Selenium and Java Programming Best Practices

**Revision History**

|  |  |  |
| --- | --- | --- |
| **Date** | **Author** | **Comments** |
| October 21, 2019 | P. Christian | Largely lifted from the web. “Original” version. Needs work. |
| January 22, 2020 | P. Christian | Added potential references. |
| February 6, 2020 | P. Christian | Incorporating B. Keenan’s comments, add’l rework. |
| February 11,2020 | P. Christian | More B. Keenan input. |
| May 28, 2020 | G. Cosmiano | Added selenium standards |
| April 26, 2021 | G. Cosmiano | Added code templates for Eclipse |
| July 22, 2021 | G. Cosmiano | Added logger shortcut for Eclipse |
| Sept 20, 2021 | G. Cosmiano | Junit 5 Dynamic Tests |
| 02/10/22 | G. Cosmiano | Updated to include [selenium code template](#_Selenium_Templates) |

Contents

[1 Overview 5](#_Toc95405136)

[1.1 Purpose 5](#_Toc95405137)

[1.2 Approach 5](#_Toc95405138)

[2 Java 5](#_Toc95405139)

[2.1 Java source file 5](#_Toc95405140)

[2.1.1 Length 5](#_Toc95405141)

[2.1.2 Organization 5](#_Toc95405142)

[2.2 Naming 6](#_Toc95405143)

[2.3 Formatting to maximize legibility 6](#_Toc95405144)

[2.4 Spacing 7](#_Toc95405145)

[2.5 Comments 7](#_Toc95405146)

[2.5.1 Two types of comments 7](#_Toc95405147)

[2.5.2 Implementation comments 8](#_Toc95405148)

[2.5.3 Bad, better, great documentation comments 9](#_Toc95405149)

[2.5.4 Documentation comments, aka Javadoc 10](#_Toc95405150)

[2.5.5 Resulting HTML javadoc 11](#_Toc95405151)

[2.6 Annotations 11](#_Toc95405152)

[2.7 Test case steps 11](#_Toc95405153)

[2.8 Switch statements 11](#_Toc95405154)

[2.9 Exception messages: 11](#_Toc95405155)

[2.10 Iterators and streams: 12](#_Toc95405156)

[2.11 Declarations and assignments 12](#_Toc95405157)

[2.12 Encapsulation 12](#_Toc95405158)

[2.12.1 Web objects 12](#_Toc95405159)

[2.13 Input data 13](#_Toc95405160)

[2.13.1 Spreadsheets 13](#_Toc95405161)

[2.13.2 Javabean 13](#_Toc95405162)

[2.14 Output 13](#_Toc95405163)

[2.14.1 Spreadsheets 13](#_Toc95405164)

[2.14.2 Junit reporting 13](#_Toc95405165)

[2.14.3 Errors 13](#_Toc95405166)

[2.14.4 Logs 13](#_Toc95405167)

[3 Junit 5 14](#_Toc95405168)

[3.1 Dynamic Test 14](#_Toc95405169)

[4 Eclipse 15](#_Toc95405170)

[4.1 Code Templates 15](#_Toc95405171)

[4.2 Logger 16](#_Toc95405172)

[4.3 Selenium Templates 17](#_Toc95405173)

[4.4 Autoformatting in Eclipse 19](#_Toc95405174)

[5 Selenium 20](#_Toc95405175)

[5.1 Naming conventions 20](#_Toc95405176)

[5.2 Page Object model 22](#_Toc95405177)

[5.3 Page Object implementation 24](#_Toc95405178)

[5.3.1 Guidelines for creating page object classes 24](#_Toc95405179)

**References**

<https://medium.com/@rhamedy/a-short-summary-of-java-coding-best-practices-31283d0167d3>

<https://www.pluralsight.com/guides/getting-started-with-page-object-pattern-for-your-selenium-tests>

<https://seleniumjava.com/2017/05/20/correct-and-incorrect-ways-of-creating-page-object-classes/>

<https://blog.getgauge.io/are-page-objects-anti-pattern-21b6e337880f>

<https://martinfowler.com/bliki/PageObject.html>

<https://www.coveros.com/using-the-page-object-model-pattern-for-selenium-tests/>

# Overview

## Purpose

The purpose of this document is to summarize the coding practices that are most effective in the Java and Selenium test automation environment, with the end goal of test automation code that is easy to understand and maintain.

## Approach

This document is extensively leveraged from references that have been published on the web and saved in the test automation team’s local reference libraries.

# Java

## Java source file

### Length

The source file length is less than 2,000 lines long.

### Organization

The source file has documentation, the package declaration, grouped imports, class comments and class/interface signature, see [Code Templates](#_Code_Templates) section to setup copywrite and class java doc.

Here is an example:

/\*\*

\*

\* **@copyright** 2020 Excellus BCBS

\* All rights reserved.

\*

\*/

**package** runner.benefit;

**import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;

**import** **static** org.junit.jupiter.api.DynamicContainer.*dynamicContainer*;

**import** java.text.ParseException;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.junit.jupiter.api.DynamicContainer;

**import** org.junit.jupiter.api.DynamicTest;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.context.support.AbstractApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** configuration.ProjectOneTestBase;

**import** configuration.TestConfiguration;

**import** configuration.TestConfigurationException;

**import** datatestobject.benefit.BenefitsDTO;

**import** datatestobject.benefit.defaults.enterdetails.BenefitEnterDetailsDefaultDTO;

**import** datatestobject.benefit.mapdefintion.BenefitsDefineMappingDTO;

**import** datatestobject.workflow.BenefitApproveAndHoldDTO;

**import** exceptions.ElementNotFoundException;

**import** exceptions.UnexpectedDataConfiguration;

**import** pages.Actions;

**import** runner.benefit.creation.BenefitRunner\_EnterDetails;

**import** runner.benefit.creation.BenefitRunner\_MappingDefintion;

**import** runner.benefit.creation.BenefitRunner\_Review;

**import** runner.benefit.workflow.BenefitRunner\_DevPending;

/\*\*

\* Description: Benefit Runner used for creating a benefit from the DEV-INITIATION state to

\* IMPL-INTIATION.

