Table of Contents

[1. Overview 2](#_Toc536435356)

[2. Solution, Projects and File Organization 2](#_Toc536435357)

[2.1 File Organization 2](#_Toc536435358)

[3. Naming Convention 3](#_Toc536435359)

[3.1 Naming Conventions 3](#_Toc536435360)

[3.2 Pre-defined Test Project Name 3](#_Toc536435361)

[4. Application Configuration File 3](#_Toc536435362)

[5. Coding Styles 5](#_Toc536435363)

[5.1 Formatting 5](#_Toc536435364)

[5.1.1 Indentation 5](#_Toc536435365)

[5.1.2 Comments 6](#_Toc536435366)

[5.1.3 Statements 7](#_Toc536435367)

[5.2 BDD Formatting 9](#_Toc536435368)

[5.2.1 Feature File: 9](#_Toc536435369)

[5.2.2 xpath: 9](#_Toc536435370)

# Overview

This document provides coding conventions to be followed when developing test projects based on Automation BDD Testing Framework using Java on IntelliJ tool.

# Solution, Projects and File Organization

## File Organization

Project Name

scr

*Test Framework, contain library such as (element controls, driver, assertion…)*

main

java

assertion

drivers

elements

utilities

resources

*Test project.*

test

*Step definition that generate from feature file.*

*Contain base class.*

features

resources

ultilities

stepDefs

pages

runner

bases

java

*Feature files.*

# Naming Convention

**Capitalization Styles**

|  |  |
| --- | --- |
| Pascal Case | This convention capitalizes the first character of each word. Ex: TestCounterId |
| Camel Case | This convention capitalizes the first character of each word except the first one. Ex: testCounterId |
| Upper Case | This convention capitalizes all character of the word. Ex: PI |

## Naming Conventions

| **Identifier** | **Casing** | **Name’s Pattern** | **Example** |
| --- | --- | --- | --- |
| Class | Pascal Case  *Nouns or noun phrase* | None-test class: {ClassName} | SearchStep |
| TestClass: {ClassName} | BELogin |
| Struct | Pascal Case  *Nouns or noun phrase* | {StructName} | Member |
| Interface | Pascal Case  *Nouns or noun phrase* | I{InterfaceName} | IButton |
| Method | Camel Case  *Verbs or verb phrase* | None-test method: {methodName} | mouseClick |
| Property | Camel Case  *Nouns or noun phrase* | {propertyName} | testResults |
| Field *(Public, Protected,*  *or Internal)* | Camel Case  *Nouns or noun phrase* | {fieldName} | userName |
| Field (Private) | Camel Case  *Nouns or noun phrase* | {fieldName} | logFileName |
| Static Field | Camel Case  *Nouns or noun phrase* | {fieldName} | expirationDate |
| Constant | UPPER Case  *Nouns or noun phrase* | {CONSTANT\_NAME} | SYSTEM\_PATH |
| Variable | Camel Case  *Nouns or noun phrase* | {variableName} | responseFromSystem |
| Parameter | Camel Case  *Nouns or noun phrase* | {parameterName} | answerId |

## Pre-defined Test Project Name

|  | **Project & Project File’s Name** |
| --- | --- |
| Feature files | BELogin.feature  FELogin.feature |
| stepDefs | BELoginStep  FELoginStep |
| page | BELogin  FELogin |

# Application Configuration File

* Each application project in ICO ATF solution has its application configuration file which is used to store the settings for the automated test running against that application. With “Library” project, its application configuration file will store the settings applied for all tests such as environment to the run the tests, browser used to run the tests, etc.
* After compiling, the application configuration file of a project will be generated in project’s bin/Debug folder with the name: **ApplicationName.ApplicationType.config** where *ApplicationName* is the name of the application, *ApplicationType* is the type of application (for example, .exe, .dll – it will be .dll for test project), and .config is the required suffix.

Convention for Application Configuration File:

| **ICO Component / Application** | **File Name in project’s folder** | **Properties Set for App Configuration File** |
| --- | --- | --- |
| Library (not the test project) | Library.dll.config [\*] | Make sure [Copy to Output Directory] is set to “Copy if newer” |
| Other projects (test project) | app.config | Make sure [Copy to Output Directory] is set to “Do not copy” |

[\*] It’s a trick to change the name of application configuration file of “Library” project to Library.dll.config rather the default name of app.config. As the result of the property set [Copy to Output Directory] to “copy if newer”, this file will not be only copy to bin/Debug folder of “Library” project, but also in bin/Debug folder of any projects which use “Library”. With our ATF structure, in order to run the test from Visual Studio IDE, it’s necessary to have the Library.dll.config in bin/Debug folder of other project, while the compiler only creates Library.dll.config in bin/Debug folder of “Library” project.

# Coding Styles

## Formatting

Code formatting is about communication, and communication is the professional developer’s ﬁrst order of business.

### Indentation

Four spaces should be used as the unit of indentation. Tabbing should only be used when the editor translates it into spaces. Vertical justified indentation should be used for group matching parenthesis and brackets when the expression or code section is bigger than one line.

**Line Length**

It is allowed lines are less than 120 characters.

**Wrapping Lines**

When a line is too long, consider reducing nesting by encapsulation. If an expression still will not fit on a single line, break it according to these general principles:

* Do not reduce the length of a descriptive method or variable name just to make an expression fit on one line.
* 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.

