Application Development and Maintenance

Application Development and Maintenance Procedures

Copyright and Trademarks

AXA Applications Development and Maintenance

This guide supports release number 01

Revision number 02, date: Dec 29, 2004

AXA Financial, Inc.
1290 Avenue of the Americas, 27th Floor
New York, NY 10001

Phone: 212.xxx.xxxx / FAX: 212.xxx.xxxx

http://www.axa-financial.com

Contents

Table of Figures

Preface

The functional area of application development and maintenance is responsible for both the development of new applications and the on-going maintenance of existing applications. This latter responsibility includes enhancements and bug fixes. This functional area is responsible for creating and maintaining high quality applications and for maintaining the user confidence in the applications and the department.

The Application Development and Maintenance standards is divided into the following categories:

- General Procedures
- Specifications
- Application Development Safeguards
- Development Environment
- Development Tools Standards

1.1 Conventions

This document uses the following conventions.

Convention	Meaning
Bold	Used in procedures to indicate the name of a user interface object such as a menu or button. Also used in procedures to indicate text that you type in a user interface object such as a text box.
Italic	Used to indicate a glossary term or book title.
Select	Used in a procedure to indicate that you should select a user interface object, such as an item from a list, or a check box.
Click	Used in a procedure to indicate that you should click the left mouse button to select a user interface object, such as a button tab, or shortcut menu.
Underline	Used to show a URL or text hyperlink

1.2 References

This document describes how IT security is conducted and the standards and procedures that are followed for each phase. This guide references the following related software products and documentation:

Guide	Author
AXA Financial	AXA Financial, LLC
Microsoft©	Microsoft, Inc.
SQL Server2000©	Microsoft, Inc.
UNIX	
RCS	Revision Control System
Oracle	Oracle Corporation, Inc.
Microsoft WORD©	Microsoft Corporation

1.3 Sources

The following sources were used to create this document:

- Microsoft Manual of Style for Technical Publications – Third Edition, Microsoft

1

General Procedures

General Procedures

The functional area of application development and maintenance is responsible for both the development of new applications and the on-going maintenance of existing applications. This latter responsibility includes enhancements and bug fixes. This functional area is responsible for creating and maintaining high quality applications and for maintaining the user confidence in the applications and the department.

1.4 General Procedures

For a system modification or enhancement to make its way to a production environment, it must follow a defined path.

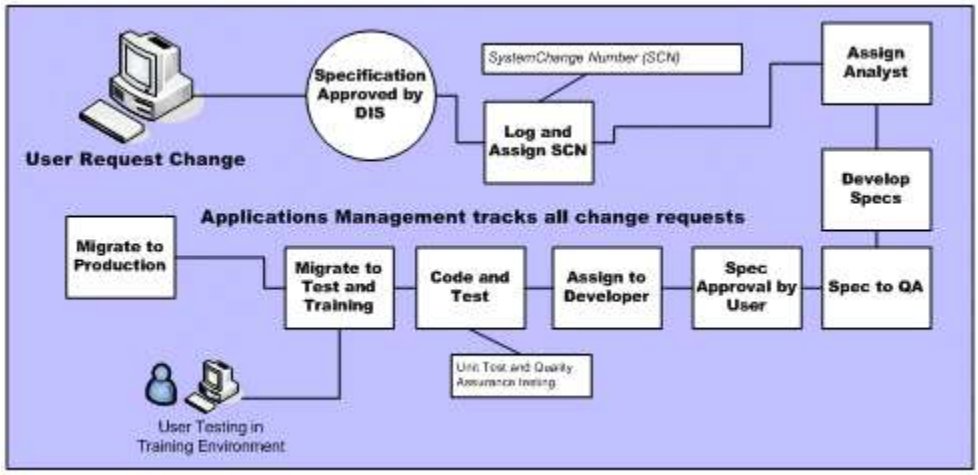

Figure 1: General Development Procedures

- User (sponsor) requests an application change

- DIS approves the change specifications for development
- Log, assign an SCN to the change request, and give the SCN to a developer
- Developer codes and unit tests the SCN
- Quality Assurance (QA) reviews for acceptance
- Changed code is moved to test environment
- Successful testing moves the code to the training environment – Users are trained
- The change is moved to the Production environment
- The SCN is tracked by the Applications Management Group or any other group performing this type of change management monitoring

1.5 Specifications

This section will discuss specifications for modifications or enhancements to existing applications. While it can be applied to the development of new applications, it should not be considered a guide to systems development. Standards and procedures for new application development are the subject of many excellent texts, and depend on the application development method used.

1.5.1 User Sponsorship

Every specification must have a single user sponsor. Other users may be involved in the definition of the specification, but there must be a single user sponsor. A user sponsor is responsible for:

- Initiating a modification and/or enhancement request
- Justifying the change
- Being the primary user-side point-of-contact for the analyst
- Approval of the specification

The user sponsor may come from any functional area (even DIS) but must have or be granted sufficient authority to define and approve the specifications.

1.5.2 Specification Content

All specifications will be developed using Microsoft WORD. They will be kept on the network server T20NETCC in the directory K:\SPECS. The filename will be the SCN number, i.e. scn1234.doc, scn0012.doc, etc.

