Coding Standards for selenium Automation Project.

1. Introduction

This document describes a collection of standards, conventions and guidelines for designing and developing framework/scripts for java code in selenium automation. This document will help to ensure consistency across the code, resulting in increased usability and maintainability of the developed code.

1. Why coding standards are so important?

Coding standards for Java are important because they lead to greater consistency within the code of all developers. Consistency leads to the code that is easier to understand, which in turn results in a code, which is easier to develop and maintain. Code that is difficult to understand and maintain runs the risk of being scrapped and rewritten.

2. Overview

This document provides guidelines for:

- Naming standards
- Comment standards
- Functions/Methods standards
- Indentations
- Automation scripts
- General guidelines

2.Standards for Variables

3. Naming Variables

1. Variable names should be defined with data type abbreviation followed by actual variable name (English descriptors that accurately describe the variable/field/class/interface)

   Example:

- int intCount

- String strMethodName
- float ftvariableName
- decimal decVariableName
- object objObjectName

2. Domain specific terminologies should be used.
3. If the users of the system refer to their clients as Customer, then the term Customer for the class should be used, not client.
4. Mixed case should be used to make names readable.
5. Abbreviations should be used sparingly, but if it is used then it should be used intelligently and should be documented

   For example, to use a short form for the word "number", choose one of nbr, no or num.
6. Long names (<15 characters is a good tradeoff) should be avoided.
7. Names that are similar or differ only in case should be avoided.

4. Naming Exception Objects

The letter 'e' should be used for a generic exception object name.

5. Declaring and Documenting Local variables

1. One local variable per line of code should be used.
2. Local variable should be declared with an end line comment.
3. Declare all local variables before the functional block or in the beginning of the script.
4. Whenever a local variable is used for more than one reason, it effectively decreases its cohesion, making it difficult to understand. It also increases the chances of introducing bugs into the code from unexpected side effects of previous values of a local variable from earlier in the code.

   Note: Reusing local variables is more efficient because less memory needs to be allocated, but reusing local variables decreases the maintainability of code and makes it more fragile.

6. Standards for Parameters (Arguments) to Member Functions

Function parameters should be named following the exact same conventions as for local variable.

Example:

If Account has an attribute called balance and you needed to pass a parameter representing a new value for it, the parameter would be called accountBalance.

7. Naming Classes

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words -avoid acronyms and abbreviations.

Example: class ReadExcelFile { . . }

8. Naming Interfaces

Interface names should be capitalized like class names.

Example: interface RegesterDelegate;
interface Storing;

9. Naming Methods

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

Example:
ResultSet getData(String query)
void initiateExcelConnection(String fileName)

10. Naming Constants

The names of the variables constants should be all uppercase with words separated by underscores ("_").

Example:
int MIN_WIDTH = 4;
int MAX_WIDTH = 5;

3.Comment Standards

1. Comments should be added to increase the clarity of code.
2. Document something why it is done not just what it is done.
3. Every change to the framework/scripts should be documented in modification history. A modification history should contain the following:

• Name of the person who changed the code:
• Date of change:
• Version:
• Changed function/event:
• Change description:

Note: Successful build should be ensured after checking in the scripts/changes into repository.

11. Implementation of Comment Formats

Programs can have four styles of implementation of comments: block, single-line, trailing and end-of-line.

3.1.Block Comments

- Block comments are used to provide descriptions of files, methods, data structures and algorithms.
- Block comments should be used at the beginning of each file and before each method. They can also be used in other places, such as within methods.
- Block comments inside a function or method should be indented to the same level as the code they describe.
- A block comment should be preceded by a blank line to set it apart from the rest of the code.
- Block comments have an asterisk "*" at the beginning of each line except the first.
  Example:

```
/*
* Here is the block comment.
*/
```

3.2.Single-Line Comments

- Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format.
- A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java:

```
if (condition)
{

/* Handle the condition. */
...
}
```

3.3.Trailing Comments

- Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.
- Avoid the assembly language style of commenting every line of executable code with a trailing comment. Here's an example of a trailing comment in Java code (also see "Documentation Comments"

```
if (a == 2)
{
return TRUE; /* special case */
}
else
{
        return isprime(a); /* works only for odd a */
}
```

3.4.End-Of-Line Comments

The // comment delimiter begins a comment that continues to the newline. It can comment out a complete line or only a partial line.