\* The class utilizes each of the BenefitRunners used for benefit creation and workflow.

\* The runner will load the expected results and the data used to drive the test automation.

\* Once the test execution is complete from the runner stores the results in the

\* TestConfiguation Object and returns it to the test class.

\*

\* **@author** Brian Keenan (bkeenan)

\* **@since** 01/23/2020

\*/

**public** **class** BenefitRunner

## Naming

* Class and interface names are CamelCase. Use the whole words and avoid acronyms and abbreviations if possible
  + Examples: class Raster or class ImageSprite
* File names are CamelCase and end with .java, matching the class name. One Public class per file
* Method names should be verbs in mixed case with each internal word capitalized
  + Examples: run(); or runFast();
* Constants should be uppercase with “\_” separating each word
  + Example: int MIN\_WIDTH = 44;
* Variables have names that easily tell the reader what the variable represents. Keep the name short, avoid including metadata in the name.
  + Examples:
    - int schoolId;
    - int[] filteredSchoolIds;
* Tags will be in upper camel case: “ProductTemplate”.
* Variable names should be in lower camel case. Intent of the variable shall be indicated with the given name; it should be a noun, simple and descriptive.

## Formatting to maximize legibility

* Indentation: use the tab key and stay consistent
* Line length: Eliminate the need for horizontal scrolling. Use line breaks as needed to avoid horizontal scrolling.
* Well formatted method declaration. Use line breaks for legibility, indentation to clearly indicate parameters, etc.
* Always use braces following an if statement, even with a single-line statement that is executed when the if condition is met. DO NOT DO THIS:

if (condition)

statement;

* Use parentheses for clarity whenever possible. Do this:

if ((a == b) && (c == d))

{

Statement(s)

}

DO NOT DO THIS:

if (a == b && c ==d)

{

Statement(s)

}

* Use indentation to clearly distinguish between conditions and statements:

if (( condition1 && condition2)

|| (condition3 && condition4)

||!(condition 5 && condition6))

{

Do something

}

## Spacing

Avoid the temptation of saving a few lines of code at the expense of readability. Do the following

1. One blank line between methods
2. For Spring – two blank lines after constructors, static block, fields and inner class.
3. Put spaces between operators. Do this foo = a + b + 1; rather than foo = a+b+1;
4. Separate all binary operators except “.” From operands using a space.
5. Open brace “{“ appears on its own line, indented. The contents of that block are further indented. The closing brace “}” is on its own line, aligned with the opening brace. See preceding section for examples.

## Comments

Comments are used to communicate the intent of the code, particularly where it is not obvious. Comments give an overview and provide additional information that is not readily available in the code itself.

### Two types of comments

Implementation comments are meant to comment out, or comment about, a section of code.

Documentation comments describe the specification of the code from an implementation-free perspective. ***When you feel compelled to add a comment, consider rewriting the code to make it clearer.***

### Implementation comments

#### Block comment

// Block comment

/\*

* Usage: Provides description of files, methods, data structures
* and algorithms. Can be used at the beginning of each file and \* before each method. Used for long comments that do not fit a \* single line. 1 Blank line to proceed after the block comment. \*/

#### Single line comment

// Single line comment

If (condition)

{

/\* handle the condition \*/

…

}

#### Trailing comment

// Trailing comment if (a == 2)

{

return TRUE; /\* special case \*/

}

else

{

return isPrime(a); /\* works only for odd a \*/

}

#### End of line comment

End of line comments continue to the newline. It can comment out a complete line or only a partial line. It ***should not be used on consecutive multiple lines for text comments***; however, ***it can be used in consecutive multiple lines for commenting out sections of code.***

// End of line comment

if (foo > 1)

{

// Do a double-flip.

...

}

else

{

return false; // Explain why here.

}

//if (bar > 1)

//{

//

// // Do a triple-flip.

// ...

//}

//else

//{

// return false;

// }

### Bad, better, great documentation comments

Read these through. Which would you like to find in code you are maintaining, under pressure?

// Bad.

// - The doc tells nothing that the method declaration didn't.

// - This is the 'filler doc'. It would pass style checks, but

//doesn't help anybody.

/\*\*

* Splits a string.

* [@param](http://twitter.com/param) s A string.
* [@return](http://twitter.com/return) A list of strings. \*/

List<String> split(String s);

// Better.

// - We know what the method splits on.

// - Still some undefined behavior.

/\*\*

* Splits a string on whitespace.
* [@param](http://twitter.com/param) s The string to split. An {[@code](http://twitter.com/code) null} string is
* treated as an empty string.
* [@return](http://twitter.com/return) A list of the whitespace-delimited parts of the input.

\*/

List<String> split(String s);

// Great.

// - Covers yet another edge case.

/\*\*

* Splits a string on whitespace. Repeated whitespace characters
* are collapsed.

\*

* [@param](http://twitter.com/param) s The string to split. An {[@code](http://twitter.com/code) null} string is treated
* as an empty string.
* [@return](http://twitter.com/return) A list of the whitespace-delimited parts of the input.

\*/

List<String> split(String s);

### Documentation comments, aka Javadoc

Javadoc is a tool that generates HTML documentation from java code using the comments that begin with /\*\* and end with \*/. See Wikipedia for more details on how it works.

Use Javadocs to comments on the methods of a class. The Javadoc does not have to be used for getters and setters for variables that do not need customization.