Title Page

Every specification will have a title page. This title page will list:

- The title for the change
- The system change number (SCN)
- The user sponsor's name
- The analyst's name

Each specification will address itself to describing the change from the functional perspective and from the technical perspective.

Functional

The functional portion of the specifications will include a business reason for the change and a functional description of the change. Business reasons explain the benefit of the change and/or why it is required. Some examples of business reasons are:

- Regulatory Compliance
- Support of new or revised business processes
- Measurable time or cost savings

Functional descriptions explain the behavior of the modified application. These descriptions should avoid table names, column names and other specifics of this type.

For example, "A new data element, WALL_HT_FT, will be added" becomes "With this change, the user will be able to enter the wall height, to the nearest foot".

Technical

The technical portion of the specification explains information the programmer needs to implement the change. Examples of this kind of information are as follows:

- Module names for affected reports/screens
- Pertinent information determined during the feasibility phase
- Details of a TABLE JOIN to insure the ROWs SELECTed are the desired ROWs
- All TABLEs required fo a new report. If three TABLEs are needed, then three TABLEs should be specified

Extraneous information to explain TRIGGER changes in a form is not be necessary unless this process was part of the required research.

- Estimated hours required to complete development and testing

Functional

This is the portion of the specification the user sponsor is required to approve, so it must be written in terms he/she will understand.

The new behavior must be described completely. This includes how the user sees and interacts with the data, data flow, processing, etc. Be sure to include any limitations or constraints on the user or the system (e.g. data will be kept for one year, data is frozen after a certain process is run, etc.). It is imperative that the analyst confirm that the described behavior is technically possible and technically desirable. For example, this could include research validating the implied relationships, or analysis of the processing load required.

Define all security requirements. Provide sufficient detail of who will have access to the data and the manner of access.

If the change alters or creates some object (screen or report) the user sees, mock-ups of the changed object must be included. If the change alters an existing object, show both the "before" and "after" versions. For reports, be sure to include examples of breaks, sub-totals, totals, etc.

The analyst must also include guidelines on how to test the change, such as limits, special exceptions, error handling requirements, etc.

1.5.3 Specification Quality Assurance and Approval

All specifications will go through several levels of quality assurance.

Once the analyst has developed a full understanding of the change, he or she will review the requirements with the lead developer of the affected application. The purpose of this review is to:

- Notify the lead developer of the change.
- Determine the soundness of the analyst's proposed approach and to suggest improvements to the approach.
- Identify any areas of risk.
- Identify any other work in progress impacting the change. Determine the course of action to resolve conflicts or interdependencies.

The lead developer must sign-off on the analyst's approach.

Once the lead developer has signed off, the analyst should complete the specification. Once completed, the specification should be turned over to the quality assurance group for review. The quality assurance group reviews the specification for:

- Clarity - Is the specification easy to understand and unambiguous?
- Consistency - Is the use of data elements, terminology, etc. consistent with the AXA Client Solutions definitions of those elements or terms? For example, would this specification allow values to be entered for an attribute in violation of the definition of that attribute?
- Adherence to standards - Is the specification consistent with the specification standards, user interface standards, report standards, etc.?

- Completeness - Does the specification completely describe the change?
- Presentation - Are there misspelled words, incomplete sentences, grammatical errors, etc.?

This step is an iterative process. If the specification does not pass quality assurance, it is returned to the analyst, who will make revisions, resubmit it to quality assurance, and so on until the specification passes quality assurance.

Once quality assurance signs off on the specification and logs its approval, the specification is returned to the lead developer who performed the initial review. This individual reviews the specification to:

- Revalidate the technical approach.
- Ensure that all failure points have been determined and addressed.
- Ensure that all implementation issues such as processing time have been addressed.

As is the case with quality assurance, this can be an iterative process. The lead developer may return the specification to the analyst for revisions if he or she is not fully satisfied with the specification. The analyst then makes the revisions, gives it back to the lead developer, who reviews it, and so on until the developer signs off on it. It is not necessary to route the specification back through the quality assurance group unless a substantial rewrite (in the opinion of the lead developer) was required. The lead developer should make himself or herself available to the analyst as specifications are developed to help avoid cases where major rewrites are required.

Once the lead developer has signed off on the specification, he or she logs the approval and returns it to the analyst. It should now be presented to the user sponsor for approval. The user sponsor should not be shown the specification prior to this point, in order to avoid incorrectly setting expectations. Parts of the document, such as screen or report mock-ups may certainly be used to clarify needs or confirm understanding prior to this point. When the specification is delivered to the user sponsor, DIS is in essence committing to deliver the work described in the specification document.

In some cases, the user sponsor will request that changes be made to the specification. Changes of a technical nature must be reviewed with the lead developer. Changes that affect standards compliance must be reviewed with the quality assurance group. Minor changes, such as slightly rewording the business case for the change, may be made by the analyst without additional review by quality assurance or the lead developer.

Eventually, the analyst produces a specification document that the user sponsor will sign. This signed original is turned over to the application management group to be filed. Since the document is kept in electronic form on the network, copies may easily be obtained.

As the change is programmed, the programmer may find it necessary or advisable to implement the change in a manner different from that specified by the analyst. After consultation with the lead developer (and the analyst and user sponsor if necessary),

the programmer may make such changes and note them in the specification. At the end of each specification, there will be a "Programmer's Notes" area. If the programmer did not have to depart from the specification, then he/she should put "No changes necessary" in this area. If changes were required, then those changes must be noted in this area.