Example:

```
//  Page Factory Initialization
ProductsLib productsTestDataObject = new ProductsLib())
```

12.Framework Code Header Comments

The framework code header should contain the following:

- The copyright and proprietary information
- Name of the framework code
- Author of the code
- Name of the reviewer
- Date of creation
- Version number

Example:

```
/**
* Project Name : Your Company Automation Framework
* Author : Your Company QA
* Version : V1.0
* Reviewed By : Manager 1
* Date of Creation : April 13, 2013
* Modification History :
* Date of change : 13-Sep-09
* Version : V1.1
* changed function : def func1
* change description :
* Modified By : Tester 1
*/
```

4.Functions/Methods

13.Function Name

Member functions should be named using a full English description, using mixed case with the first letter of any non-initial word capitalized. The first word of the member function should be a verb.

Examples:

openAccount()
printMailingList()
save()
delete()

14.Naming Accessor Member Functions

4.1.Getters:

Getters refer to member functions that return the value of a field/attribute/property of an object.

- Use prefix "get" to the name of the field/attribute/property if the field in not Boolean.
- Use prefix "is" to the name of the field / attribute / property if the field is Boolean.
- A viable alternative is to use the prefix 'has' or 'can' instead of 'is' for boolean getters.

Examples:

getFirstName()
isPersistent()

4.2.Setters:

Setters refer to member functions that modify the values of a field. Use prefix 'set' to the name of the field.

Examples:

setFirstName()

4.3.Constructors:

In Java, constructor is a member function that performs any necessary initialization when an object is created. Constructors are always given the same name as their class name.

Examples:
Customer()
SavingsAccount()

15. Function Header

Function header should contain following detail as mentioned in the below example:

```
/**
*****************************************************************************
* @Project Name : TACOE - Selenium Framework.
* @Function Name : getValuesFromExcel()
* @Description : This function is used to fetch data from excel sheet.
* @param : fileName - Name of the workbook from which the test data
* needs to be fetched.
* @param : sheetName - Name of the workbook from which the test data
* needs to be fetched.
* @Return : Resultset
* @Date : June 2018
* @Author :  Tester Name
*****************************************************************************
*/
```

Note: It's not necessary to document all the factors described above for each and every member function because not all factors are applicable to every member function.

16. Internal Documentation:

Comments within the member functions:

- Use C style comments to document out lines of unneeded code.
- Use single-line comments for business logic.
- Following should be documented inside the definition:
- Control Structures:
- comparison statements and loops
- Why, as well as what, the code does
- Local variables
- Difficult or complex code
- The processing order If there are statements in the code that must be executed in a defined order.
- Document the closing braces If there are many control structures one inside another.

17. Function Complexity

Framework code should be designed and developed with minimal possible loops and conditions for reduced complexity and enhanced maintainability.
Each function should contain max 30 lines of code. If it crosses more than 30 lines, function should be broken in sub modules.

18. Function Structure

The following tips provide guidance for creating easy-to-read and easy-to-maintain code:

- Modularize the code for increased reusability and reduced redundancy.
- Code should be well-indented with tabs. (Tab width should be 4).
- Values passed and returned to the functions should use simple variables.
- The use of global variables within the function should be reduced. The scope of the variable should be decided based on the standards.

5.Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

Lines longer than 80 characters should be avoided, since they're not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length generally no more than 70 characters.

