Institutional Research and Academic Planning (IRAP)

Coding Standards Document

Version 0.2

*THIS PAGE HAS BEEN LEFT INTENTIONALLY BLANK *

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

1.1 PURPOSE

The goal of this document is to create uniform coding habits amongst Institutional Research and Academic Planning (IRAP) analysts in the department so that reading, checking, maintaining and reusing code written by different persons becomes easier. The intent of these standards is to define a natural style and consistency, yet leave code authors the freedom to practice their craft without unnecessary burden. When analysts adhere to common coding standards, the following can occur:

- Programmers can go into any code and figure out what's going on, so maintainability, readability, and reusability are increased.
- Code walk through become less painful.
- New people can get up to speed quickly.
- People new to a language are spared the need to develop a personal style and defend it to death.
- People new to a language are spared making the same mistakes over and over again, so reliability is increased.
- People make fewer mistakes in consistent environments.
- Idiosyncratic styles and college-learned behaviors are replaced with an emphasis on business concerns - high productivity, maintainability, shared authorship, etc.

Most arguments against a particular standard come from the ego. So, in the interests of establishing the IRAP department as a showcase for learning and knowledge sharing, be flexible, control the ego a bit, and remember all that we do is to benefit the team as a whole.

Note: Queries used in this document are examples only. They are not functioning queries!

1.2 SCOPE

This document describes general coding standards and many of the guidelines can be applied directly to multiple programming languages in use within the department. Queries used are for documentation purposes only. They may not be functioning programs that can be used. Refer to the IRAP Code Library within Atlassian JIRA for the inventory of code that have been certified and published for reuse.

If you do not have access to JIRA, please contact any of the following IRAP resources:

- Ola Popoola – ola.popoola@ucop.edu
- Sanketh Sangam – sanketh.sangam@ucop.edu
- Poorani Rajamanickam – poorani.rajamanickam@ucop.edu

2. THE IRAP CODE LIBRARY (ICL)

1.1 ICL STORAGE REQUIREMENTS

All code must be uploaded into Atlassian JIRA and placed within the IRAP Code Library (ICL) project. The following fields are required for any published submission into the IRAP Code Library:

- **Summary** – provide a brief description of the code that is being submitted into the library.
- **Subject Area** – Identify the subject area(s) your code covers.
- **Description** – Provide a description of the code to be submitted – What does it do? What use will it serve? Every time code is updated, notes will be added to this field.
- **Code Environment** – Identify whether the code to be submitted can be used for star schema(s) or data mart(s) or both.
- **Code Type** – Identify whether the code to be submitted is SAS, SQL, PL/SQL, R, Python, SAS EGP, Tableau or Other.
- **Code Comments Complete** – Indicate whether all comments that will enhance code readability has been included before submission by author.
- **Code Functionality Complete** – Indicate whether code to be submitted has been fully tested and certified to be functional by author.
- **Code Naming Convention Complete** – Indicate whether code to be submitted follows the standard naming convention prescribed as per coding standards.
- **Code Complexity Level** – Indicate whether code to be submitted is not complex, has minimal complexity, is semi-complex or complex.
- **Attachment** – Upload code being submitted with appropriate file extension.
- **Publication Date** – provide the date that the code is submitted.
- **Reporter** – Auto-populated based on user credentials.
- **Assigned** – Assign to IRAP resource is asset is a placeholder and code will be supplied later.
- **Code Update Date/Timestamp** – A record the date and time of most recent update to code.

There may be cases where a placeholder is created for an asset that will be added later. In cases like this, only the following fields are initially required:

- Summary
- Subject Area
- Description
- Reporter
- Assignee

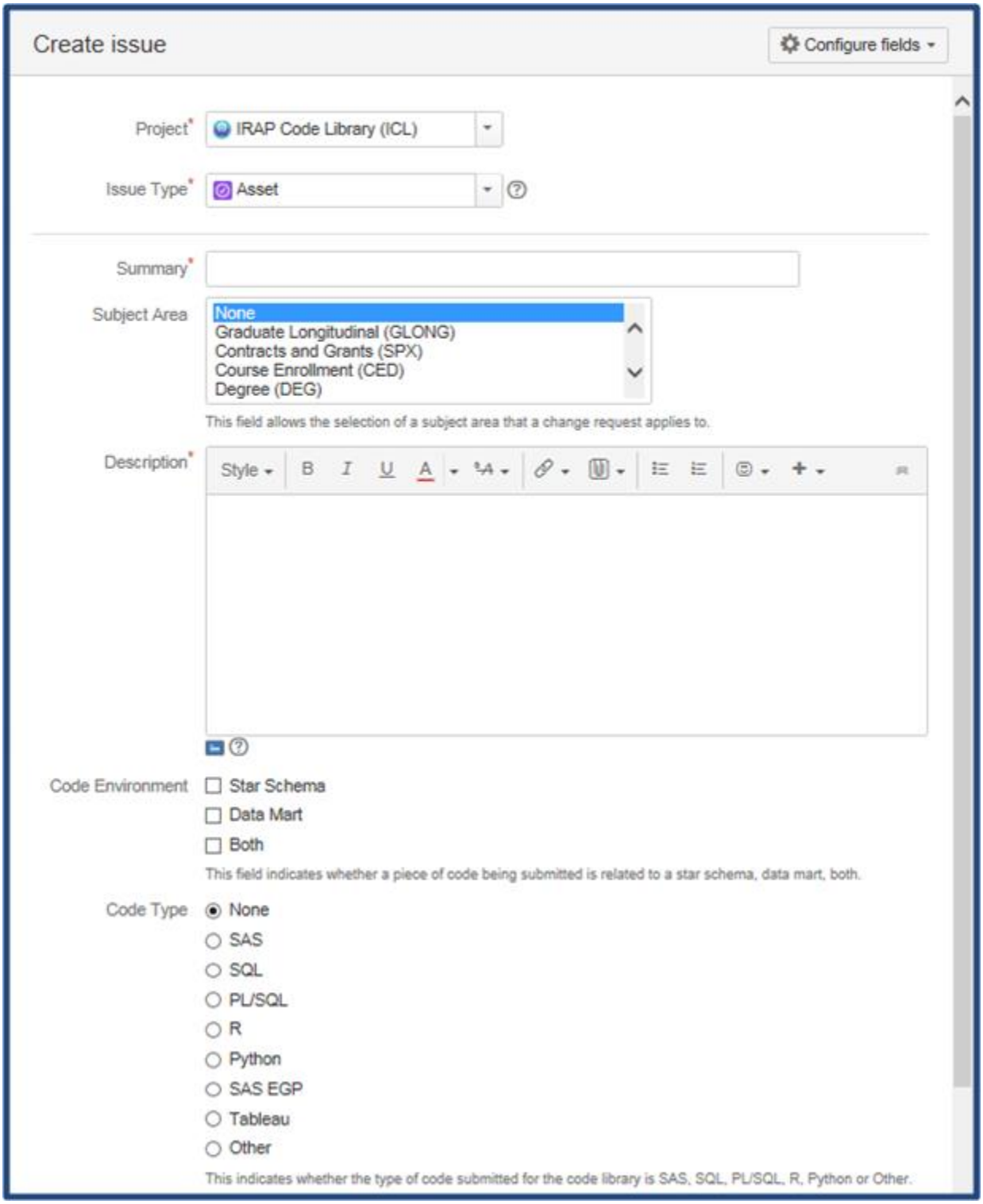

Figure 1: Code Library Submission Form - A

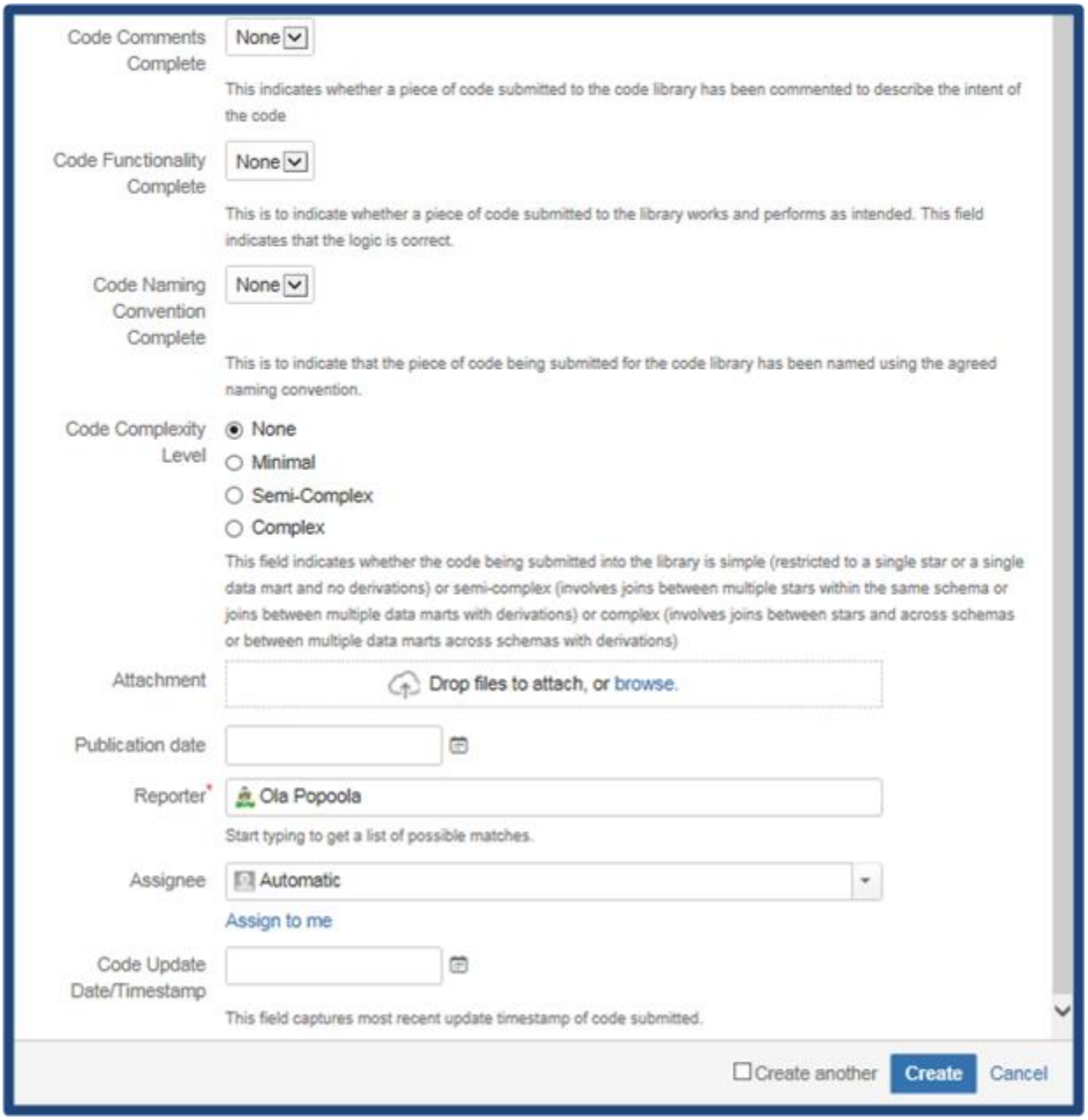

Figure 2: Code Library Submission Form – B

1.2 IRAP CODE LIBRARY WORKFLOW PROCESS

There is a workflow process designed in Atlassian JIRA to help move a submitted code asset to published status. The workflow consists of the following steps:

- To Do – a code asset will be in the 'to-do' status if a placeholder has simply been created for an asset that will be added later or
- Draft – a code asset is still in progress or in DRAFT status.
- In Review – a code asset has been deemed complete by the author and ready for peer review.
- Approved – a code asset has been approved by the review team. Selection of peers to participate in review process is entirely up to the author. The expectation is that reviewers will be team members who can act as additional eyes on the code asset for accuracy purposes.
- Published – a code asset has been certified and published for use by the IRAP team.

Figure 3: Code Library Submission Workflow

2 DOCUMENTATION AND CODING STANDARDS

2.1 INDENTATION

Proper and consistent indentation is important in producing easy to read and maintainable programs. Indentation should be used to:

1. Emphasize the body of a control statement such as a loop or a SELECT statement

Example

```
SELECT          COUNT (STUD_ID),
                STUD_LOC_CMP_CD,
                STUD_FST_NAM,
                STUD_LST_NAM,
                STUD_DT_OF_BTH,
                STUD_GNDR_DESC
FROM            STUD_BI.STUDENT_D
WHERE           STUD_LOC_CMP_CD = '03'
AND             STUD_DMSTC_FGN_CZ_STAT_DESC = 'FOREIGN'
AND             STUD_CUR_ACTV_FL = 'Y'
GROUP BY        STUD_LOC_CMP_CD,
                STUD_FST_NAM,
                STUD_LST_NAM,
                STUD_DT_OF_BTH,
                STUD_GNDR_DESC
```

2. Emphasize the body of a conditional statement. The THEN keyword will be placed on the line below the IF keyword but aligned with it. The ELSE IF keyword will also be aligned with the IF.