1.6 Application Development Safeguards

"An ounce of prevention is worth a pound of cure". Any systems development organization must implement safeguards to protect users and developers. This section will describe these safeguards at AXA Client Solutions.

1.7 Development Environment

"An ounce of prevention is worth a pound of cure". Any systems development organization must implement safeguards to protect users and developers. This section will describe these safeguards at AXA Client Solutions.

Multiple Databases

Each Oracle installation will have at a minimum two related databases:

- Production
- Training

Each installation where there will be development work will also have two additional databases:

- Testing
- Development

Each database will have its own set of application code. When an upgrade is performed to the RDBMS, the upgrade will be migrated across the databases, from development to production (see Section 1.1.1 for more on the guidelines for RDBMS upgrades).

The production database will contain all the "live" data and the production application code.

The training database will contain a subset of the production data and have its own application code. This database will vary from the production database when there are new features upon which to train users. When there are no such features, the data schema and application code will be identical to production, so that new users can be trained. On a regular basis, the data in this database will be re-created as a sub-set of production data.

The testing database will only be present if the site is performing application development. Its purpose is to serve as an integrated test environment for new code. Once new or modified code and database objects have been unit tested in the

development database, the changes will be moved to this database for integrated testing. This instance will contain a sub-set of the production data plus data added for testing purposes. When the data in this database does not adequately support the testing function, it can be re-created from the production database.

The development database will only be present if the site is performing application development. Its purpose is to host the development of new functions. Modifications and enhancements will be developed and unit tested in this database. It will be seeded with a subset of production data, but will not be regularly refreshed with production data.

1.7.1 Source Code Control Procedures

Revision Control System (RCS)

The UNIX utility RCS (Revision Control System) will be used to provide source code control. On platforms where RCS is not available, another version control system will be used. No development will be performed without a version control system in place.

RCS Files Application Directory

/usr/src/<application>/rcs

<application> is the three-character application abbreviation

Large RCS directories can be broken down by type

rcs/forms - all forms

rcs/reports - all reports (Oracle Report)

rcs/sql - all sql scripts

rcs/c - all C, Pro*C programs

rcs/misc - shell scripts, etc.

All Oracle Forms and Oracle Reports will be stored in text format (e.g. .FMT) as opposed to the binary format.

Each code module that is checked into the RCS directory will have at a minimum the following comments in the source code:

- The system change number (SCN) for the change/new feature
- The maintenance history
- The name of the programmer
- The check-in date

In addition, the reference number, programmer name, date, and a brief comment will be provided as check-in comments.

All developers will be able to check out source code from the RCS system. However, the ability to check source code back into the RCS system will be limited to the code migration coordinators (see Section 3.3.3 for more information on code migration).

1.7.1.1 Source Code Migration Procedures

General Environmental Migration Procedures

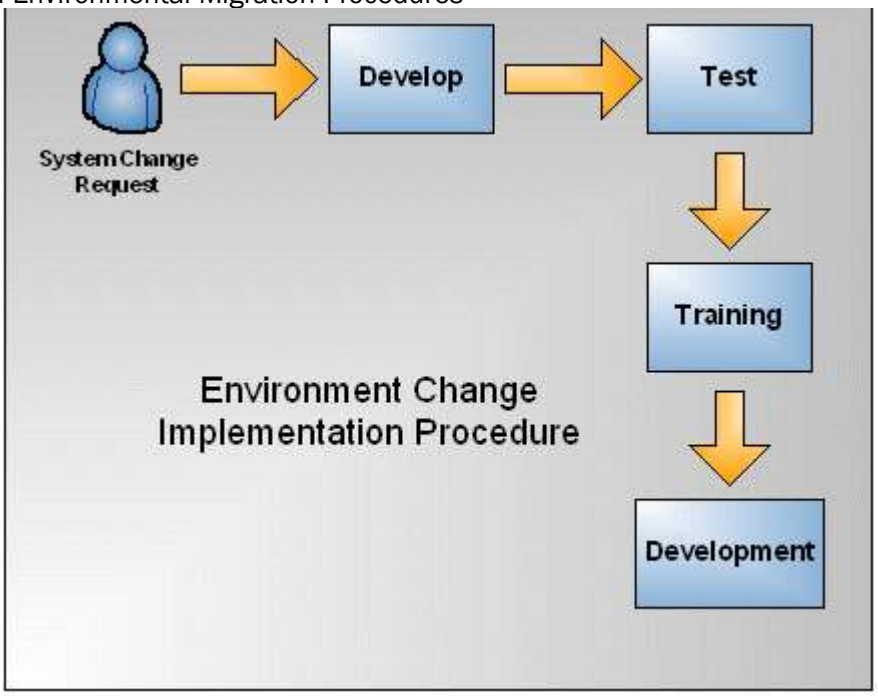

Figure 2: High-level Migration Procedures

All system changes will originate in the development environment; migrate to the test environment, then the training environment, and finally the production environment. Throughout the migration period, the developer retains primary responsibility for the change.

