**Locators :**

**tagname**

To create a webpage we need to use tagname <html> and to create a component or element we need to use tagname <input>

To create a listbox we use <select> tag

To create a link we use <a> tag (anchor)

To create a web table we use <table> tag

1. Tag Name - Any keyword which is present after < symbol is known as tagname. E.g. <html> <body> <input>
2. Attribute – Any keyword which is present after tagname with = symbol. E.g. type = ‘text’ value = ‘Login’ (type : attribute name & ‘text’ : attribute value)
3. Text - Any keyword which is present (><) in between greater than symbol (>) & less than symbol (<). E.g. link1, India, Aus

<html>

<body>

UN<input type = ‘text’ id = ‘abc’> </br>

PWD<input type = ‘password’ id = ‘456’> </br>

<a href = ‘url’> Link1 </a>

<input type = ‘Button’ value = ‘Login’> </br>

</body>

</html>

­­­­­­

**Locators :**

1. Locators are used to identify an element present in a webpage with the help of locator types.
2. To identify an element present in a webpage we need to use findElement() method which is present in WebDriver interface.
3. findElement() method will identify an element with the help of “By” class which contains static methods.
4. All the static methods present in a By class are known as locator types.

**There are different types of locators:**

* 1. ID
  2. Class
  3. Name
  4. Tagname
  5. Linked Text
  6. Partial Linked Text
  7. CSS Selector
  8. X-path

**Xpath :**

WebDriver driver = new ChromeDriver();

driver.get(“url”);

driver.findElement(By.xpath(“xpath\_expression”)).sendkeys(“abc”);

**Types of xpath :**

1. **xpath by attribute**

Syntax :

//tagname[@attribute name = ‘attribute value’]

e.g. :- //input[@id = ‘abc’], //input[@id='autocomplete']

driver.findElement(By.xpath(“//input[@id = ‘abc’]”).click();

1. **xpath by text**

Syntax :

//tagname[text() = ‘textvalue’]

e.g. :- //a[text() = ‘link1’]

1. **xpath by contains()**
2. Here we can mention the substring without writing a whole text, when link/text is long in size.
3. If any non-breakable space (nbsp) is there.

We can use contains in two ways :

1. Contains with text() :

//tagname[contains(text(),’textvalue’)]

//h2[text()='Facebook helps you connect and share with the people in your life.']🡪 normal text method

//h2[contains(text(),'Facebook ')]🡪using contains text

//a[contains(text(),‘link1’)]

1. Contains with attribute :

//tagname[contains(@attribute name, ‘attribute value’)]

//input[contains(@name,‘first’)], //img[contains(@alt,'gle')]

1. **xpath by index**

(//tagname[@attribute name = ‘attribute value’] )[2]

(//input[@type = ‘text’] )[2]

(//input[@name='radio'])[2]

(//h1[contains(text(),'Practice Page')])[1]

1. **Absolute and Relative xpath**

html/body/div[2]/input[1]

//div[2]/input[1]

Root🡪 html/body/ div[1]/ input[1]🡪absolute

// div[1]/input[1]🡪 relative xpath

/html/body/div[2]/input[1]-🡪//div[2]/input[1]

**HTML Tree Diagram**

<html>

<body>

<div>

UN<input type = ‘text’>

PWD<input type = ‘password’>

</div>

<div>

male<input type = ‘radio’>

female<input type = ‘radio’>

</div>

<div>

<a href = ‘\_’> link1

<a href = ‘\_’> link2

<button type = ‘submit’ value = ‘login’>

</div>

</body>

</html>

**Absolute Relative**

/html/body/div[3]/a[1] //div[3]//a[1]

/html/body/div[3]/input //div[3]//input or input[5]

/html/body/div[2]/input[2] //div[2]//a[2] or input[4]

**Disadvantages of Absolute xpath**

1. Absolute xpath is too lengthy and time consuming.
2. Identifying an element by developing html tree diagram is difficult.

**Absolute xpath**

Absolute xpath is used to navigate from root of parent to its **immediate** child. To achieve absolute xpath, we need to use ‘/’.

/html/body/div[2]/input[1]

**Relative xpath**

Relative xpath is used to navigate from parent to **any child**. To achieve relative xpath, we need to use ‘//’.