#### Example of Javadoc in the code

/\*\*

* Returns an Image object that can then be painted on the screen.
* The url argument must specify an absolute {[@link](http://twitter.com/link) URL}. The name
* argument is a specifier that is relative to the url argument.
* <p>
* This method always returns immediately, whether or not the
* image exists. When this applet attempts to draw the image on
* the screen, the data will be loaded. The graphics primitives
* that draw the image will incrementally paint on the screen.
* [@param](http://twitter.com/param) url an absolute URL giving the base location of the image
* [@param](http://twitter.com/param) name the location of the image, relative to the url argument
* [@return](http://twitter.com/return) the image at the specified URL
* [@see](http://twitter.com/see)  Image

\*/

public Image getImage(URL url, String name)

{

try

{

return getImage(new URL(url, name));

}

catch (MalformedURLException e)

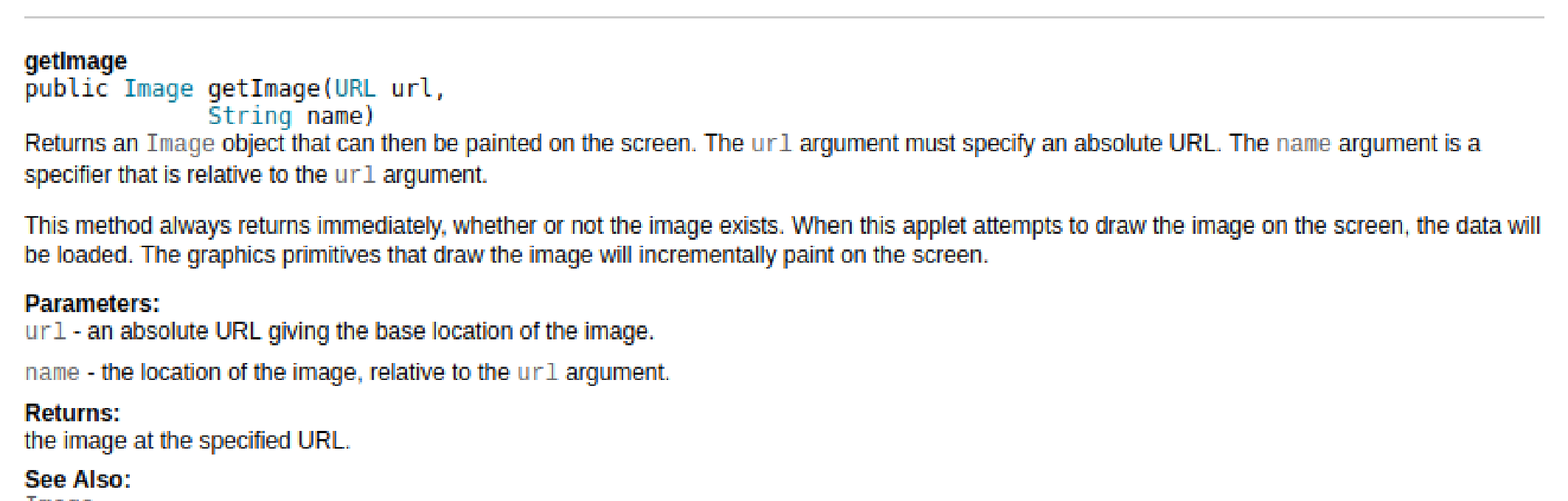
{

return null;

}

}

### Resulting HTML javadoc



## Annotations

## Test case steps

It is helpful, probably even necessary, to identify the test steps from the manual test cases which are provided as inputs to the test automation process. This is particularly useful during maintenance, when a small change may need to be inserted into a long chain of actions.

Comments can be used to indicate the steps from the manual test cases.

Junit supports logging the test case steps.

## Switch statements

Always have a **default** case even without code.

Use /\* falls through \*/ to indicate the control falls to next case:

switch (condition)

{

case ABC:

Statement

/\* falls through \*/

case DEF:

statement;

break;

default:

statement;

break;

}

## Exception messages:

Format the statements for ease of legibility like this:

throw new IllegalStateException("Failed to process"

+ " request " + request.getId()

+ " for user " + user.getId()

+ " query: '" + query.getText() + "'");

## Iterators and streams:

Use the same indenting principles demonstrated in the examples above to keep these often-complex statements legible.

## Declarations and assignments

Put declarations only at the beginning of blocks. A block is code surrounded by curly braces. Do not wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

**One declaration per line**

int level // indentation level

int sizeMeter // size of table

**Avoid mixing types – don’t do this:**

Int foo, fooarray[];

**Avoid multiple assignments – don’t do this:**

Foobar.fChar = barFoo.lchar = ‘c’;

**Avoid embedded assignments – don’t do this:**

d = (a = b + c) = r;

// instead, do this:

a = b + c;

d = a + r;

**Declare arrays this way:**

String[] args

## Encapsulation

### Web objects

A small change in a web object can potentially cause changes rippling through the code. We will use encapsulation to bind together the data and functions that manipulate the data and hide the implementation details from the test logic. This is also known as data hiding.

We will use setter and getter public methods to modify and view the web objects, in most cases.

## Input data

Input data can come from spreadsheets or Javabeans.

### Spreadsheets

Some test programs will retrieve input data from spreadsheets. Accessing this data, and writing results to a spreadsheet, will be encapsulated.

### Javabean

Javabeans can be used as a source of input data.

## Output

### Spreadsheets

### Junit reporting

### Errors

### Logs

Used for intent and history.

# Junit 5

## Dynamic Test

The [Dynamic Test](https://junit.org/junit5/docs/snapshot/user-guide/#writing-tests-dynamic-tests) is a feature in Junit 5 that allows users to create tests on runtime. However, dynamic test **cannot** be used with an assertion where the object being evaluated (example: Selenium object or page object) will change during test method runtime because per Junit 5 documentation below,

“

These test cases are static in the sense that they are fully specified at compile time, and their behavior cannot be changed by anything happening at runtime.

”

This means, a test code below will not work (or fails) because the Selenium Page Object (LoginPO) will change during runtime. The code below will generate at runtime **static** tests using test factory and the assertions will happen **after** the test method are fully executed. Therefore, the username, password and login button would no longer exist at the time assertion occurs since the test clicks the login button at the end of the test method.



The work around for this is to set variables (example Data Test Objects – DTO) for those that needs to be validated from Selenium Page Object. Here is how the code above should be written to use dynamic tests correctly:



# Eclipse

## Code Templates

Setup the code templates as follows

1. Open Eclipse
2. Window à Preferences à Java à Code Style à Code Templates
3. Expand **Code** and select **New Java files**
4. Add the following lines but change the first/last name (highlighted in yellow) to your name

/\*\*

\*

\* **@copyright** ${year} Excellus BCBS

\* All rights reserved.

