SQL Code

16) PL/SQL Package Version

Have you ever had that suspicion that one of the dozens of packages you just migrated are not the correct version of code. Well here is a simple method to allow you to audit exactly what version of code is installed in each instance.

In all of your PL/SQL packages have a Package Version function.

```
FUNCTION PACKAGE_VERSION
  RETURN VARCHAR2
IS
BEGIN
  RETURN '2016/01/13';
END PACKAGE_VERSION;
```

This should be the very first function in the Package Body, right after any declarations, right after the header revision comments. The string in this PACKAGE_VERSION function is manually maintained.

Now you can query code versions quickly in an instance, save the results to Excel and compare across instances the state of all of your code versions. The benefit to the Site Name profile option is that if your DBA's include a clone date in it you get this detail in your extract. Also in this example I have two separate DB's, EBS and Archive. In this environment they are always paired up, PROD EBS => PROD Archive, TEST EBS => TEST Archive. I am able to check the code versions in both since they should be in sync as well using a DB link.

```
select d.NAME sid,
       v.PROFILE_OPTION_VALUE site,
       xxcust_sla_interface.PACKAGE_VERSION,
       xxcust_sub_interface.PACKAGE_VERSION,
       xxcust_archiving_source.PACKAGE_VERSION,
       xxcust_archiving_dest.PACKAGE_VERSION@XXCUST_DEST_ARCH
  from FND_PROFILE_OPTION_VALUES v,
       FND_PROFILE_OPTIONS o,
       v$database d
 where o.PROFILE_OPTION_ID = v.PROFILE_OPTION_ID
   and o.PROFILE_OPTION_NAME = 'SITENAME';
```

17) Structure Your Code

Use case and indentation to make the code quickly readable. Either do this manually as you write the code or using a tool like TOAD. I have seen developers that seem to pride themselves on writing code that is all left indented. Why make it that hard on yourself trying to see how things are structured.

These are the exact same code fragments. What does the top one do? You can easily interpret the second on in a matter of seconds.

```
if (x_return_status != fnd_api.g_ret_sts_success) then for i in 1 .. x_msg_count
loop fnd_msg_pub.get(p_msg_index => i,p_encoded => fnd_api.g_false, p_data => v_message, p_msg_index_out =>
v_msg_index_out);
process_error_message (p_order_source_id => p_order_source_id, p_orig_sys_document_ref => p_orig_sys_document_ref,
p_orig_sys_line_ref => p_orig_sys_line_ref, p_error_message => 'error: ' || v_msg_index_out); end loop; end if;
```

```
if (X_RETURN_STATUS != FND_API.G_RET_STS_SUCCESS)
then
  FOR i IN 1 .. x_msg_count
  LOOP
    fnd_msg_pub.GET(p_msg_index => i,
                    p_encoded => Fnd_Api.G_FALSE,
                    p_data => v_message,
                    p_msg_index_out => v_msg_index_out);
    PROCESS_ERROR_MESSAGE (p_order_source_id => p_order_source_id,
                           p_orig_sys_document_ref => p_orig_sys_document_ref,
                           p_orig_sys_line_ref => p_orig_sys_line_ref,
                           p_error_message => 'ERROR: ' || v_msg_index_out);
  END LOOP;
end if;
```

18) Reference Parameters in Calls

Oracle PL/SQL does not force you reference parameter names in the calls to other routines. I think it just makes the code easier to read and it allows you to ensure you mapped complex calls properly. If you get 20+ parameters in a call it is tough to verify if 16 and 17 are swapped by accident.

This

```
fnd_msg_pub.GET(i,
                Fnd_Api.G_FALSE,
                v_message,
                v_msg_index_out);
```

Or This

```
fnd_msg_pub.GET(p_msg_index     => i,
                p_encoded       => Fnd_Api.G_FALSE,
                p_data          => v_message,
                p_msg_index_out => v_msg_index_out);
```

19) Create an Output Framework

Create a constant Output Header and Footer format that can be used on multiple programs. Just copy and paste from your old code to the new code.

```
fnd_file.PUT_LINE (fnd_file.OUTPUT,'Running XXORM MarketLive I/F at '
                   || to_char(sysdate,'DD-MON-YYYY HH24:MI:SS')
                   || ' Request ID ' || FND_GLOBAL.CONC_REQUEST_ID);
fnd_file.PUT_LINE (fnd_file.OUTPUT,'Parameters:');
fnd_file.PUT_LINE (fnd_file.OUTPUT,'  P_ORDER_SOURCE_ID = '
                   || p_order_source_id);
fnd_file.PUT_LINE (fnd_file.OUTPUT,'  P_ORIG_SYS_DOCUMENT_REF = '
                   || p_orig_sys_document_ref);
fnd_file.PUT_LINE (fnd_file.OUTPUT,'  P_INTERFACE_ORDERS = '
                   || p_interface_orders);
```

In the example above I am writing to FND_FILE. But you can easily write a wrapper function that uses FND_GLOBAL.CONC_REQUEST_ID to either write using DBMS_OUTPUT or FND_FILE. This allows you to run/test your code directly in PL/SQL and have the same code run as a concurrent request with no modifications.

20) Create a Logging Framework

Constant Log Header and Footer format that can be used on multiple programs. Notice that unlike the Output file header above I am using the actual PL/SQL procedure call for the log file since this file is used by IT for debugging. This allows you to quickly jump to that code to review it along with the log file.

```
fnd_file.PUT_LINE (fnd_file.LOG,'Running XXORM_MARKETLIVE_IF.RUN_CP_MARKETLIVE_IF at '
                   || to_char(sysdate,'DD-MON-YYYY HH24:MI:SS')
                   || ' Request ID ' || fnd_file.CONC_REQUEST_ID);
fnd_file.PUT_LINE (fnd_file.LOG,'Parameters:');
fnd_file.PUT_LINE (fnd_file.LOG,'  P_ORDER_SOURCE_ID = '
                   || p_order_source_id);
fnd_file.PUT_LINE (fnd_file.LOG,'  P_ORIG_SYS_DOCUMENT_REF = '
                   || p_orig_sys_document_ref);
fnd_file.PUT_LINE (fnd_file.LOG,'  P_INTERFACE_ORDERS = '
                   || p_interface_orders);
```

As done above in the header, wrap the write routines so it will correctly go to DBMS_OUTPUT or FND_FILE.