Example

```
IF    (AB 540 Supp Tuition Exemption > 0 OR
      Veterans Supp Tuition Exemption > 0)
THEN set AB540 Flag to 'Y'
ELSE  set AB540 flag to 'N'
END;
```

3. Emphasize a new scope block.

Example

```
DECLARE
      v_father_name VARCHAR2(20):='Patrick';
      v_date_of_birth DATE:='20-Apr-1972';
BEGIN
      DECLARE
            v_child_name VARCHAR2(20):='Mike';
```

```
      BEGIN
            DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
            DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
            DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
      END;
      DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
END;
```

2.2 HEADER DOCUMENTATION

A header should appear at the start of any piece of code submitted into the code library. The change history is a critical piece as it allows the reader to provide documentation around changes to the code.

Example

```
-- ****************************************************************
-- Description: Describe the purpose of your code
-- Author:      <your name>
-- Revision History
-- Date            Author       Reason for Change
-- ----------------------------------------------------------------
-- 03 JAN 2015     J. Davis     Created.
-- ****************************************************************
```

2.3 INLINE COMMENTS

Inline comments explaining the function of the code submitted or key aspects of the algorithm should be used frequently. It helps to promote code readability. This will allow a person not familiar with the code to more quickly understand it. It also helps the programmer who wrote the code to remember details that can very easily be forgotten over time. Having good comments also reduces the amount of time required to perform updates to the code due to policy changes.

Inline comments appear in the body of the code itself. When properly implemented, they explain the logic or parts of the algorithm that are not readily apparent from the code itself. Inline comments are also useful in explaining the tasks being performed by a block of code. A good rule of thumb is that inline comments should make up 20% of the total lines of code in a program, excluding the header documentation blocks.

2.3.1 COMMENTS IN SQL OR PL/SQL

Comments in SQL and PL/SQL are ignored by the program complier. Although the primary purpose is to document code, you can also use it to disable obsolete pieces of code. The following two options exist:

1. -- Use two dashes for single line comments

2. /* - To start any comment that spans multiple lines and */ - to end any comment that spans multiple lines

Example – Single Line Comment

```
SQL> DECLARE
  2   HOW_MANY        NUMBER;
  3   NUM_TABLES      NUMBER;
  4   BEGIN
  5   -- Begin processing
  6   SELECT COUNT(*) INTO HOW_MANY
  7   FROM USER_OBJECTS
  8   WHERE OBJECT_TYPE = 'TABLE'; -- Check number of tables
  9    NUM_TABLES := HOW_MANY; -- Compute some other value
 10  END;
 11  /

PL/SQL procedure successfully completed.
SQL>
```

Example – Multi Line Comment

```
SQL> DECLARE
  2    SOME_CONDITION         BOOLEAN;
  3    PI                     NUMBER := 3.1415926;
  4    RADIUS                 NUMBER := 15;
  5    AREA                   NUMBER;
  6  BEGIN
  7   /* Perform some simple tests and assignments */
  8   IF 2 + 2 = 4 THEN
  9   SOME_CONDITION := TRUE;
 10  /* we expect this THEN to always be performed */
 11   END IF;
 12   /* the following line computes the area of a circle using pi,
 13   which is the ratio between the circumference and diameter.
 14   After the area is computed, the result is displayed. */
 15   AREA := pi * radius**2;
 16   DBMS_OUTPUT.PUT_LINE('The AREA is: ' || TO_CHAR(area));
 17  END;
 18  /
The AREA is: 706.858335

PL/SQL procedure successfully completed.

SQL>
```

Note: The multi-line comment style can also be used for single-line comments but the single line comment style cannot be used for comments that span multiple lines.