//div[2]//input[1]

html/body/section/div[1]/div[1]

//section[2]//div//div

**CSS selector**

|  |  |  |
| --- | --- | --- |
| **Method** | **Target Syntax** | **Example** |
| Tag and ID | css=*tag*#*id* | css=input#email |
| Tag and Class | css=*tag*.*class* | css=input.inputtext |
| Tag and Attribute | css=*tag*[*attribute*=*value*] | css=input[name=lastName] |
| Tag, Class, and Attribute | css=tag.class[attribute=value] | css=input.inputtext[tabindex=1] |

**How to handle dynamic XPath**

dynamic XPath expressions in automation testing is a common challenge, as web pages often generate elements with changing attributes or values. Dynamic XPath refers to XPath expressions that are subject to change due to factors like session IDs, timestamps, or random values. Here are strategies to handle this situation:

**Using Partial Matching:**

Utilize XPath functions like contains() or starts-with() to match a portion of the dynamic attribute value. For instance:

Xpath - //input[contains(@id, 'dynamicPart')]

**Using text() and ancestor Relationships:**

If the dynamic element is identified by its text content or its relationship with another element, you can use these attributes for more stable identification:

Xpath - //label[text()='Dynamic Text']/following-sibling::input

**Using following and preceding Axes:**

These axes allow you to navigate to elements after or before another element. This can be helpful when dealing with sibling elements that have a predictable order.

Xpath - //div[@class='dynamicDiv']/following::input

**Combining Multiple Attributes:**

If an element has multiple attributes that remain stable, use them in combination to form a unique identification:

Xpath - //input[@id='staticID' and @class='dynamicClass']

**Parent-Child Relationship:**

Traverse the hierarchy of parent and child elements to create a more stable XPath:

Xpath - //div[@class='parentDiv']//input[@class='dynamicClass']

**Using position() Function:**

If the element's position in the hierarchy is stable, you can use the position() function to target it:

Xpath - (//button[@class='dynamicClass'])[2]

**Using Indexed Position:**

If you know that the dynamic element is consistently the nth child of a certain type, you can use its index:

Xpath - //div[@class='dynamicContainer']/input[3]

**Using descendant Axis:**

This axis can be useful when the dynamic element is nested within another element:

Xpath - //div[@class='parentDiv']//descendant::input[@class='dynamicClass']

**Synchronization**

Synchronization/wait: matching selenium test script speed with browser speed

Selenium – Synchronization - matching selenium test script speed with browser speed.

Wait-

1.Static wait

2. Dynamic wait

**Static wait: java wait**

**1. Thread.Sleep**

Thread.sleep(3000); // Sleep for 3 seconds

Thread.Sleep is a static wait

Wait time is fixed, irrespective of condition. Script will be hold for mentioned seconds.

**Dynamic wait: Selenium**

**2. Implicit Wait**

driver.manage().timeouts().implicitlyWait(Duration.*ofMillis*(6000));

Applicable: complete webpage // wait for full webpage loding

1 parameter: time value (seconds)

//2000ms🡪 100ms🡪1900 release wait time

Advantage of implicit wait is that suppose we have mension time 5000 ms and our page load in 1000 ms then it will save remaining 4000 ms for nest task.

**3. Explicit wait**

WebDriverWait w = new WebDriverWait(driver, Duration.*ofMillis*(15000));

w.until(ExpectedConditions.*visibilityOfElementLocated*(By.*xpath*("(//h6[text()='Sign In'])[2]")));

//w.until(ExpectedConditions.*elementToBeClickable*(By.*xpath*("(//h6[text()='Sign In'])[2]")));

//w.until(ExpectedConditions.*visibilityOf* (By.*xpath*("(//h6[text()='Sign In'])[2]")));

Applicable: single element

1 parameter: time value(seconds)

2 parameter: Condition (isselected, isdisplayed, isenabled)

An explicit wait Applicable for single element.

It will wait until specific element to be load.

**4. Fluent Wait**

FluentWait<WebDriver> w = new FluentWait<WebDriver>(driver).

withTimeout(Duration.ofMillis(15000)).

pollingEvery(Duration.ofMillis(1000)). ignoring(NoSuchElementException.class,TimeoutException.class);

w.until(ExpectedConditions.elementToBeClickable(By.xpath("(//h6[text()='Sign In'])[2]")));

driver.findElement(By.xpath("(//h6[text()='Sign In'])[2]")).click();

Applicable: single element

1 parameter: time value(seconds)//1000ms

2 parameter: Condition (isselected, isdisplayed, isenabled)

3 frequency: time(time in sec)- 5 m sec🡪10ms

Fluent Wait will sets the maximum duration for the Selenium WebDriver to wait until a specific web element visible on webpage. it will try to find the element again and again during the mensioned wait time.