Use a global variable that holds Debug Level, this can be set from a concurrent program parameter, and therefore a parameter to your PL/SQL. Excessive Log file writing is often a root cause for many performance issues. You should be able to turn it on and off at will. It is far easier to include this as you write you code than trying to come back later and retrofit it.

Also write log messages to a custom table, use PRAGMA AUTONOMOUS_TRANSACTION so you can commit log table entries independently of the data being processed.

21) Always Set Request Status

Ensure you program completes with a status value:

- Normal: no issues encountered, viewing output file is optional
- Warning: issues the user can research and possibly fix, need to view output file
- Error: fatal issues, SQL or PL/SQL, unhandled exceptions, need IT assistance to resolve

Setup a standard block of code you can copy and paste into you code when you start development.

Define global constants

```
  g_retcode_normal  VARCHAR2(2000) := '0';
  g_retcode_warning VARCHAR2(2000) := '1';
  g_retcode_error   VARCHAR2(2000) := '2';
```

Here is a sample block of code. Start development with this. Notice that I am setting the initial values to assumed non-error states.

```
PROCEDURE SUBMIT_TEST(p_errbuf              OUT VARCHAR2,
                      p_retcode             OUT VARCHAR2,
                      p_source_name         IN  VARCHAR2,
                      p_reset_all           in varchar2
                     )
IS
```

```
  v_source_name_code      VARCHAR2(100);
BEGIN
  g_retcode := g_retcode_normal;
  g_errbuf := NULL;
  g_error_cnt := 0;
```

Put you actual code here

```
  IF (g_error_cnt > g_retcode_normal)
  THEN
    p_retcode      := g_retcode_warning;
    p_errbuf       := 'See Output File for Error Messages';
    OUT ('Please review out and log files for errors');
  ELSE
    OUT ('Test Scenarios Data Generation successful, no errors encountered');
  END IF;
END SUBMIT_TEST;
```

22) PL/SQL Result Cache

I prefer writing a bunch of small self-contained PL/SQL functions in my code. That allows me to unit test them and expose them for other debugging purposes. However, this sometimes affects performance we many repeated calls to get the same data over and over again.

As of Oracle 11G DB you can have your PL/SQL Function cache results based on a set of Input Parameters. Oracle handles this caching for you automatically in memory. You specify what causes the cache to be invalidated in the package body RELIES_ON clause.

Make sure your input parameters are unique enough to correctly define the values, you don't want to cache an AMOUNT column since that will just waste memory. Also Cache the highest level of data you can. If you have a variety of nested functions cache the outer most result so your code does not have to do that processing over and over again.

Be aware how your code is run. If multiple processes are using the cached function result they will all try to add values to cache. This can cause Latch Waits, so be careful how you use this functionality.

Package Spec

```
FUNCTION MY_LOOKUP (p_condition1    VARCHAR2,
                    p_condition2    VARCHAR2,
                    p_condition3    VARCHAR2)
RETURN VARCHAR2
      RESULT_CACHE;
```

Package Body

```
FUNCTION MY_LOOKUP (p_condition1    VARCHAR2,
                    p_condition2    VARCHAR2,
                    p_condition3    VARCHAR2)
  RETURN VARCHAR2
  RESULT_CACHE RELIES_ON (MTL_CATEGORIES_B)
IS
BEGIN
……..
END MY_LOOKUP;
```

Don’t Hard Code Configure Instead

23) Don’t Hard Code

This might seem obvious, but often it happens because you can’t find a place to store or get a value. If you ever find yourself assigning values in your code, either strings or numeric, ask yourself is there some place else I can get this data. If not seeded or currently available then consider the following optins:

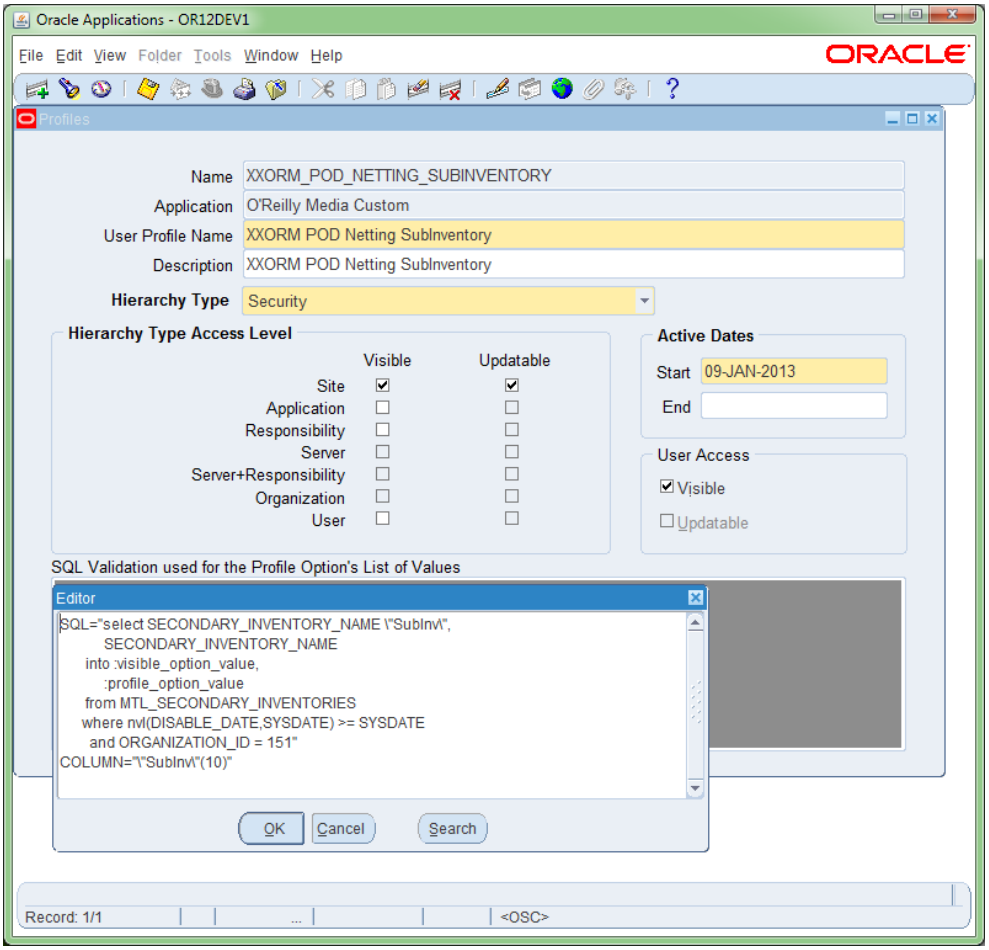
- Custom Profile Option
- Descriptive Flex Fields, DFF
- Key Flex Fields, Inventory : Item Categories