2.3.2 COMMENTS IN SAS

You can use the comment statement anywhere in a SAS program to document the purpose of the program, explain unusual segments of the program, or describe steps in a complex program or calculation. SAS ignores text in comment statements during processing. The following two options exist:

1. *message; - This specifies the text that explains or documents the statement or program.
    a. Range: These comments can be any length and are terminated with a semicolon.
    b. Restrictions
        i. These comments must be written as separate statements.
        ii. These comments cannot contain internal semicolons or unmatched quotation marks.
        iii. A macro statement or macro variable reference that is contained inside this form of comment is processed by the SAS macro facility. This form of comment cannot be used to hide text from the SAS macro facility.
    c. Tip: When using comments within a macro definition or to hide text from the SAS macro facility, use this style comment:

Example - *message;

```
*This code finds the number in the BY group;
```

2. /* message */
    a. Range: These comments can be any length.
    b. Restriction: This type of comment cannot be nested.
    c. Tips
        i. These comments can contain semicolons and unmatched quotation marks.
        ii. You can write these comments within statements or anywhere a single blank can appear in your SAS code.
        iii. In the Microsoft Windows operating environment, if you use the Enhanced Editor, you can comment out a block of code by highlighting the block and then pressing CTRL-/ (forward slash). To uncomment a block of code, highlight the block and press CTRL-SHIFT-/ (forward slash).

Example - /* message */

```
input @1 name $20.  /* last name     */
    @200 test 8.  /* score test    */
    @50 age 3.;  /* customer age */
```

2.4 SPACING

The proper use of spaces within a line of code could greatly enhance readability. Basic rules of thumb are as follows:

1. A keyword followed by a parenthesis should be separated by a space.

Example

```
SELECT          COUNT (DISTINCT STUD_ID),
                STUD_GNDR_CD,
                STUD_3CAT_HM_LOC_NAM,
                STUD_DMSTC_FGN_CZ_STAT_DESC
FROM            STUD_BI.STUDENT_D
WHERE           STUD_CUR_ACTV_FL = 'Y'
AND             STUD_END_EFF_DT = '12-31-9999'
GROUP BY        STUD_GNDR_CD,
                STUD_3CAT_HM_LOC_NAM,
                STUD_DMSTC_FGN_CZ_STAT_DESC
FETCH FIRST 10 ROWS ONLY;
```

2. A blank space should appear after each comma in an argument list.

```
SELECT          DISTINCT STUD_ID, STUD_GNDR_DESC
FROM            STUD_BI.STUDENT_D
WHERE           STUD_CUR_ACTV_FL = 'Y'
AND             STUD_END_EFF_DT = '12-31-9999'
AND             STUD_GNDR_CD = 'F'
AND             STUD_DMSTC_FGN_CZ_STAT_DESC = 'Foreign'
FETCH FIRST 10 ROWS ONLY;
```

3. All binary operators except the period (.) should be separated from their operands by spaces.

Example BEFORE

```
Total_Cost=Price+Price*Sales_Tax;
```

Example AFTER

```
Total_Cost = Price + Price * Sales_Tax;
```

Example BEFORE

```
IF source<=10 then source1='2';
ELSE IF source=11 then source1='3';
ELSE IF source<=13 then source1='4';
ELSE source1='5';
END;
```

Example AFTER

```
IF source <= 10 then source1 = '2';
ELSE IF source =11 then source1 = '3';
ELSE IF source <=13 then source1 = '4';
ELSE source1 ='5';
END;
```

2.5 WRAPPING LINES

When an expression will not fit on a single line, break it according to these guiding principles:

1. Break after a comma

Example – BEFORE

```
SELECT          COUNT (DISTINCT STUD_ID), CAMPUS_ACRONYM, ACADEMIC_YR,
                TERM_NAME, ENR_UC_ETHN_6_CAT, COUNTY_OF_RES
FROM            IRAP_BI.ENROLLMENT_DM
GROUP BY        CAMPUS_ACRONYM, ACADEMIC_YR, TERM_NAME,
                ENR_UC_ETHN_6_CAT, COUNTY_OF_RES;
```

Example – AFTER

```
SELECT          COUNT (DISTINCT STUD_ID),
                CAMPUS_ACRONYM,
                ACADEMIC_YR,
                TERM_NAME,
                ENR_UC_ETHN_6_CAT,
                COUNTY_OF_RES
FROM            IRAP_BI.ENROLLMENT_DM
GROUP BY        CAMPUS_ACRONYM,
                ACADEMIC_YR,
                TERM_NAME,
                ENR_UC_ETHN_6_CAT,
                COUNTY_OF_RES;
```