\*

\*/

${filecomment}

${package\_declaration}

/\*\*

\*

\*

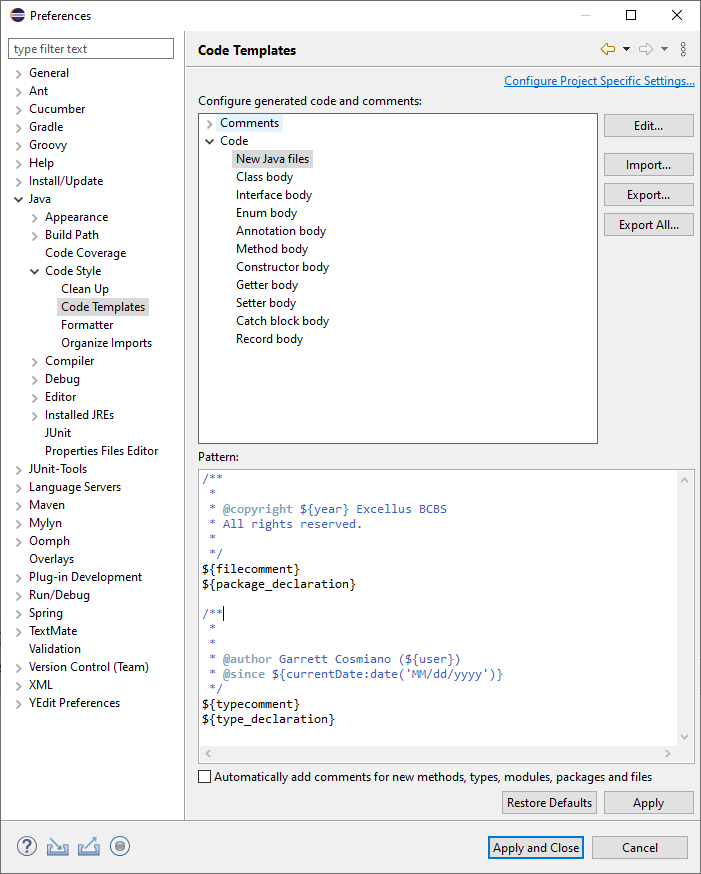
\* **@author** First\_name Last\_name (${user})

\* **@since** ${currentDate:date('MM/dd/yyyy')}

\*/

${typecomment}

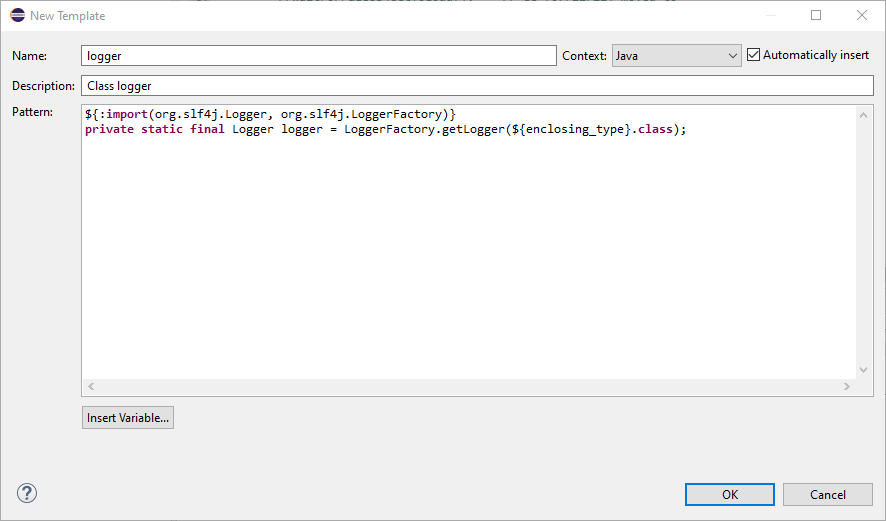
${type\_declaration}



## Logger

Setup logger (using [SLF4J](http://www.slf4j.org/)) shortcut template

1. Open Eclipse
2. Window à Preferences à Java à Editor à Templates
3. Click **New**… button to add a new shortcut
4. Fill the form as follows then click OK