We will cover each of these in the following sections in more detail.

24) Custom Profile Options

Custom profile options allow you to setup and configure data once. And if requirements change you can easily adjust the values. This is especially handy when you have data that changes when PROD is cloned to a non-PROD instance. In addition, the data entered in the profile option value can be validated using the SQL Validation entry. You can optionally set the profile option value at various levels like: site, application, responsibility and user level.

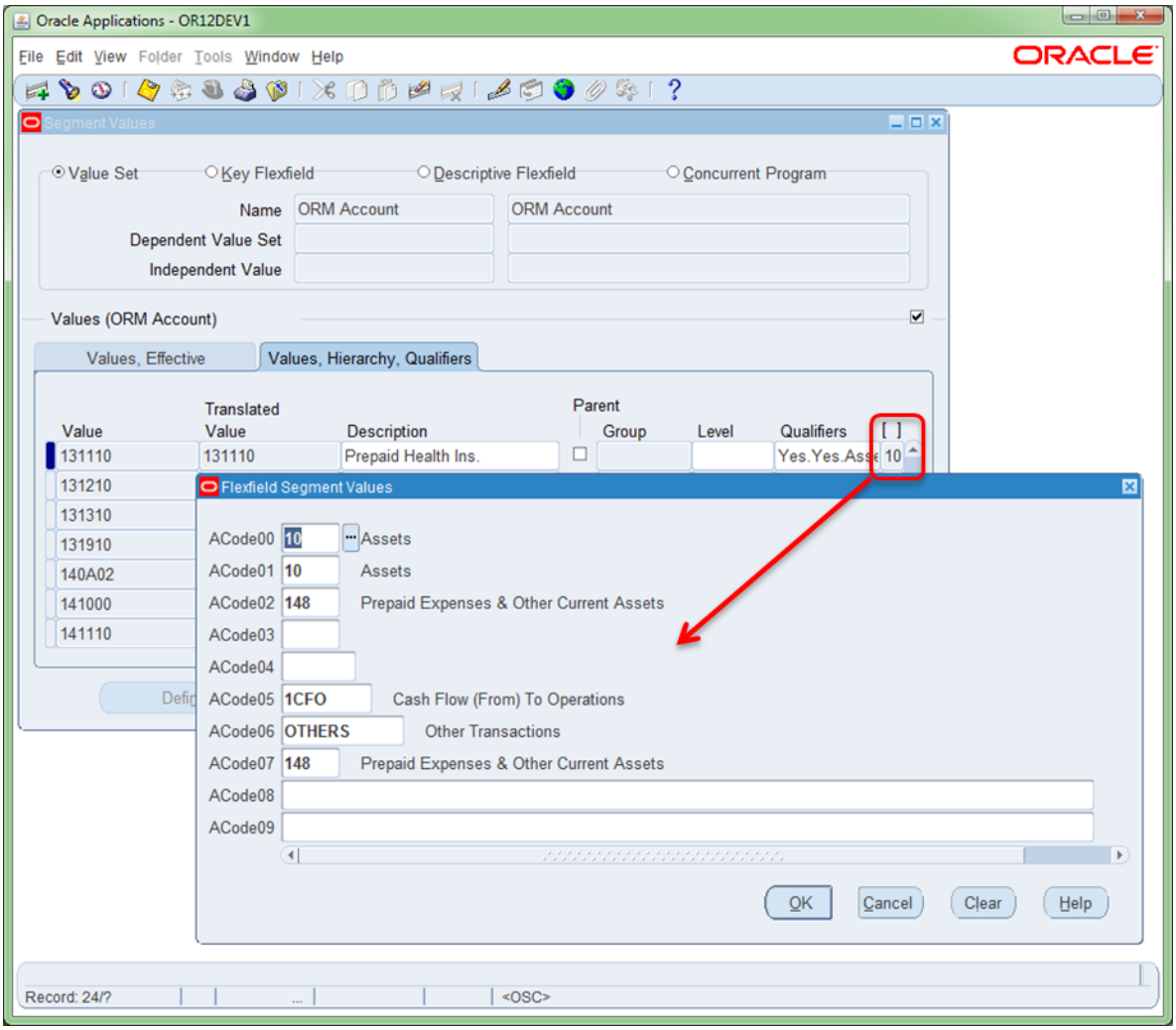
Responsibility: Application Developer
Navigation Path: Profile Option