When an expression will not fit on a single line, it should be broken according to the below general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

- Here are some examples of breaking method calls:

```
function(longExpression1, longExpression2, longExpression3,
longExpression4, longExpression5);

var = function1(longExpression1,
function2 (longExpression2,
longExpression3));
```

- Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
+ 4 * longname6; // PREFER

longName1 = longName2 * (longName3 + longName4
- longName5) + 4 * longname6; // AVOID
```

- Following is the example of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
Object andStillAnother) {
...
}
```

- Line wrapping for if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
if ((condition1 && condition2) || (condition3 && condition4)
    ||!(condition5 && condition6)) {
doSomethingAboutIt();
}
```

- Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta: gamma;

alpha = (aLongBooleanExpression) ? beta
: gamma;
alpha = (aLongBooleanExpression)
? beta
: gamma;
```

6.Automation Scripts

Folder/package structure should be created based on the functionality/purpose of the scripts/files present in the respective folder.

Example:

- **Generic.fetchdata:** This package should contain all the java files related to fetching test data from the excel sheet for data driven test automation framework.

- **Generic.global**: This package should contain all java files related to global functionalities like Constants.java, DateTime.java, CreateLog.java, UtilFunctions (Re-usable functions), TestBaseClass.java, etc.

Example:

- **Constants.java->**This java file is used to maintain constant value for the overall projects.
- **DateTime.java ->** This java file should contain all the methods to get/create current time stamps.
- **CreateLog.java ->** This java file should contain all methods to generate the log files and capture test results**.**
- **ReadExcelFile.java ->**This java file should contain all methods to read and fetch test data from excel sheet.
- **UtilFunctions.java ->**This java file should contain all methods which are common across the application**.**
- **ConnectDB.java ->** This java file should contain functions definitions required to connect database**.**

- **PageFactory** : This package should consists of all java files specific to the web element's Ids present in the web page and domain specific functions.
- **Example:**
- LoginPage.java
- AccountPage.java

- **TestData :** This package should consists of all the excel sheet containing the test data and environment specific data. The test data for each module should be placed in separate excel document.

Example:

- TestData.xls

- Environmnet.xls
  Header names in the excelsheet should start uppercase.
  Example:
  TestId, Description, Enabled, UserId, Password, etc.

- **TestResult:** This folder should contain all the log files which is resultant of test case execution. The log files should be named as per the time of execution which makes the tracking of results easier. The Test results should be maintained in the form of log files which are generated runtime during every execution. These log files should be saved as text documents for easy understanding. The messages provided within the scripts should be saved in the log files.
  The naming convention followed for the log folders is mm-dd-yyyy_hh_mm_ss. Each log folder should contain various log files based on the <testcasename>.log
  Example: RC_T1_AdminTools_001_PASS.log.
- **TestScripts:** This package consists of the test scripts for all module and setup script for the test execution. Testscripts names should be decided based on the functionality and the testscript should contain group of test cases related to that functionality**.**
  Example:
  UsageTypes.java -> This java file should contain group of methods(testcases) as mentioned below:

```
public void RC_T1_UsageTypes_001_CreateUsageTypes()
public void RC_T1_UsageTypes_004_EditUsageTypes()
public void RC_T1_UsageTypes_006_SearchUsageTypeGrid()
```

7.General Guidelines

- An ampersand (&) for concatenating strings should be used instead of '+' symbol.
- Objects should be set to nothing for cleaning the memory.
- Only one variable should be declared in a line and all variable should be initialized as null/0/' ' while declaring them.
- There should not be more than 80 characters per line.
- The code should be properly indented.
- Variables should be declared using appropriate data types.
- Success/ failure can be logged inside sub-methods, instead of re-writing in all called places.
- Finite number of loops should be defined when we use "While" loops.
- If the first line of the method is failed, then control should move to catch block. It should not try executing second line of the method.
- Try-Catch-Finally blocks should be used for all methods.
- If similar kind of logic is used in more than one place, then reusable components should be used. Redundancy should be avoided.
- Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.