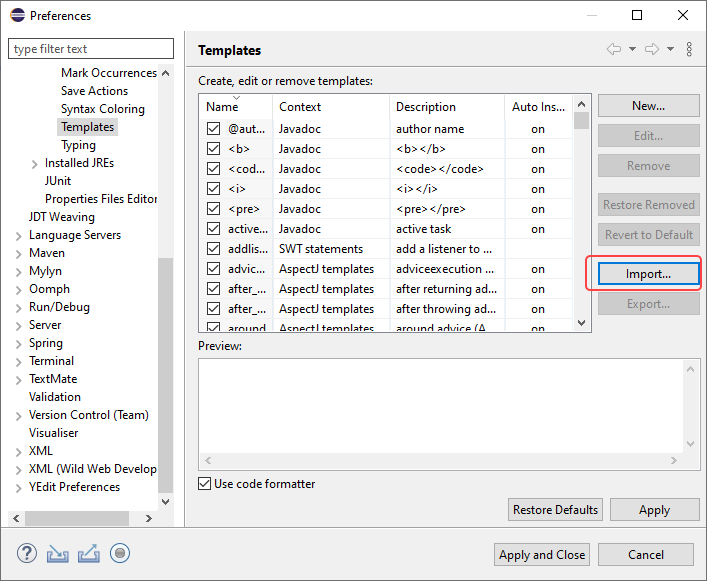


1. Click **Apply and Close** button

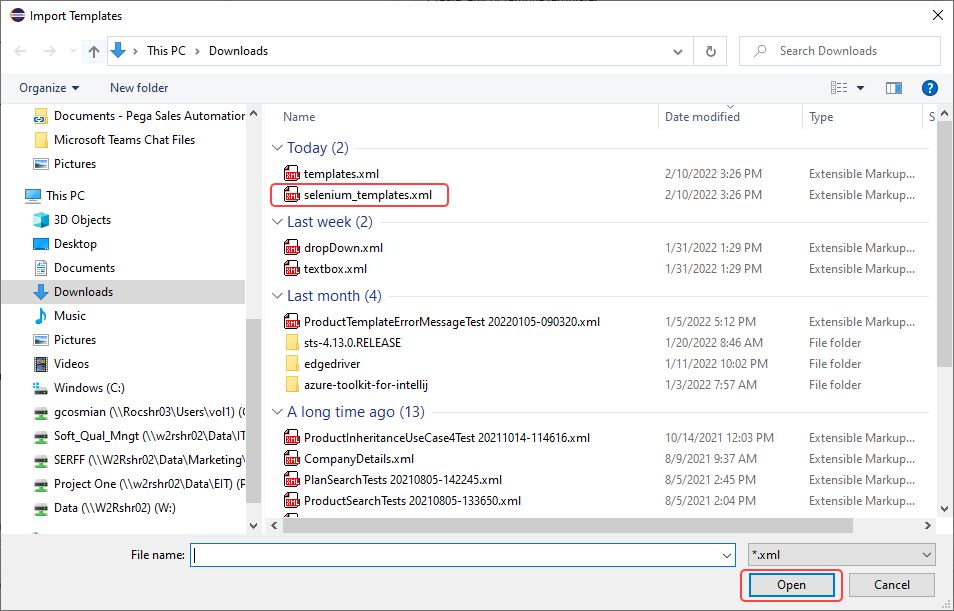
## Selenium Templates

NOTE: make sure to follow the WebElement [naming convention](#_Naming_conventions)

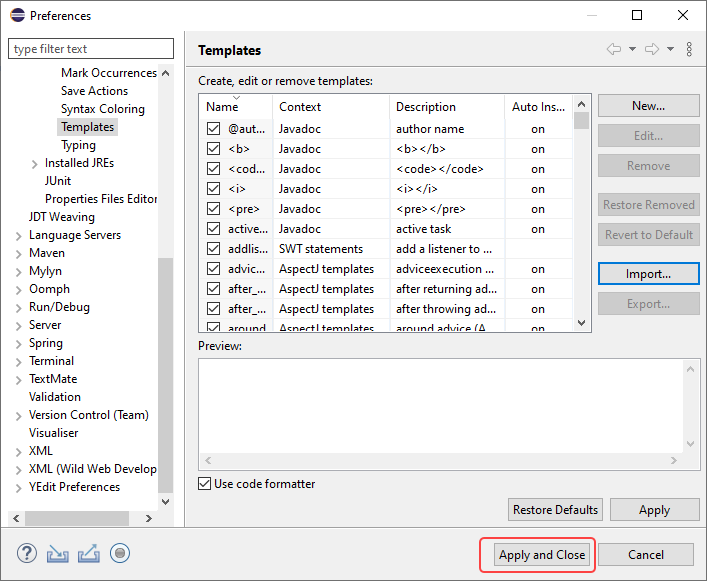
1. Download [​xml icon selenium\_templates.xml](https://lifethc.sharepoint.com/:u:/r/sites/SQAArchitectureAndAutomationTeam_EIT_GRP/Shared%20Documents/General/Technical%20Testing%20References/Java/Java_Code_Eclipse_Templates/selenium_templates.xml?csf=1&web=1&e=kjlJYS) from [SharePoint](https://lifethc.sharepoint.com/sites/SQAArchitectureAndAutomationTeam_EIT_GRP/Shared%20Documents/Forms/AllItems.aspx?id=%2Fsites%2FSQAArchitectureAndAutomationTeam%5FEIT%5FGRP%2FShared%20Documents%2FGeneral%2FTechnical%20Testing%20References%2FJava%2FJava%5FCode%5FEclipse%5FTemplates&viewid=0381e173%2D4e09%2D459b%2Db69d%2Da5a45405edcb).
2. Open Eclipse (double C:\Users\gcosmian\eclipse\eclipse\_2021-06\eclipse.exe)
3. From the Preference dialog box, go to **Java** > **Editor** > **Templates**
4. Click **Import** button



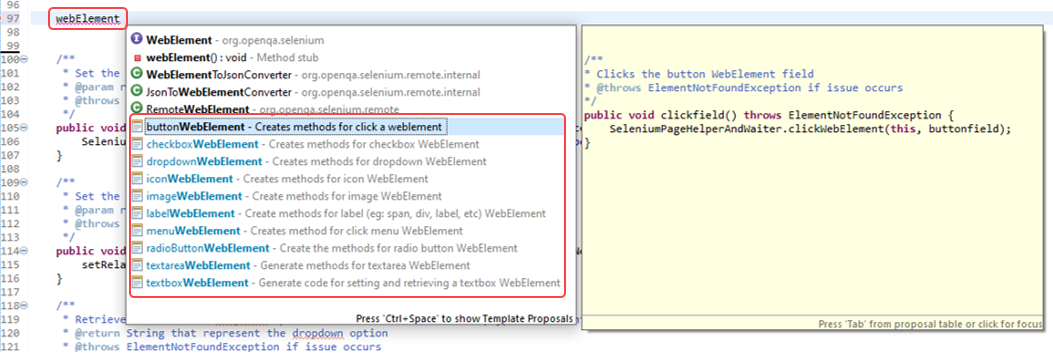
1. Select the selenium\_templates.xml file.