25) DFF on Value Set Values

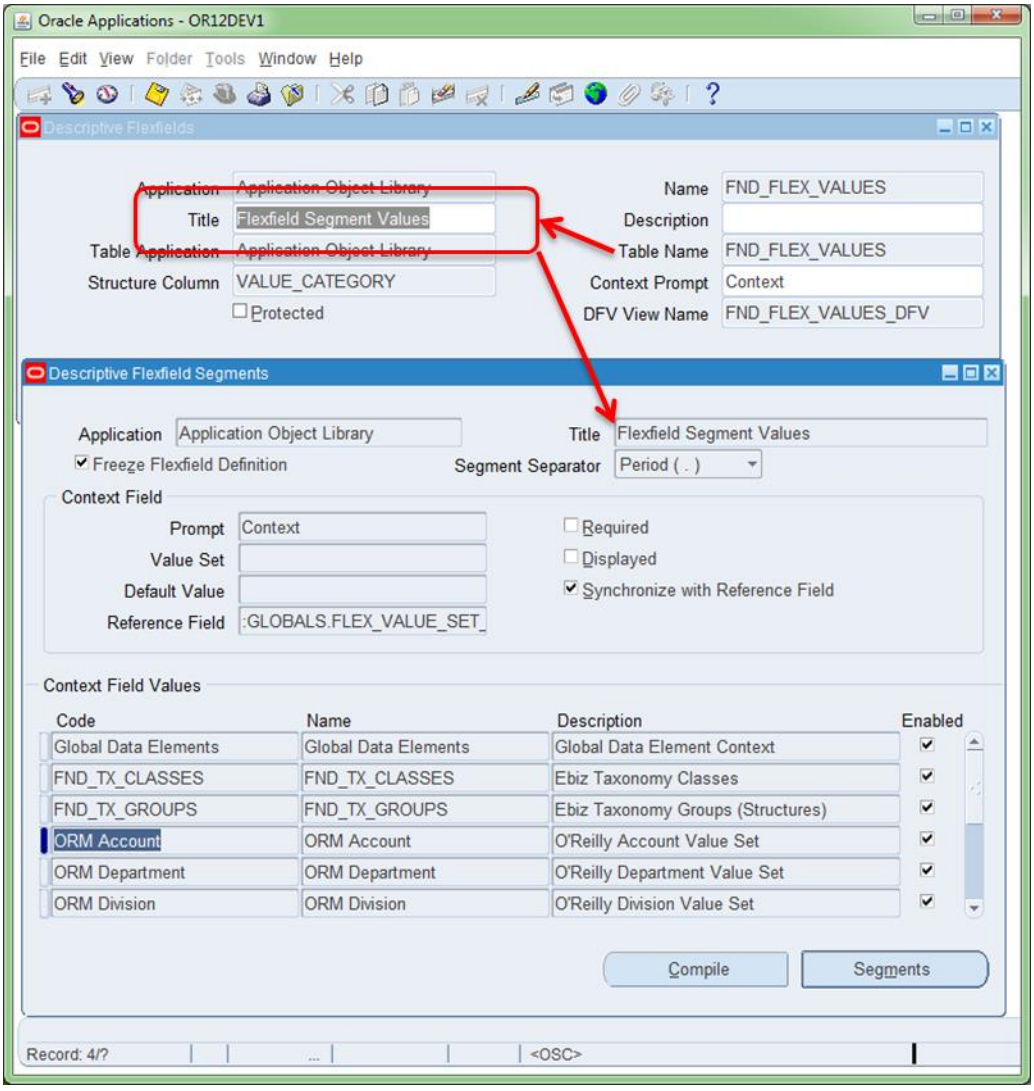
Obviously you can define DFF's on existing EBS data tables. This is a great way to add additional data elements that relate one to one with an existing row of data. What if you want to add a DFF to variable user entered data, like a Value Set Value? This is a great way to force additional data to be captured when a value set value is being created. Now when that value is used anywhere in EBS you have this extra dimension of meaning that can be used for processing logic or reporting purposes.

In this example below the GL Account values have additional coded meanings added to them. You have to be careful not use this when other functionality like Hierarchies might be more appropriate, but this is a great way to add data elements to a single value.



The Value Set Values DFF is called: 'Flexfield Segment Values'
Make this context sensitive, with a Reference Field: :GLOBALS.FLEX_VALUE_SET_NAME

Each context sensitive DFF column can be optionally required, and validated values come from Value Sets as well.



This gives you one extra dimension to a given Value Sets Values, up to 50 additional columns of data per value. The DFF can be context sensitive so it has different meanings for each value set. Now when a user assigns a value to the GL Account they must fill in the DFF columns providing additional details for programs and reports to use. This places the users in control of their data. Removes dependency on programs or IT staff.

Again don't abuse this feature and try to use standard functionality where possible.

26) Item Categories Structure

The above DFF on value set values give you a single dimension of data. What if you need additional dimensions of data? What if you have a requirement for more than one condition columns to provide multiple result values? Then use the Inventory Item Category Structure Key FlexField. This provides 20 segments of data that can be validated using Value Sets. There is a UI to enter data, use the Item Category Codes screen to set values.