2. Break after an operator

Example

```
SELECT          DISTINCT STUD_ID,
                UNITS_CRED_EXAM_AP + UNITS_CRED_EXAM_IB +
                UNITS_CRED_EXAM_UC + UNITS_CRED_CCC +
                UNITS_CRED_OTHER_NONUC AS EXTERNAL_UNITS
FROM            IRAP_BI.ENROLLMENT_DM
```

3. Prefer high-level breaks to lower-level breaks

Example – BEFORE

```
TOTAL_ITEM = ITEM_1 * (ITEM_2 + ITEM_3 + ITEM_4 – ITEM_5) + 2 * (
                            ITEM_6 + ITEM_7 – ITEM_8) + 4 * ITEM_9;
```

Example – AFTER

```
TOTAL_ITEM = ITEM_1 * (ITEM_2 + ITEM_3 + ITEM_4 – ITEM_5)
            + 2 * (ITEM_6 + ITEM_7 – ITEM_8) + 4 * ITEM_9;
```

Align the new line with the beginning of the expression at the same level on the previous line.

Example - BEFORE

```
TOTAL_ENROLLED_STUDENTS = COUNT_ENROLLED_UCB + COUNT_ENROLLED_UCSF + COUNT
ENROLLED_UCD + COUNT_ENROLLED_UCLA + COUNT_ENROLLED_UCR +
COUNT_ENROLLED_UCSD + COUNT_ENROLLED_UCSC + COUNT_ENROLLED_UCSB +
COUNT_ENROLLED_UCI + COUNT_ENROLLED_UCM
```

Example – AFTER

```
TOTAL_ENROLLED_STUDENTS =    COUNT_ENROLLED_UCB + COUNT_ENROLLED_UCSF +
                             COUNT ENROLLED_UCD + COUNT_ENROLLED_UCLA +
                             COUNT_ENROLLED_UCR + COUNT_ENROLLED_UCSD +
                             COUNT_ENROLLED_UCSC + COUNT_ENROLLED_UCSB +
                             COUNT_ENROLLED_UCI + COUNT_ENROLLED_UCM
```

2.6 VARIABLE DECLARATIONS

Variable declarations that span multiple lines should always be preceded by a type. It is best to have one variable per line for readability purposes.

Example BEFORE

```
DECLARE     FIRST_NAME, MIDDLE_NAME, LAST_NAME, ADDR_LN_1, ADDR_LN_2,
            CITY, STATE       VARCHAR(25);
DECLARE     LAST_NAME   VARCHAR(25);
DECLARE     BIRTH_DT, ENROL_DT, GRADUATION_DT DATE;
```

Example AFTER - BETTER

```
DECLARE     FIRST_NAME, MIDDLE_NAME, LAST_NAME       VARCHAR(25);
DECLARE     ADDR_LN_1, ADDR_LN_2, CITY, STATE        VARCHAR(25);
DECLARE     BIRTH_DT, ENROL_DT, GRADUATION_DT        DATE;
```

Example – AFTER - BEST

```
DECLARE     FIRST_NAME          VARCHAR(25);
DECLARE     MIDDLE_NAME         VARCHAR(25);
DECLARE     LAST_NAME           VARCHAR(25);
DECLARE     ADDR_LN_1           VARCHAR(25);
DECLARE     ADDR_LN_2           VARCHAR(25);
DECLARE     CITY                VARCHAR(25)
DECLARE     STATE               VARCHAR(25);
DECLARE     BIRTH_DT            DATE;
DECLARE     ENROL_DT            DATE;
DECLARE     GRADUATION_DT       DATE;
```

2.7 SQL QUERIES AND CASE SELECTIONS

- For schema names, use uppercase letters.
- For table names, use uppercase letters.
- For column names, use uppercase letters.
- For table aliases, use lower or uppercase – no mixed case. Append alias with either _D (dimension table) or _F (fact table) or _M (for MQTs) or _V (for views)
- Comments can be mixed case – some flexibility allowed here.

2.8 PROGRAM STATEMENTS

Program statements should be limited to one per line. Use spaces so expressions can read like sentences.