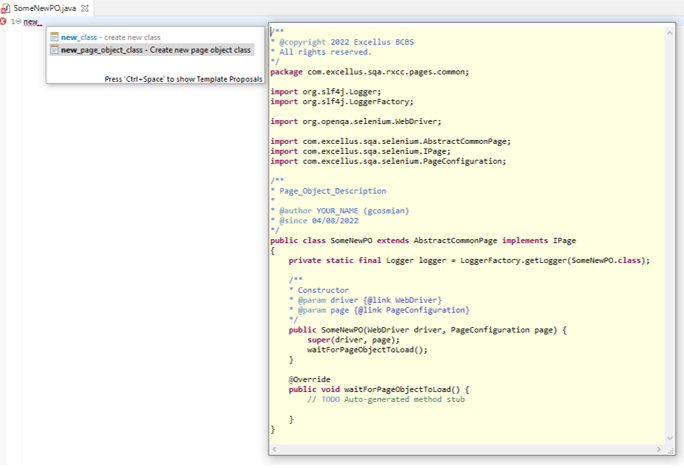
1. Click **Apply and Close** button



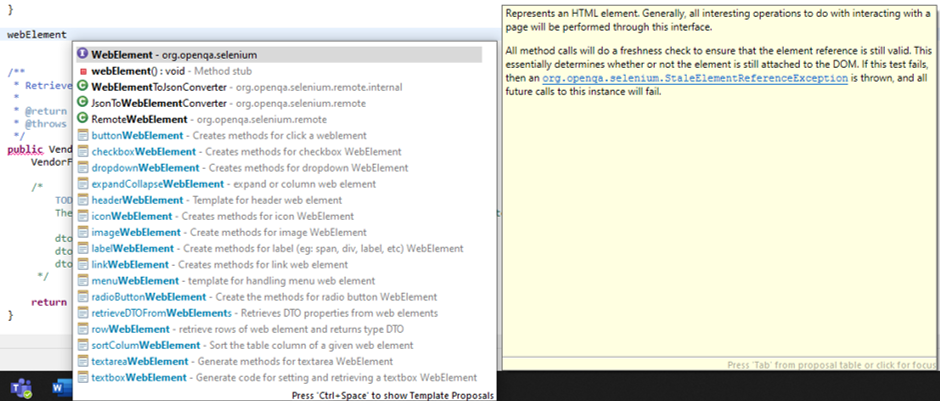
1. Using the new templates
   1. Create page object class and create the necessary web elements (see [​docx icon Selenium and Java Programming Best Practices.docx](https://lifethc.sharepoint.com/:w:/r/sites/SQAArchitectureAndAutomationTeam_EIT_GRP/Shared%20Documents/General/Technical%20Testing%20References/Java/Selenium%20and%20Java%20Programming%20Best%20Practices.docx?d=w69955b107ddd4f2694f36578860c3f0c&csf=1&web=1&e=QnDqVv) for naming convention of web elements)
   2. In the class, type **webElement** and press CTRL-space to bring up shortcuts



* 1. Select the corresponding WebElement type (example buttonWebElement). This will create the necessary methods.
  2. Change the fields, param, etc accordingly. Use ‘TAB’ key to move between fields and param.
  3. The template will provide class template (new\_page\_object\_class) for page object class



* 1. The templates will create default methods for the corresponding WebElement. Testers can update these codes as they see fit. The available templates are:



## Autoformatting in Eclipse

We have standardized using autoformatting in Eclipse, with a customized formatting profile. To apply:

1. Open Eclipse
2. WindowàPreferencesàJavaàCode StyleàFormatter
3. Import. Import this file: [\\w2rshr02\Data\IT\Soft\_Qual\_Mngt\AutomationServices\ReferenceDocuments\ProcessDocuments\Programming Best Practices\Selenium and Java\Eclipse\_Java\_Formatring.xml](file://w2rshr02/Data/IT/Soft_Qual_Mngt/AutomationServices/ReferenceDocuments/ProcessDocuments/Programming%20Best%20Practices/Selenium%20and%20Java/Eclipse_Java_Formatring.xml)
4. Apply and close.
5. Ctrl-Shift-f will apply the new formatting to files that existed prior to applying the autoformatting profile.