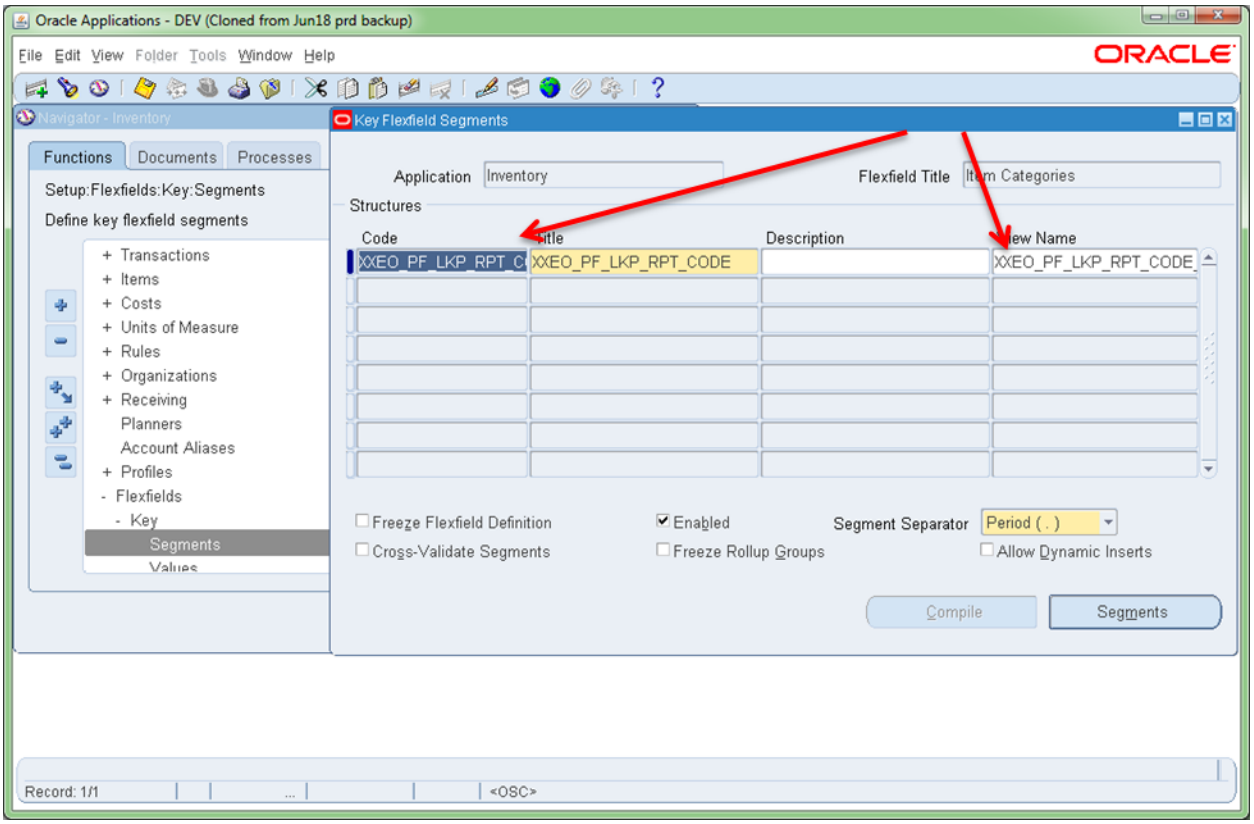
You might ask if this will confuse inventory users. The simple answer is that we are only defining the category structure. We are not assigning the Category Structure to an Item Category Set so it will not confuse or impact the Inventory users.

Here is an example of how to use this functionality. Let's say we have 9 different conditions that will uniquely define 2 result values. Category Structures have 20 different columns so I will map condition columns to Segments 1-15, and results to Segments 16-20.

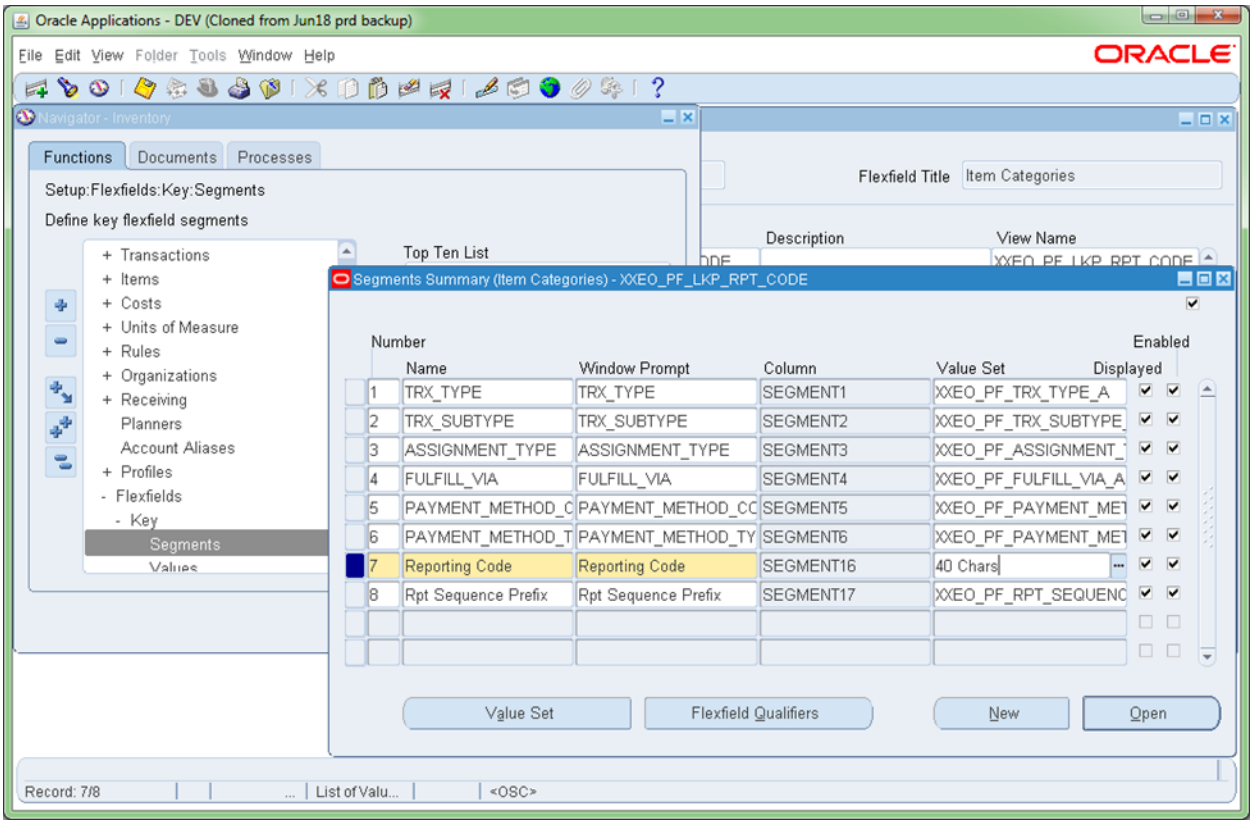
Segment	Prompt/Description	Value Set	Required	C/R
1	TRX_TYPE	XXEO_PF_TRX_TYPE_A	Y	C
2	TRX_SUBTYPE	XXEO_PF_TRX_SUBTYPE_A	Y	C
3	ASSIGNMENT_TYPE	XXEO_PF_ASSIGNMENT_TYPE_A	Y	C
4	FULFILL_VIA	XXEO_PF_FULFILL_VIA_A	Y	C
5	PAYMENT_METHOD_CODE	XXEO_PF_PAYMENT_METHOD_CODE_A	Y	C
6	PAYMENT_METHOD_TYPE	XXEO_PF_PAYMENT_METHOD_TYPE_A	Y	C
7	ACCOUNT	XXEO_PF_ACCOUNT_A	Y	C
8	AMOUNT	XXEO_PF_SPECIAL_STRINGS	Y	C
9	VAT_COUNTRY	XXEO_PF_COUNTRIES_A	Y	C
16	Reporting Code	40 Chars	Y	R
17	Rpt Sequence Prefix	XXEO_PF_RPT_SEQUENCE_PREFIX	N	R