Some examples:

|  |
| --- |
| D:\Working\Java Coding Standards\wrapping-line1.png |
| D:\Working\Java Coding Standards\wrapping-line2.png |
| D:\Working\Java Coding Standards\wrapping-line3.png |
| D:\Working\Java Coding Standards\wrapping-line4.png |

### Comments

Sometimes it is useful to comment out code while you are developing or debugging. However, *never* leave these commented-out lines in the code when it is checked in. This is what source code configuration is for. If you have multiple working versions of your code, check both in so you can switch between them and we have a record of your changes.

|  |  |  |
| --- | --- | --- |
| **Comment Type** | **Usage** | **Example** |
| Documentation Comments | Documentation comments are processed by javadoc, to produce API documentation in HTML format. Use documentation comments immediately before declarations of interfaces, classes, member functions, and fields. | D:\Working\Java Coding Standards\javadoc-comment.png  /\*\*  \* Adds two ints (a+b) and returns the result  \*  \* @param a first integer to add  \* @param b second integer to add  \*  \* @returns the sum of a+b  \* @throws OverflowException  \* if a+b exceeds the value representable by int  \*/  public int add(final int a, final int b) { |
| C-style Comments | The javadoc program ignores C-style comments. Use C-style comments for multi-line comments that are not part of the API documentation. | D:\Working\Java Coding Standards\non-javadoc-comment.png |
| Single-line Comments | The javadoc program ignores Single-line comments. Use single line comments internally within member functions to document business logic, sections of code, and declarations of temporary variables. | D:\Working\Java Coding Standards\single-line-comment.png |

### Statements

**Simple Statements**

Each line should contain at most one statement. Example:

|  |
| --- |
| D:\Working\Java Coding Standards\single-statement.png |

**Compound Statements**

Compound statements are statements that contain lists of statements enclosed in braces:

|  |
| --- |
| D:\Working\Java Coding Standards\compound-statement.png |

* The enclosed statements should be indented one more level than the compound statement.
* Braces are used around all statements, even single code-line statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

**return Statements**

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

|  |
| --- |
| D:\Working\Java Coding Standards\return-statement.png |

**if-else Statements**

The if-else class of statements should have the following form:

|  |
| --- |
| D:\Working\Java Coding Standards\if-else-statement.png |

**if** statements always use braces {}.

Positive conditionals are easier to read than negative conditionals.

**for Statements**

A for statement should have the following form:

|  |
| --- |
| D:\Working\Java Coding Standards\for-statement.png |

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

|  |
| --- |
| D:\Working\Java Coding Standards\empty-for-statement.png |

When using the comma operator in the initialization or update clause of a **for** statement, avoid the complexity of using more than three variables. If needed, use separate statements before the **for** loop (for the initialization clause) or at the end of the loop (for the update clause).

**while Statements**

A **while** statement should have the following form:

|  |
| --- |
| D:\Working\Java Coding Standards\while-statement.png |

An empty while statement should have the following form:

|  |
| --- |
| D:\Working\Java Coding Standards\empty-white-statement.png |

**do-while Statements**

A **do-while** statement should have the following form:

|  |
| --- |
| D:\Working\Java Coding Standards\do-while-statement.png |

**switch Statements**

A **switch** statement should have the following form:

|  |
| --- |
| D:\Working\Java Coding Standards\switch-statement.png |

Every time a case falls through (doesn’t include a **break** statement), add a comment where the **break** statement would normally be. This is shown in the preceding code example with the /\* falls through \*/ comment. Every **switch** statement should include a **default** case. The **break** in the **default** case is redundant, but it prevents a fall-through error if later another case is added.

**try-catch Statements**

A **try-catch** statement should have the following format:

|  |
| --- |
| D:\Working\Java Coding Standards\try-catch-statement.png |

A **try-catch** statement may also be followed by **finally**, which executes regardless of whether or not the try block has completed successfully:

|  |
| --- |
| D:\Working\Java Coding Standards\try-catch-finally-statement.png |

## BDD Formatting

### Feature File:

| **No** | **Convention** |
| --- | --- |
| 1 | Should follow best practice in Cucumber slide |
| 2 | Should using background for same steps. |
| 3 | Test Case are not too long |
| 4 | Should have Description in each Feature file |

### xpath:

| **No** | **Convention** |
| --- | --- |
| 1 | A locator should be as small as possible  By.xpath(“//div[@id=’notifications’]”)  By.xpath(“//\*[@id=’notifications’]”)  By.xpath(//div[@id=’notifications-20140337152741’]  By.xpath(“//div[contains(@id, ‘notifications’)]”) |
| 2 | A locator should survive changes to its own properties  By.xpath(“//div/[@class=’menu-item menu-item-active accordeon’]”)  By.xpath(“//div/[contains(@class, ‘menu-item-active’)]”) |
| 3 | Should use String.format()   * If you plan on your app being localisable you should also get into the habit of specifying argument positions for your format tokens as well: * "Hello %1$s the time is %2$t" * This can then be localised and have the name and time tokens swapped without requiring a recompile of the executable to account for the different ordering. With argument positions you can also re-use the same argument without passing it into the function twice:   String.format("Hello %1$s, your name is %1$s and the time is %2$t", name, time) |