Example BEFORE

```
IF (fye>0) then do;
    pcnts1=(fs1/fye)*100; pcnts2=(fs2/fye)*100; pcnts3=(fs3/fye)*100;
END;
```

Example AFTER

```
IF (fye > 0) then do;
      pcnts1 = (fs1/fye) * 100;
      pcnts2 = (fs2/fye) * 100;
      pcnts3 = (fs3/fye) * 100;
END;
```

2.9 USE OF PARENTHESIS

It is better to use parenthesis liberally even in cases where operator precedence unambiguously dictates the order of evaluation of an expression.

Example BEFORE

```
pcnts11 =  fs11/fye * 100;
```

Example AFTER

```
pcnts11 =  (fs11/fye) * 100;
```

Example BEFORE

```
WHERE color = 'red' AND size in ('1', '2')
```

Example AFTER

```
WHERE (color = 'red' AND (size in '1', '2'))
```

2.10 SQL FORMATTING

Be particularly mindful of the following when writing SQL code:

Use uppercase for all SQL Keywords such as SELECT, INSERT, UPDATE, FROM, WHERE, GROUP BY and ORDER BY

1. Use single quote characters to delimit strings.
2. Use blank lines to separate code sections.
3. Format JOIN operations using indents.
4. Use ANSI Joins instead of old style joins.

Example BEFORE

```
SELECT      EMP.EMPLOYEE_NAME, DEPT.DEPARTMENT_NAME
FROM        EMPLOYEE EMP, DEPARTMENT DEPT
WHERE       EMP.DEPT_ID = DEPT.DEPT_ID; --old style join
```

Example AFTER

```
SELECT      EMPLOYEE_NAME, DEPARTMENT_NAME
FROM        EMPLOYEE
INNER JOIN  DEPARTMENT ON EMPLOYEE.DEPT_ID = DEPARTMENT.DEPT_ID; -- ANSI
join
```

5. Avoid using SELECT *! Name your columns explicitly and only pull the columns you need.
6. Avoid using <> as a comparison operator. Use IN instead.

Example BEFORE

```
WHERE CAMPUS_CODE <> '02'
```

Example AFTER

```
WHERE CAMPUS_CODE IN ('01', '03', '04', '05', '06', '07', '08', '09', '10)
```

7. Do not use column numbers in the ORDER BY clause

Example BEFORE

```
SELECT      FST_NAME,
            LST_NAME,
            GENDER,
            DOB,
            TITLE
FROM        HR.DEPARTMENT
ORDER BY    2, 1;
```

Example AFTER

```
SELECT      FST_NAME,
            LST_NAME,
            GENDER,
            DOB,
            TITLE
FROM        HR.DEPARTMENT
ORDER BY    LST_NAME,
            FST_NAME;
```

2.11 NAMING CONVENTIONS

Follow the standards below for naming convention:

1. File name should include some context around the function of the code.
2. File name should include initials of the author.
3. File name should include the code type (could be the file extension or embedded in the name if code is in a text file with a .txt extension.

Example:

SQL code to calculate cumulative debt from Financial Aid data may be named as follows:

`CALC_CUMM_DEBT_CF.SQL` where

- CALC_CUMM_DEBT – Calculating Cumulative Debt
- CF – Chris Furgiuele
- SQL – Code type of Structured Query language

2.12 UNIT TESTING

Sufficiently test your code before deployment to the code library. You can verify that your numbers are correct by viewing already published data. If your code is extensive, test it in sections and merge sections for additional testing.

2.13 BENEFITS OF CODING STANDARDS

- Provides everyone on the team a set of rules and guidelines for formatting source code.
- Allows for easier code integration.
- Improves team member onboarding and integration - Shallower learning curve and useful training for new hires.
- Better teamwork. Don't waste time with needless debates. Spend your creative energy on things that matter.
- Allows for easier long term code maintenance.
- Minimizes unnecessary communication.
- Improved development speed - saves resources due to less man hours.
- Better code readability.
- Improved code quality.
- Fewer bugs. Good standards should minimize common coding mistakes and improve accuracy.

It is hoped that the coding standards becomes something analysts learn from, want to follow, and want to contribute to. It is intended to make lives easier, not harder. The coding standards will be a living document; with a mechanism for evolving it and for dealing different code submissions within the code library and the inevitable changes that will be applied to them.