Developer/Programmer Change Migration Procedures

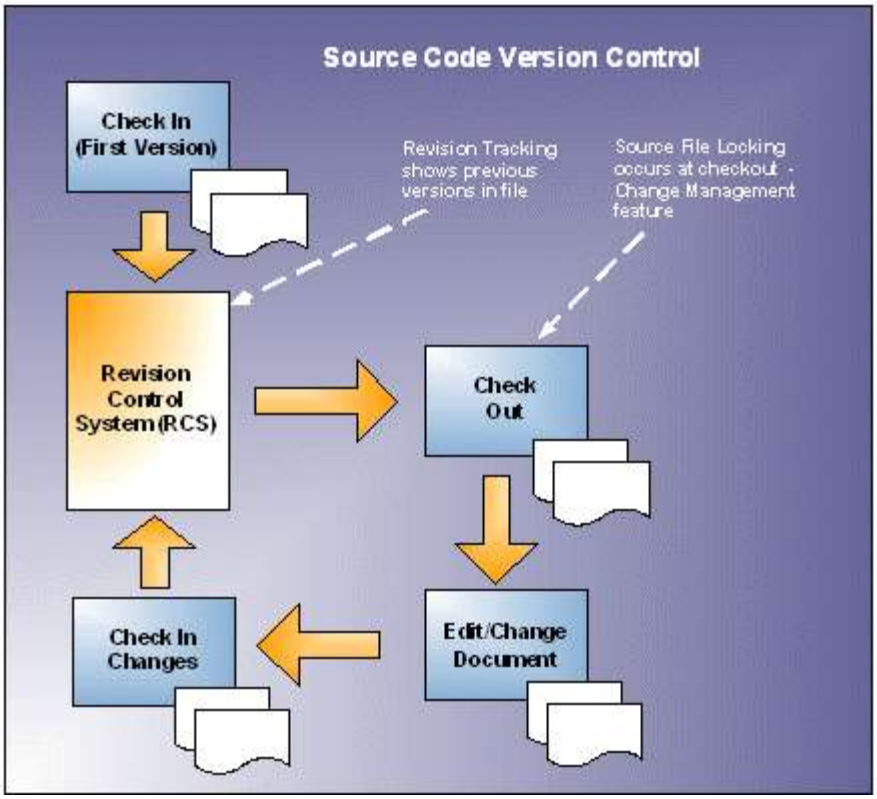

Figure 3: Revision and Version Control Procedures

When the programmer is ready to begin work on a module he/she will check it out of the appropriate rcs directory, locking the file to prevent other check outs for edit. The programmer will check out the file into a directory under his/her home directory. If the development is in a client-server environment, the developer may transfer the file to his/her PC/workstation. Edits to the file will be done either in the developer's home directory (or sub-directory) or on the developer's workstation.

If the modification requires altering or creating database objects, these will be altered/created in the programmer's schema. To add a column to an application table, for example, the programmer will create a copy of the application table in his/her own schema, and test the alter table script and other changes against that copy of the table. (Note: depending on the environment, altering a table may be necessary, but so doing may cause chaining.)

When code QA and unit test has been successfully completed for the modification, the developer will copy the modified or new source code to the

/usr/src/<application>/checkin

directory and will submit a Code Migration Form to the migration coordinator. The Code Migration Form (CMF) informs the migration coordinator that code is to migrated from one database to another and of any special procedures that must be accomplished with the move (such as running an alter table script). The senior

programmer responsible for code QA must authorize the code to be moved to the test environment. See Appendix E for a sample Code Migration Form.

Migration Coordinator Procedures

At this stage in the migration, the migration coordinator will:

- check to make sure all of the source code is in the checkin directory (usr/src/<application>/checkin)
- verify that the source code header comments are complete
- check the code back into RCS (if it is existing code) or create a new RCS library (if it is new code)
- compile the code to create the executables
- if required, coordinate with the DBA for the DBA to run any scripts containing DDL
- move the executables to the testing directories
- log the file names and version numbers migrated
- notify the appropriate individual (QA) that the change is available to test
- notify the developer that the change has been migrated

This procedure accomplishes several goals:

1. Reduces the number of unnecessary RCS check-ins, thereby reducing the number of wasted version numbers and speeding retrieval of older versions, and helps prevent the RCS directory from becoming cluttered with unneeded files.
2. Ensures the executables match the source code in the library.
3. Coordinates the movement of code with other activities needed to correctly install the change.

If during the testing phase additional changes must be made to the code, the developer must check out the source, modify it, complete a unit test, and submit a new CMF to move the code into the testing environment. This will create a newer RCS version of the code.

- Once the modification has been successfully tested, the developer will submit a second CMF requesting that the code be moved to the training database. This move must be authorized by the QA department. The migration coordinator will then:
- coordinate with the DBA for the DBA to run any scripts containing DDL.
- copy the executables from the test environment to the training environment
- log the file names and version numbers migrated
- notify the user sponsor that the change has been migrated to training
- notify the developer that the change has been migrated to training

- if any database objects have been created or altered, notify Data Administration of the change

The user sponsor is responsible for ensuring that the appropriate end users are trained prior to the change being moved to production. Some changes will not require additional user training, while others will require a large number of users to be trained. When the user sponsor is satisfied that the user training is complete, he or she will notify the migration coordinator to move the change into production. However, the user sponsor must be given a deadline for completion of training. Changes should not be allowed to accumulate in the training or testing environments. Once training is complete, the migration coordinator will then:

- coordinate with the DBA for the DBA to run any scripts containing DDL.
- copy the executables from the training environment into the production environment
- log the file names and version numbers migrated
- notify the user sponsor that the change has been migrated to production
- notify the developer that the change has been migrated to production

1.7.1.2 Guidelines

The following migration guidelines should be followed:

- A code module or database object may not be modified if a previous modification to that module has not reached production. For example, if a form is part of two different modification requests, the first modification must be migrated completely before the second request can begin. If possible, it is advisable to delay the beginning of work on the second request for at least a week after the first request reached production.
- Migration into the test environment should be on a regularly scheduled basis (Friday afternoon, for example). This will allow testers to complete testing prior to the introduction of new code. The DBA may have to acquire the database to run DDL, and this could be scheduled for the morning of the same day on an as-needed basis.
- Migration into the training environment must occur at times when no training is occurring.
- Migration into production should also be on a scheduled basis. DBA access will often have to occur at night when no users are logged on to the production database. Code migration should occur on a set day (Wednesday?) each week.

1.8 Development Tools Standards

User Front End

Oracle Forms will be the tool used to develop end-user screens.

Reports

Oracle Reports will be the primary report development tool. Pro*C will be used to develop reports where complex logic or special performance requirements warrant its use.

Ad-Hoc Query

Oracle Data Browser will be the end-user ad-hoc query tool supported by DIS.

There are many other query tools. While DIS does not prohibit their use, it will not provide support for them.

Batch Processes

Pro*C will be used to develop batch processes that interact with the database.

Desktop Integration

Oracle Data Browser, Oracle DDE Manager for Windows, and Oracle Glue all provide inter-application integration in the Microsoft Windows environment and are supported by DIS. As is the case with ad-hoc query tools, there are other tools available, but they will not be supported by DIS.

1.9 Coding Standards

1.9.1 Naming Conventions

1.9.1.1 General

- Names must follow Oracle Flexible Architecture (OFA) Rules.
- Use full descriptive, pronounceable names where possible.
- Use the same name to describe the same entity or attribute across tables.
- Do not use special characters (e.g. "$" or "#") in object names.
- Separate words using an underscore.
- Similar object names must differ by at least 2 positions.

1.9.1.2 Tables

- Table names should be no more than 20 characters.
- Table names should be plural.
- Table names should be prefixed with an application alias (three characters maximum) of the application that maintains the table.

- Each table will have an associated four-character alias. This alias is to be used in SQL statements, etc. and as the prefix for each column name. Include the table alias in the comment on the table. See Appendix A for the template script for creating tables.

1.9.1.3 Columns

- Column names should be no more than 20 characters.
- Column names should be singular.
- The table alias always prefixes the column name.
- The same data element will be named the same across tables, with the exception of the table alias prefix. For example, if there is a table named SHOULDERS, and one named LANE, both with an attribute for pavement type code, the column names would be SHDR_PAVEMENT_CODE and LANE_PAVEMENT_CODE.
- Do not try to imply datatype or constraint in the column name. Do try to describe the data element. (Dates are an exception to the datatype rule. CLOSURE_DATE is a descriptive name, whereas DESCRIPTION_VARCHAR is not).
- All column names should end in a "class word" denoting the general class to which the attribute belongs. Valid class words are:

✧ AMT A numerical value, including counts and dollar values. For example, the total number of lanes column would be TOTAL_LANES_AMT, not NUM_TOTAL_LANES.

✧ CODE A key value from a look-up table, such as SURFACE_CODE.

✧ DATE A date. YEAR, MONTH, and DAY are separate class words.

✧ DAY A day value (as opposed to DATE, MONTH, or YEAR).

✧ DESC Free form descriptive text, different from NAME. An example might be CONDITION_DESC.

✧ ID An identifier, such as a generated sequence, a driver license number, etc. Notice a driver's licensce would not be DRIVER_LICENSE_NUM, as NUM is not a class word. And it is not DRIVER_LICENSE_AMT, since it is not an amount. The correct column name would be DRIVER_LICENSE_ID.

✧ IMG Images.

✧ IND An indicator, such as Yes/No or On/Off. For example, the column name for an attribute recording if a monitoring station is a fog sensor might be FOG_SENSOR_IND.

✧ MONTH For month values only, where DATE is inappropriate, e.g. PEAK_MONTH.

✧ NAME For specific names, such as LAST_NAME, and items such as NATIONAL_PARK_NAME or REST_STOP_NAME.

✧ SND Sounds.

✧ TIME A time value that is independent of any date.

✧ YEAR For year values, where DATE is not appropriate, e.g. MODEL_YEAR.

1.9.1.4 Constraints

Constraint naming rules are as follows:

CONSTRAINT	NAMING RULE
Unique	Un_tablename
Check	CKnn_tablename
Primary Key	PK_tablename
Foreign Key	FK_1st tablename (Master)

Figure 4: Abbreviated Naming Conventions

1.9.1.5 Views

The conventions for naming views are the same as for naming tables.

1.9.1.6 Indexes

- Create an unique index for a table indirectly through the UNIQUE constraint XUnn_tablename.
- Non - unique indexes for a table will always be named a non-unique - XNnn_tablename where n is a sequential number, Bitmapindex - XBnn_tablename.
- Do not attempt to include the indexed column names in the index name.