# Selenium

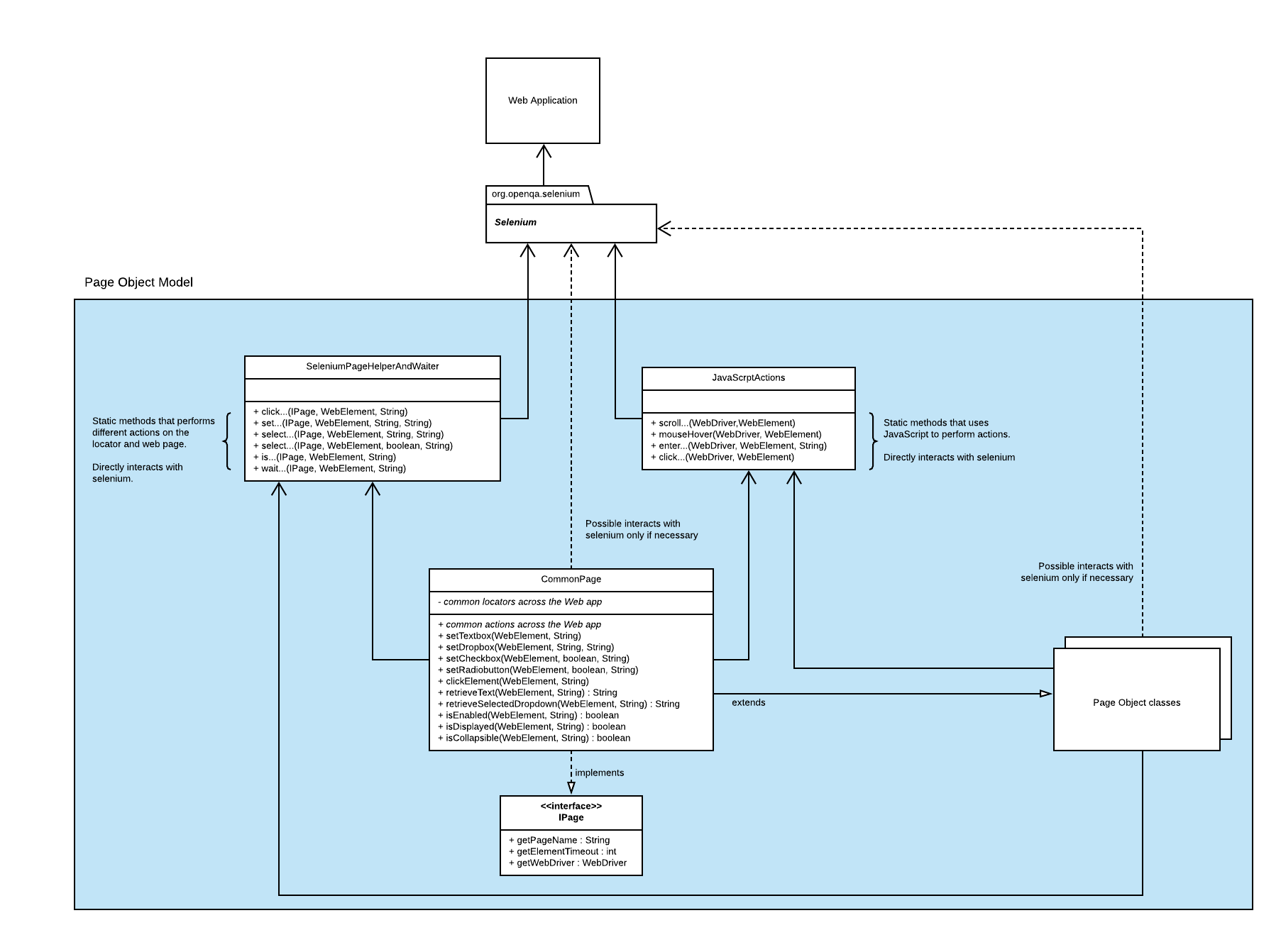
## Naming conventions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Variable Prefix** | **Variable Example** | **Action Prefix** | **Action Example**  **(method)** | **XML element example** | **Screenshots** |
| textbox | textboxEffectiveDate | set  retrieve | setEffectiveDate  retrieveEffectiveDate | input[@type='text']  textarea |  |
| button | buttonContinue | click | clickContinue | button[@type='button'] |  |
| dropdown | dropdownPlanDesign | select  retrieve | selectPlanDesign  retrieveSelectedPlanDesign  retrievePlanDesignOptions | select |  |
| checkbox | checkboxCopay | set  is | setCopay  isSelectedCopay | input[@type='checkbox'] |  |
| header | headerProdTempOvrw | retrieve  is | isDisplayedProdTempOvrw | h1  h2  h3 |  |
| expandCollapse | expandCollapseClientConfig | is  expand | isExpandedClientConfig  expandClientConfig(boolean) | //div[contains(@class, 'collapsible')]//h2[text()='Client confguration’] |  |
| radio | radioYesHSAQualified  radioNoHSAQualified | set  is | setYesHSAQualified  isYesHSAQualified  setNoHSAQualified  isNoHSAQualified  setHSAQualified(boolean yes) | input[@type='radio'] |  |
| label | labelMarketSegment  labelMrktSgmntNoItem | retrieve  is | isDisplayedMrktSgmt  isDisplayedMrktSgmtNoItem | div  span  label |  |
| table | tableAuditHistory |  |  | table |  |
| row | rowAuditHistory |  |  | tr |
| columnHeader | columnHeaderDescription | sort | sortDescription(Asc/Des) | th |
| column | columnAuditHistory |  |  | td |
| image | imagePegaLogo | is |  | img |  |
| icon | iconRecentFollowers | click | clickRecentFollowers | a[@data-ctl='Icon']  a[contains(@class,'icon')]  i[contains(@class,'icon')] |  |
| menu | menuBulkActions  menuNotifyPref  menuProfile  menuAboutThisApp  menuLogoff | click | clickLogoff | span  span[contains(@class,'menu')]  a[contains(@class,'menu')]  i[contains(@class,'menu')]  td[contains(@role,'menu')] |  |
| link |  |  |  |  |  |
| progressBar |  |  |  | div[@role='progressbar']  span[@role='progressbar'] |  |
| tab | tabRequestingProvider  tabServicingProvider  tabMember  tabClaimAuth | click  isSelected | clickRequestingProvider  isSelectedRequestingProvider | div[@data-role='tab'] |  |
| frame |  |  |  | iframe |  |
|  | *placeholder* |  |  |  |  |

## Page Object model

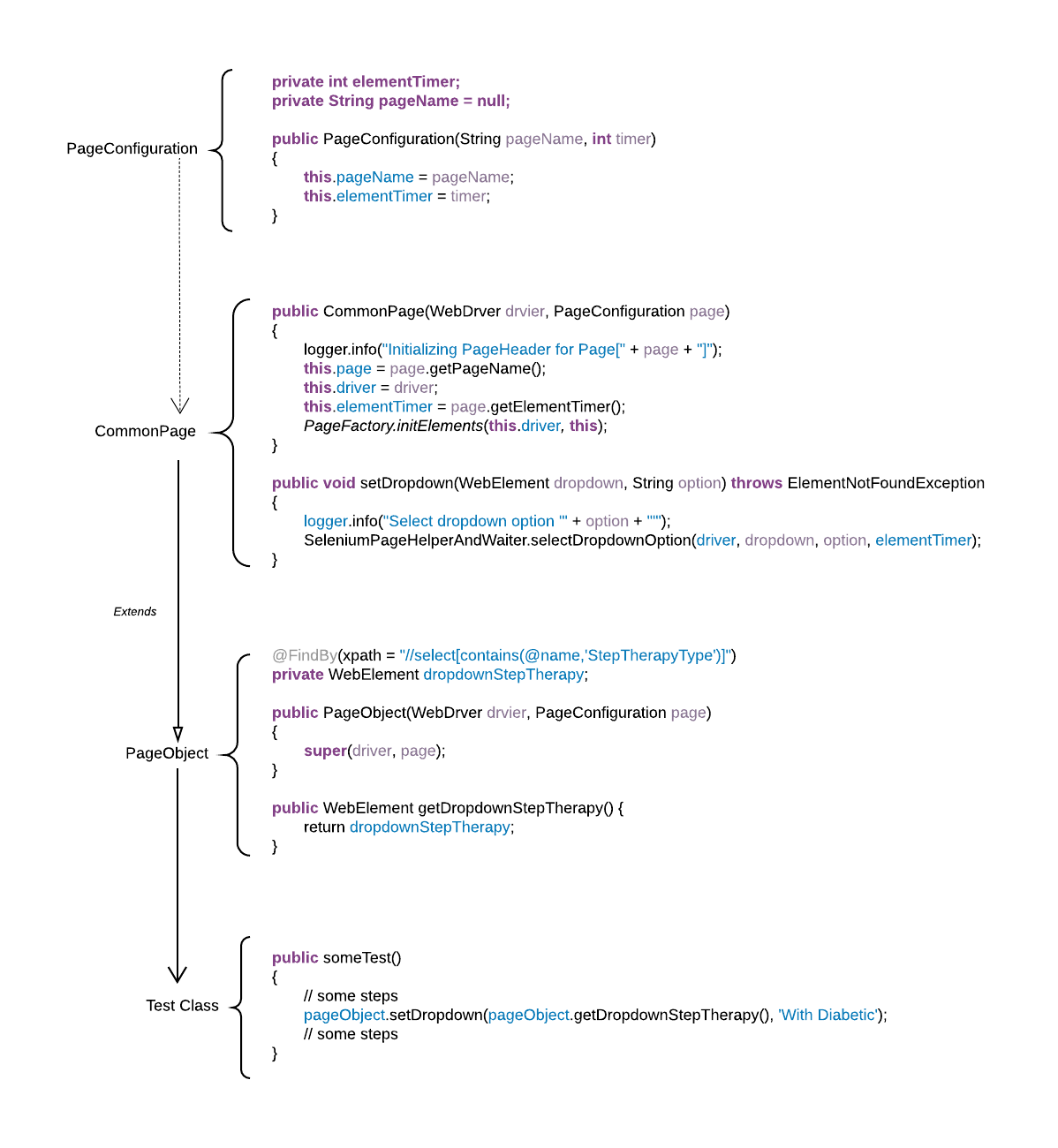
The test framework will follow the design pattern [Page Object](https://www.selenium.dev/documentation/en/guidelines_and_recommendations/page_object_models/). The main advantages of using this approach are:

1. Interaction/interface with the Web UI pages are done within the page object instead of test.
2. Avoids duplicated code caused by duplicate functionality or duplicate usage of locators.



The test framework will have these main page object components

* IPage – an interface that defines the method to retrieve the page name, default timeout to wait for a locator and WebDriver.
* CommonPage
  + Contains locators and its corresponding actions common to all/majority of the Web app pages.
  + Contains simple generic web element actions (methods) that takes at least a WebElement parameter. The purpose is to avoid page classes to have methods that performs simple action like selecting from dropdown. Instead, the method setDropdown(WebElement, String) is called passing the WebElement dropdown and an option to select, example code snippet below



* SeleniumPageHelperAndWaiter – contains static methods that directly interacts with Selenium.
* JavaScriptActions – contains static methods which uses JavaScripts to perform selenium actions.
* Page object classes – these are the classes that defines the locators for each web page with corresponding page actions/features.

## Page Object implementation

### Guidelines for creating page object classes

* **Extends CommonPage**: It is ideal to create a super class that is extended by each page object. The super class defines the WebDriver, common properties and common WebElements as well as initialize the WebElements that have been annotated with @FindBy. In our test framework, this is CommonPage class, code snippet below:

****

As mentioned, the page object will extend the CommonPage and it is required to have a constructor that calls the CommonPage constructor, example code snippet:



*Add note about page wait (Brian)*

* **PO classes - page component/feature:** Avoid creating page object classes that represent the entire web page. Instead, break them up into smaller classes that represents a specific component/feature of the web page that helps the user perform a specific task. This keeps it easier to maintain and can be reused if this component/feature of the web page is used in multiple pages. Here is an illustration where two web pages uses the same feature, ***Line of Business****.*



* **Page classes:**  A page class should represent the entire page which should consist of page component/feature. Here is an illustration of this

****

Code snippet of the page class

**Note** – the button Generate or Re-Generate and checkbox is part of the page class instead of PO classes.



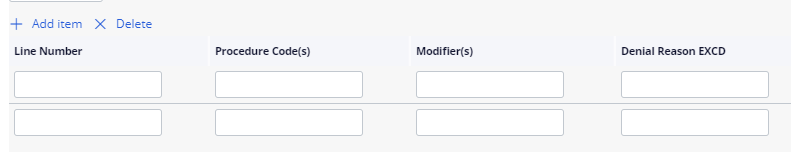
* **WebDriver:** The WebDriver (however it was initialized) must a single instance throughout navigating between page objects. Meaning the WebDriver being passed between page object must always be the same instance.
* **Handling tables:** There are several ways to handle tables and one recommendation is to use the an xpath the defines the table **rows** and then navigate through each row and each column. Here is an example:

****

Code snippet to retrieve data



Here is another example to set the data in the table



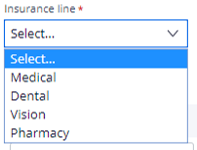
Code snippet to set data in the table



* **No business logic**: The page object should not contain any business logic, example: disabling checkbox that would disable/remove other WebElements. The page object primary objective is to locate elements and perform actions on those elements.
* **User interaction:** The page object should contain methods that groups user interaction for a given page feature. Classic example is the login page. Instead of defining 3 methods (i.e. enter username, enter password, click login button), define one method that takes two parameters (i.e. username and password)

****

* **No validations:** The page object should not contain any type of assertion.
* **Date picker:** The date picker is out of scope for testing and it the recommendation is to use .sendKey("some text") of the date picker textbox.
* **Dropdown:** It is recommended to use enum for dropdown options to avoid passing the wrong options and this provides easy validation.



****

Code snippet for selection an option from dropdown



Code snippet for validation of the dropdown options



* **Using @FindBy vs By vs String xpath**: The page object doesn’t require to use @FindBy annotation to define the WebElement or locator. As a rule of thumb, when a WebElement does not have direct action associated with it (example: img or label) or the object is conditional (can appear/disappear base on condition) then us **By**. Otherwise use **@FindBy** annotation.

****

These **By** locators can be validated using the CommonPage methods below



* **Using waits for web elements**: Using Thread.sleep to wait for web element is not a good practice and should be avoided as must as possible. Instead, use these Selenium helper methods (static) from the class ***SeleniumPageHelperAndWaiter***.

****

**Example usage** – the code below waits for a web element *textboxPlanOption* to either disappear or appear base on the selected options on a dropdown.

****

*Thread.sleep should be the last resort.*