Define you category structure key flexfield. Enter a View Name, and EBS will automatically build a view with column names that match you category structure. The View Name should be the Code value with an '_V' suffix. As an example the names I am using are:
XXEO_PF_LKP_RPT_CODE => XXEO_PF_LKP_RPT_CODE_V

Key Flex Field => 'Item Categories'



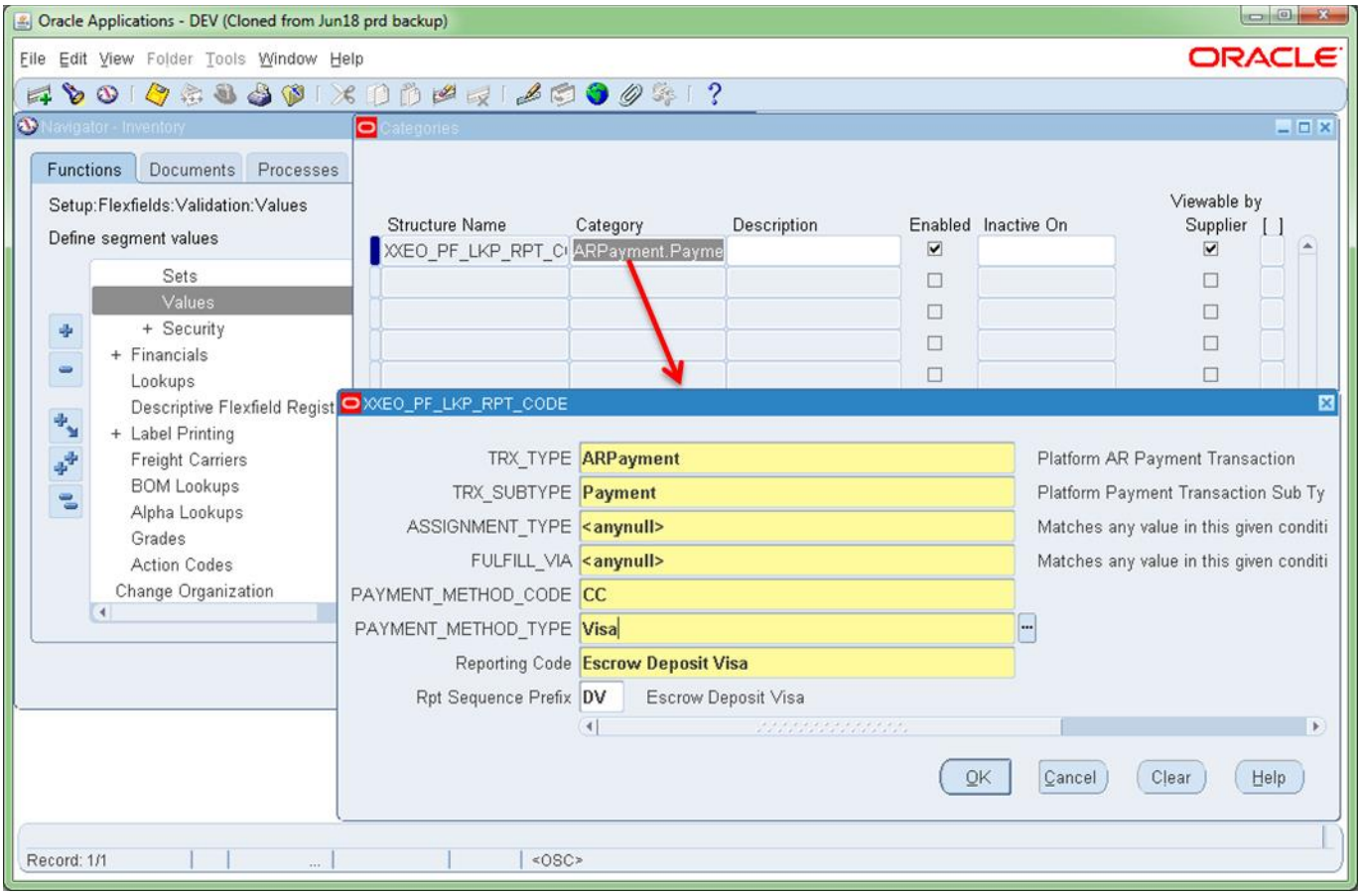
Define your Category Structure segments, you have 20 segments only. Each Segment is limited to 40 characters. Follow a convention for naming or organizing columns: conditions on left, results on right. You also convention for prompts for condition and result columns so users know what they are used for.



Save and compile the flexfield.

Now we can see what entering values in into the Category Structure Codes looks like. Notice the <anynull> values, more on that later. Each segment is validated based on value sets so we will end up with really clean and reliable configured data.

Responsibility: Inventory
Navigation: Setup : Categories : Category Codes



Now that we have this nice 2 dimensional matrix of configuration data entered how do we use it. We can write a single PL/SQL utility package that can be used to search a category structure's condition columns and return the best matching result columns. This can be a generic procedure because you can assume:

You have 1 to 19 condition columns, at least 1 result
You have 1 to 19 result columns, at least 1 condition

Mapping of the columns to your specific data set is in the call to the generic function.