1.9.1.7 Sequences

- Sequence names should be no more than 20 characters, excluding the suffix.
- The sequence name should be suffixed with a "Seq_".
- Since a sequence can be used for more than one column, do not name the sequence for the column it populates. Instead name it for the purpose it serves.

1.9.1.8 Procedure, Packages, Functions and Triggers

- Procedure, package, function and trigger names may be up to 30 characters.
- All procedure names will be suffixed by "PR_".
- All package names will be suffixed by "PG_".
- All function names will be suffixed by "FN_".
- All trigger names will be suffixed by "TR_".

1.9.1.9 Modules

For portability purposes, all modules names (saved to the file system) will not exceed eight (8) characters. In addition all module names must:

- Begin with a letter
- Not include any special characters other than an underscore "_".
- Have a connotative name.

Extension names are generally set by the tool that creates the module or by convention:

Creation Tool	Naming Convention
SQL Scripts	.sql
PL/SQL	.sql
Libraries	.pll
Pro*C	.pc
Shell Scripts	.sh

Figure 5: Extension Naming Conventions

1.9.2 Table and Index Creation

All tables and associated indexes will be created using the template included in Appendix A.

- All tables and indexes must have storage clauses, even unique indexes created by constraints.
- All tables must be commented.
- All columns should be commented.

1.9.3 Coding Standards - SQL

Consistent SQL coding standards not only improves code maintenance, they improve the use of the Shared Pool. If a SQL statement sent to the server is already in the

Shared Pool, the sever will not parse the new statement, instead using the parse tree and execution plan already in the Shared Pool. But the two statements must be identical. Differences in case or format will cause the two statements not to match, reducing the utility of the Shared Pool.

1.9.3.1 General

In a where clause (or having clause), constants or bind variables should always be on the right hand side of the operator.

1. Never use unqualified inserts. Always specify the list of columns.
2. Avoid using functions on indexed columns, as this prevents the use of the index.
3. Do not use complex queries when the same end may be achieved by an equivalent join query.
4. Use existence tests instead of sub-queries, in, or not in whenever possible.

1.9.3.2 Format

1. All SQL statements will be in lower case. Do not use upper case for column names, table names, etc.

```
select wall_ht_ft
from wall;
```

2. Each key word should always start on a new line. The key words are:

   select

   insert into

   update

   delete

   from

   where

   order by

   group by

   for update of

   connect by

3. Subordinate key words should start on a new line and should be indented two spaces to the left of the key words listed above. The subordinate key words are:

   into

   having

   values

   set

   and

   or

4. Lines not starting with any of the above should be indented to align with the first non-key word on the previous line. If there isn't one, indent four spaces.
5. Multiple objects associated with a key word, such as a list of columns, should each be on a separate line.

6. Where possible, the operators in the where clause predicates should start in the same column.
7. The use of the four character column prefix eliminates the need to use table aliases except in correlated sub-queries where the same table is used, and correlated, in both the parent query and the sub-query. In these cases simple single-character aliases (a, b, c) should be used.
8. The select key word for sub-queries should be indented four spaces from the where key word above it. The normal formatting rules apply to sub-queries.
9. Always use a space on both sides of an operator. For example, use col1 = col2, not col1=col2

Formatting examples:

```
select abc_column1,
abc_column2,
xyz_column20,
xyz_column30
into variable1,
variable2,
variable3,
variable4
from abc a, xyz
where abc_column1  = xyz_column20
and abc_column3  = trunc(sysdate) - 2
and xyz_column30 between 1 and 25
and exists (
select 1
from def, abc b
where def_column1   = b.abc_column1
and b.abc_column2 = a.abc_column2)
order by abc_column1,
abc_column2;

insert into some_table (
column1,
column2,
column3,
```

```
column4)
values (
123,
'abc',
sysdate,
user);

update my_table
set some_column = 'Hello world'
where another_column = 'TEST';
```

1.9.4 Coding Standards - PL/SQL

1.9.4.1 Usage

PL/SQL blocks are used in database triggers, procedures, functions, Forms triggers, Report, and 3GL languages such as Pro*C. PL/SQL can also stand alone as a program unit. There should be no stand alone PL/SQL program units in production systems.

1.9.4.2 General

1. Initialize all variables when declared.
2. Use %type and %rowtype whenever possible.
3. Use cursor for loops instead of opening, fetching, and closing cursors. If a cursor is explicitly opened, it must be explicitly closed.
4. Use the explicit data type conversion functions (e.g. to_date) rather than relying on the implicit conversion abilities of SQL.
5. Avoid the use of goto. Instead, use the control structures to control flow.
6. Always include exception handlers. Use nested blocks to control the propagation of exceptions. Never use "when others then null". But do use the "when others" exception handler to create a helpful error message and avoid an unhandled exception error.
7. When performing transactions, always commit at the end of a transaction, rather than at end of job.
8. For performance, do not select from dual. Use the assignment operator instead.
9. Variable names should be prefixed with "c_" if a constant, "g_" if a global variable, and a "v_" if a regular variable, “arg_” for arguments, “x_” for user defined exception
10. Cursor names will have a "cr_" prefix.