3. CODE REVIEW PLAN

3.1 BASIC CODE REVIEW CHECKLIST

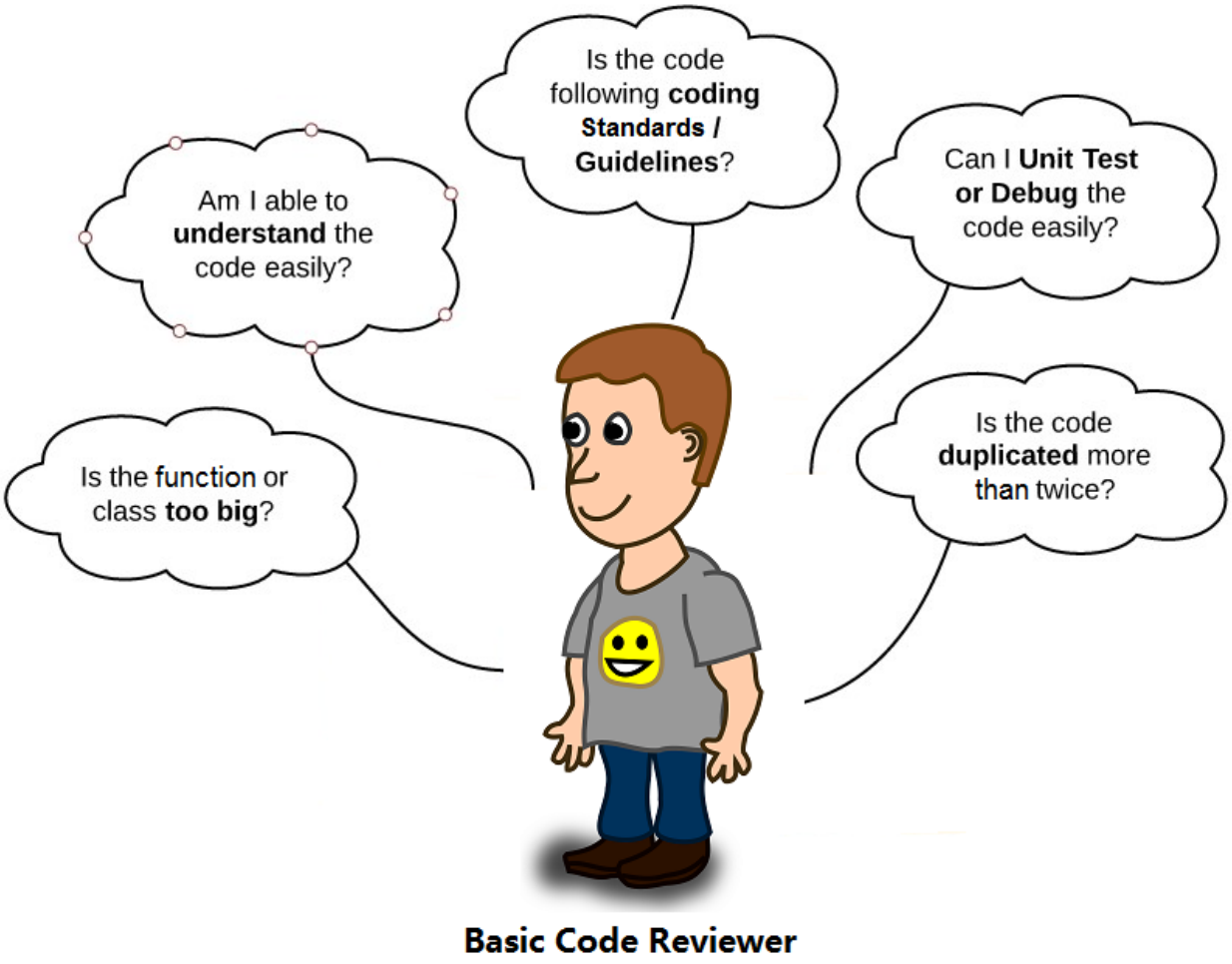

Figure 4: Basic Code Review Checklist

3.2 PEER CODE REVIEW

In a healthy culture, team members engage their peers to improve the quality of their work and increase their productivity. They understand that the time they spend looking at a colleague's work product is repaid when other team members examine their own deliverables.

The best analysts seek out reviewers. Peer review is a great way of ensuring that coding standards are met and accuracy is ensured. Grab a teammate to walkthrough your code with you. Two pairs of eyes are always better than a single pair. Encourage your peer reviewer to ask questions if needed.