When we perform the comparison we can take into account fuzzy condition columns, with values like: <any>, <anynull>, etc
We always take an exact match first. When we have fuzzy conditions they are weighted left to right, left segments are the most important conditions.

We can use PL/SQL Results Cache to improve performance of this search. Once a given set of conditions and results have been evaluated they results are stored in memory for fast retrieval.

You can now configure instead of changing hard coded values. You don't need to create a custom table and form to hold a matrix of conditions and results. You can use a standard EBS form and value set validation capability to hold the data. This empowers your users to configure the programs behavior without involving IT.

27) Value Set Values

How do we take a Value Set's values and add some static constants? I want a value set that contains EBS Currency Codes, and some special strings, like the examples above of <any>, <anynull>, etc.

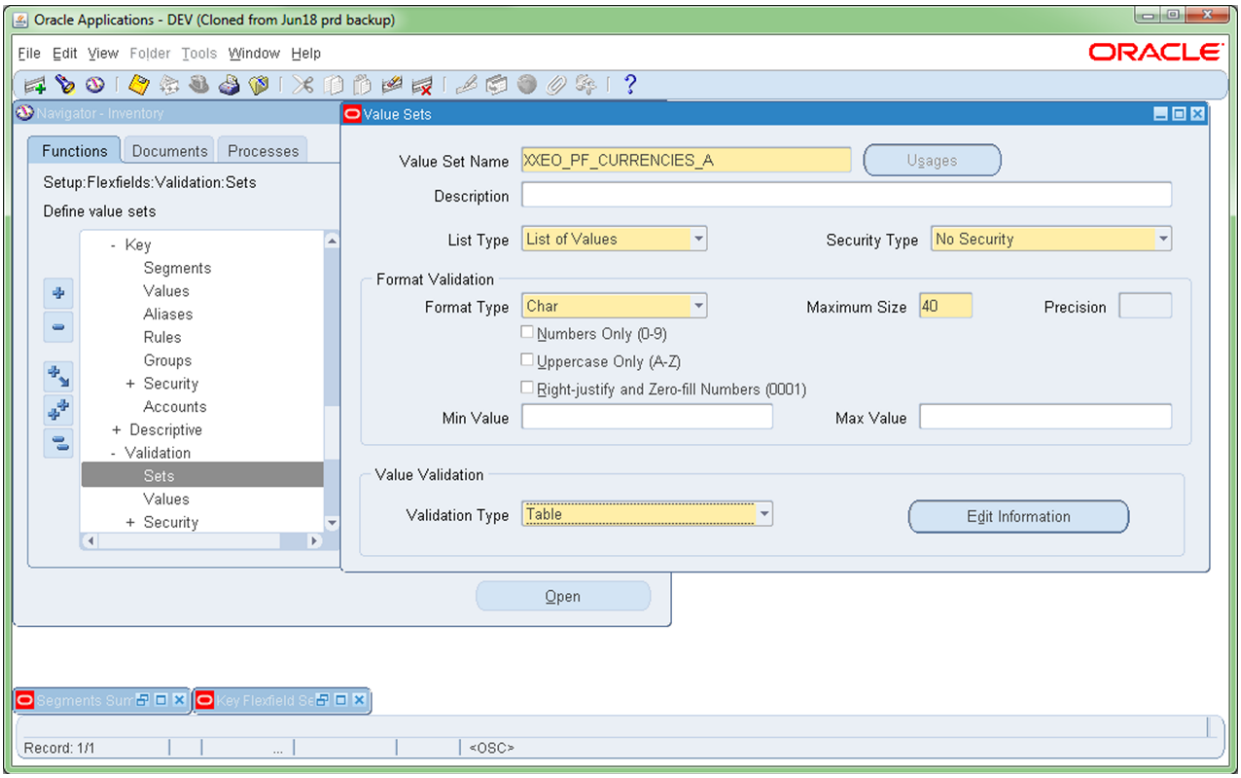
We first need to Create a Value Set that holds the static constants in this example I am naming this value XXEO_PF_SPECIAL_STRINGS. The table below shows the values and what meaning those values have.

String	Description
<any>	Matches any value in this given condition
<anynull>	Matches any value in this given condition, including NULL values
<null>	Matches only rows with a NULL condition
<gt0>	Greater than zero
<ge0>	Greater than or equal to zero
<eq0>	Equal to zero
<ne0>	Not Equal to zero
<le0>	Less than or equal to zero
<lt0>	Less than zero