1.9.4.3 Format

1. Follow the SQL statement formatting standards for all SQL statements within a PL/SQL block.

2. The following should always be alone on a line:

```
declare
begin
exception
end
```

3. The "declare", "begin", "exception", and "end" key words of a block should all be aligned vertically.

```
declare
  v_number      number                  := 100;
begin
  .
  .
exception
  .
  .
end;
```

4. A nested block should be indented two spaces into its parent block.

```
begin
  .
  .
begin
    .
    .
exception
    .
    .
end
  .
  .
```

5. The "if", "elsif", "else" and "end if" key words in an if statement should be vertically aligned, as should be the "loop", "while", "for" and corresponding "end loop" key words.
6. If the condition of an if or elsif statement cannot fit on one line, the continuation of the statement should be aligned with the first non-key word on the line above it, and the statements within the if or elsif block (after the then) should be indented

two spaces from that point. All other statements within elsif or else blocks for the same if statement should be vertically aligned with that block.

```
if (some_long_column_name <= trunc(sysdate)
and some_other_long_column_name = column1
and yet_another_long_column_name < v_abc) then
do_something_here;
else
do_something_else;
end if;
```

7. Exception names should be indented two spaces from the exception statement, and the statements to be executed should be indented two spaces from the exception name.

```
exception
when no_data_found then
do_something;
when too_many_rows then
do_something_else;
when others then
generate_error;
```

8. The select clause of a cursor definition will be on the next line and indented two spaces from the cursor cr_cursor_name is statement. Standard SQL statement formatting applies to this select statement.

```
cursor cr_some_name is
select column1,
column2
from my_table
order by column1;
```

1.9.5 Coding Standards - Pro*C

1.9.5.1 Usage

Pro*C will be used for batch processes, for data loading where SQL*Loader is not suitable (e.g. when conversion occurs or when specific rules must be enforced), for report generation where Oracle Reports is not suitable (e.g. when specific formatting requirements cannot be accomplished), and for other specialized tasks.

1.9.5.2 General

1. Columns of type varchar2, varchar, char, or date will be declared as varchar variables.
2. Columns of type number will be declared as int or double as appropriate.
3. Host arrays will be used whenever possible.
4. Embedded PL/SQL blocks will be used whenever possible.
5. Every Pro*C program will include the line, where nn is equal to the number of cursors open simultaneously plus 5.

   ```
   exec oracle option (maxopencursors=nn);
   ```

6. All cursors not in PL/SQL blocks must be globally declared.
7. All declared cursors should be explicitly closed during end of program processing. Do not close declared cursors prior to this point.
8. Open cursors just before fetching. A cursor may be re-opened many times, causing the bind variables to be re-evaluated each time the cursor is opened. Avoid re-opening cursors when the values of the bind variables have not changed.
9. Use the exec sql whenever sqlerror goto some_label_name to trap errors generated in embedded SQL statements.
10. Use the template Pro*C program, template.pc, as the starting point for all Pro*C programs. This program may be found in Appendix D.
11. ANSI C is the standard.

1.9.5.3 Format

1. The parts of a Pro*C program should be in the following order:

   Program comments

   #includes

   #defines

   typedefs

   Function prototypes

   Global C variable declarations

   Global SQL variable declarations

   Global cursor declarations

   main

   functions

2. Functions are formatted as follows, with the opening curly brace on its own line:

   ```
   function_name()
   {
     .
     .
   ```

```
}
```

3. Statements within a function are indented two characters from the function name.
4. Statements within a control structure are indented two characters from the control key word. As with PL/SQL if statements, if the condition in an if or while statement is too long to fit on one line, line the continuation of the condition under the beginning of the condition, and the statements within the condition are to be indented two spaces to the right of that (see the example under #5 below).
5. The opening curly brace of an if, while, or do statement should be on the same line as the condition. The closing curly brace should be vertically aligned with the if, while, or do statement and on its own line.

```
while (some_counter > 0) {
if (some_variable > 0 && some_other_variable < 0
&& yet_another_variable = 1) {
do_something(some_variable);
do_something_else();
 };
some_counter--;
};
```

6. Use blank lines liberally to provide a visual break between logical blocks of code.
7. Variables should be initialized when declared.

1.10 Use of Procedures, Triggers, Packages, Libraries

1.10.1 Internal PL/SQL Vs Libraries Vs Stored Procedure and Functions

The Oracle database and Cooperative Development Environment (CDE) tools allow the use of PL/SQL program units (function or procedure) at three different levels:

1. In the tool itself (Forms or Reports);
2. As an external library; and
3. Stored in the database.

Program units in Forms or Reports are available to that module, and may be copied or referenced by another module provided the "parent" module is stored in the database. In this case, the PL/SQL program unit is copied or referenced at generation time. If it is changed in the parent module, all other modules must be re-generated for the change to take effect.

External PL/SQL libraries are separately generated collections of functions and procedures that are bound to the calling module at runtime. Changes to the library take affect the next time the calling module is executed. Re-generation of the calling

module is not required. External libraries are stored in the filesystem and may be called by Forms or Reports.

Stored procedures and functions are saved in the database and are available to any module that can run PL/SQL, such as Forms, Report, or Pro*C.

Guidelines for usage are:

- Store a PL/SQL program unit in a Form, Reports, or Menu when it is specific to that module and no other module will require that functionality.
- In general, PL/SQL routines that can be made into a procedure or function and that may be called by more than one module should be stored in the database.
- External libraries should be reserved for cases where special factors either prevent or make it unreasonable to store the procedure or function in the database. An example is reading image files into a form where the image files are stored on the client. In this case, no database access is required to read the image, so using an external library keeps all the processing on the client. Standard error handling or common operations such as setting visual attributes are other examples.