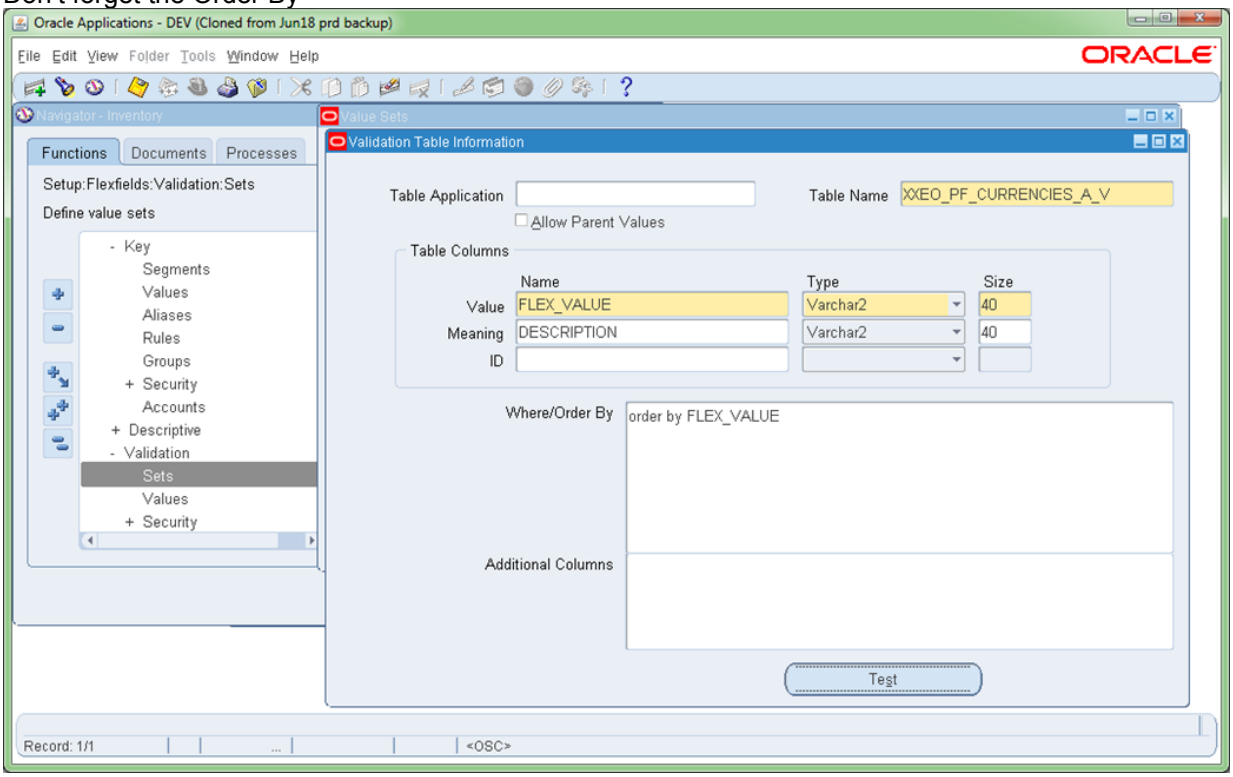
Now Create a view that unions in the base values and static constants. For this example I am calling my view XXEO_PF_CURRENCIES_A_V. Here is the code.

```
CREATE OR REPLACE FORCE VIEW APPS.XXEO_PF_CURRENCIES_A_V
(
   FLEX_VALUE,
   DESCRIPTION
)
AS
   SELECT CURRENCY_CODE, NAME
     FROM FND_CURRENCIES_VL v
    WHERE CURRENCY_FLAG = 'Y'
      AND SYSDATE BETWEEN NVL (START_DATE_ACTIVE, SYSDATE)
                      AND NVL (END_DATE_ACTIVE, SYSDATE)
   UNION ALL
   SELECT FLEX_VALUE, DESCRIPTION
     FROM XXEO_PF_VALUE_SET_VALUES_V
    WHERE FLEX_VALUE_SET_NAME = 'XXEO_PF_SPECIAL_STRINGS';
```

Now create another Value Set that is table based on the view we just created. In this example this Value Set is named XXEO_PF_CURRENCIES_A. Notice Format Type: Char and Max Size: 40, remember that category segments are limited to 40 characters in length if you are going to use this value set for your category segment values.



View columns:
FLEX_VALUE = Actual value code we will store in the category structure
DESCRIPTION = Verbose description to help the user select
Don't forget the Order By



The new List of Values will have the constant strings first:
<any>, <anynull>, etc
Then the EBS Currency Codes. The list will be sorted alphabetically

Use New Technology

28) Oracle Reports Replacement

If you ever find yourself having to open up Oracle Report RDF to make a change ask yourself why? You should instead consider porting this to BI Publisher. You can extract the SQL from the RDF and use BI Publisher to execute the SQL, generate an XML data stream and format using a template.

Take a look at the following My Oracle Support Note for additional details.
How To Migrate Reports To XML Publisher?
(Doc ID 1329693.1)

29) Oracle Forms, OA Framework Pages

If you ever need to create a custom data entry screen don't use Oracle Forms or a custom OA Framework Page, use Applications Express (APEX) instead. APEX can be used to create HTML data entry screens or reports. You can use any browser on desktops, tablets and smart phones to view and enter data.

APEX tightly integrates with Oracle EBS and is recommended for light weight EBS extensions by Oracle's ATG team.
Extending Oracle E-Business Suite Release 12.1.3 and Above Using Oracle Application Express (APEX)
MOS Note: 1306563.1 (Mar-2015)

I have several other papers on using this tool for custom development.

Extend EBS Using Applications Express
http://jrpjr.com/paper_archive/Collab14_Extend_EBS_Using_APEX.pdf

Migrate your Discover Reports to Oracle APEX
http://jrpjr.com/paper_archive/collab15_disco_to_apex.pdf

Best of all it is Free if you have licensed the DB for development activities.

30) Custom Web Application Desktop Integrators

If you need to integrate Excel into EBS consider using Custom Web Application Desktop Integrators. Oracle now allows you to create Custom WEB ADI's. This can be used for data entry and reporting.

Oracle® E-Business Suite Desktop Integration Framework Developer's Guide Release 12.2 Part No. E22005-10

MOS Note: Oracle Web Applications Desktop Integrator Documentation Resources, Release 12 (Doc ID 396181.1)

Check out my other papers on the topic as well.

Custom SubLedger Accounting JE WebADI I/F
http://jrpjr.com/paper_archive/collab11_custom_webadi_pres.pdf

References

The following are references related to workflows and approvals that might be beneficial to you.

1) My Web Site and Past Presentations
http://jrpjr.com

2) R12.2 Development and Deployment of Customizations
http://jrpjr.com/paper_archive/collab15_R12_2_dev.pdf

3) Extend EBS Using Applications Express
http://jrpjr.com/paper_archive/Collab14_Extend_EBS_Using_APEX.pdf

4) Migrate your Discover Reports to Oracle APEX
http://jrpjr.com/paper_archive/collab15_disco_to_apex.pdf

5) Custom SubLedger Accounting JE WebADI I/F
http://jrpjr.com/paper_archive/collab11_custom_webadi_pres.pdf

6) Custom Web ADI Integrators
http://jrpjr.com/paper_archive/5_12_peters.pdf

Conclusion

I hope you have found this helpful. While many of these topics may have been obvious I felt it was important to at least cover them initially since I don't know the level that the reader is coming into this at.