1.10.1.1 Packages

Stored procedures and functions may be grouped into a package. An entire package is read into memory at once, potentially improving performance where there is a high degree of interdependence among the routines in the package. However, use of large packages can actually decrease performance as memory is swapped in order to load all of a package into memory.

Packages should be kept small. Only group those routines with a high degree of interdependence into a package. Do not use packages as a method of code control. In other words, do not group routines into a package because they all relate to the same functional area and you want to be able to see them all at the same time.

Packages may also be cached using the dbms.package.keep package. This package will load a package into the cache so it is always in memory. While not for all packages, this approach can be used to improve performance in critical areas.

1.10.1.2 Database Triggers

As a guideline, do not become trigger-happy. Database triggers are extremely useful tools to ensure that a certain action occurs each and every time upon insert, update, or delete. When the requirement states that the action must occur upon insert, update, or delete regardless of who does it, or how the data change happens, use a trigger. Otherwise, look at some of the other methods (stored procedures, external libraries, etc.) in addition to triggers.

If a large amount of logic is required in a trigger, consider building a procedure to perform that logic and call it from the trigger. The procedure is stored in complied form in the database, while the trigger is not.

1.10.2 Code Documentation Standards

All program units, PL/SQL routines, SQL scripts, C programs, etc. will be commented.

In general, comments should explain the why of the processing. For example, the following comment provides no value:

```
/* set variable now to the current date and time */
now := sysdate;
```

Instead, explain the reason for doing so:

```
/* save current date and time to compare to date and time
•       at end of run
    */
now := sysdate;
```

PL/SQL, SQL, and C routines will conform to the following standards:

C - style slash-star comment delimiters will be used: /* */

The top of each unit will have a comment block:

```
/* ========================================
* Name     :
* Created On:
* Created By:
* SCN No   :
* Purpose  :
 *
 * ========================================
* Modification History:
* Version  CMA#  By       Date     Comments
 *
 */
```

Use blanks lines to separate logical pieces of code, function definitions, etc.

Comment variable declarations.

In the comment area of the Module Property Sheet, include the following:

Name: (module name)

Created On:

Created By:

SCN No:

Purpose:

Modification History:

Version:

SCN No:

By:

Date:

Comment:

Comment the purpose of all internal PL/SQL procedures and functions. Comment the code as documented in the PL/SQL section above.

All code units should have a comment line at the end of the code containing the name of the procedure.

1.11 Code Quality Assurance and Testing

1.11.1.1 In General

Code Quality Assurance has several goals:

1. Catch errors at an early stage.
2. Improve efficiency.
3. Enforce standards.
4. Continue the programmer's professional growth.

Code QA methods usually involve structured walkthroughs, peer review, or review by a senior programmer. Well-managed structured walkthroughs are often the most effective in meeting the code QA goals. Walkthroughs can be effective for particularly critical, complex, or illustrative code and should be used at AXA Client Solutions when appropriate. However, structured walkthroughs can become quite time-consuming. For this reason, they are not a practical choice for on-going code QA.

Peer reviews are only as effective as the person performing the review. They are open to collusion ("don't criticize me and I won't criticize you") and are usually not very effective in meeting the code QA goals.

Given these constraints, all program units will be reviewed by a designated senior programmer after unit testing by the programmer and before migration to the test environment.

1.11.1.2 Code Review

When the developer has completed unit test, he or she will notify the appropriate senior programmer or DBA that the code is ready for code QA. This includes ensuring that the code is available to the senior programmer or DBA by setting permissions, granting access, etc. The developer must also ensure that the senior programmer or DBA has a copy of the specifications.

At the senior programmer's or DBA's discretion, he or she will review the code either with or without the developer present. The senior programmer or DBA will review the code with an eye to:

- meeting requirements-does it do what it is supposed to do?
- is there a better way to do it (e.g. have common routines been utilized fully)?
- does it meet standards?

The code review continues until the senior programmer believes the code is ready for production. He/she then approves the code for migration to the test environment.

1.11.1.3 System Test

When the code is migrated to the test environment, the developer notifies QA. QA is then responsible for reviewing the specifications and developing and executing a test plan based on the analyst's testing guidelines.

All modules should be tested for the following:

- Correctness and consistency of results
- Incomplete or unhelpful error messages
- System performance
- Security violations

Modules that alter data require a more thorough test. These modifications must be tested to ensure:

- No data integrity rules are violated, either by user action or by processing.
- All updates/inserts/deletes are consistent with the definitions of the entities and attributes.
- All processing dependencies are understood and tested.

System test continues until QA is convinced that the code is ready for production. During this time, QA works with the developer and the senior programmer who reviewed the code (if necessary) to fix bugs or resolve other issues. When QA deems the code is ready, QA notifies the developer. The developer then notifies the user sponsor that the code is ready for user test. The user sponsor should be given a reasonable amount of time to test the change, based on the scope of the change. However, a deadline should be set for when user testing should be complete.

Once the user sponsor has tested the change, the developer may request that it be moved to the training environment.

Glossary

This glossary describes AXA Financials business terms and related technical terms.

Index