Suppose we have 6000 ms for an element to be available on the web page, but it will check its avilable once in every 500 seconds.

More flexible than explicit wait,allowing setting polling frequency and maximum wait time for a

condition.

It also specifies how often WebDriver will check for the condition before raising an 'ElementNotVisibleException'.

**How does an explicit wait differ from an implicit wait in Selenium?**

*Explicit* waits allow the script to pause until a specific condition is met, offering more precise control. Implicit waits, on the other hand, instruct the WebDriver to wait for a certain amount of time when trying to locate elements, applied globally throughout the script.

**Implicit Wait**

Implicit wait is used in cases where the WebDriver cannot locate an object immediately because of its unavailability. The WebDriver will wait for a specified implicit wait time and it will not try to find the element again during the specified time period.

Once the specified time limit is crossed, the webDriver will try to search the element once again for one last time. Upon success, it proceeds with the execution; upon failure, it throws exception.

It is a kind of global wait which means the wait is applicable for the entire driver. Hence, hardcoding this wait for longer time periods will hamper the execution time

**Error**

Suppose we miss syntax. Error comes when we don’t follow the syntax rules.

We cant proceed further with error.(we cant compile)

**Exception**

During execution of java program ,JVM faces the abnormal situation **based on the code declaration** is known as execution

**Cron pattern in selenium**

A Selenium script can be scheduled to run using a cron job, which is a time-based job scheduler. To run a Selenium script on a cron schedule, you'll need to first set up a Selenium environment, write your Selenium script, and then create a cron job that executes the script at the desired intervals.

Cron Jobs are used widely used in industries to schedule some important and mandatory jobs automatically like Scraping stock data, Updating the database at some intervals, generating and sending reports, etc.

**Code**

\* \* \* \* \* /path/to/your/script.py

**Where:**

\* \* \* \* \* specifies the time: \* means every minute, hour, day, month, and day of the week, respectively.

/path/to/your/script.py is the absolute path to your Selenium script.

If your script requires specific environment variables or uses a non-default shell, you may need to include those in the cron entry.

For example:

Code

\* \* \* \* \* env APP\_NAME=my-app /path/to/your/script.py

This sets the APP\_NAME environment variable before running the script.

**Example Cron Entry:**

\*/5 \* \* \* \* /path/to/your/script.py

This means the script will run every 5 minutes, starting from the first minute of each hour.

\*/2\* \* \* \* \* : This will run every 2 minutes

0 \*/8 \* \* 1-6 : This will run every 8 hours from Monday to Friday

\* \* \* jan,feb \* : Run only in the month of January and February

0 4 \* \* \* : Run at 4 am everyday

**4 Steps to configure extent reports**

1. **Add dependencies in pom.xml🡪** aventstack, relevantcodes extentreports
2. **Add Listener class in Listener package (Standard format)**
3. **Add listener in testing.xml file**
4. **Please refresh your project to see reports**

### **What are Extent Reports?**

Extent Report is an open-source library for generating test reports in automation testing. It can be easily integrated with major testing frameworks like JUnit, TestNG, etc. These reports are HTML documents that shows results as pie charts. They also allow the generation of custom logs, snapshots, and other customized details.

Once an automated test script runs successfully, testers need to generate a test execution report. While TestNG does provide a default report, they do not provide the details.

**Benefits of using Extent Reports**

They can be integrated with TestNG and Junit

They allow testers to track multiple test case runs in a single test suite

They show the time needed for test execution

They can be customized to graphically represent each step in a test.

It is an open-source library.

It provides a pictorial representation of the test results.

It can be customized as required.

It allows users to attach screenshots and logs to the test report for a detailed summary of the tests.

It can be easily configured with Jenkins, Bamboo, etc.

It is a status chart, which shows no of TC pass, failed result.

It will have multiple options to filter TC according to pass, fail result. When TC pass it will show time execution time. If fails it will show why it is faild, root cause, exceptional logs etc.

It gives overview of how much work done , work in progress, and success. It is better communication tool to share about progress of our automation project with client.

**Steps to add log4j**

Add dependencies “slf4j-api”,” log4j”

Use standard “log4j.properties” file, keep in project

Create an object of logger and give logger name

**public** **static** Logger *logger*;

@Test

**public** **void** f()

{

*logger*= *Logger*.*getLogger*("My\_New\_log");

PropertyConfigurator.*configure*("log4j.properties");

*logger*.info("Hello");

}



<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>2.0.6</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

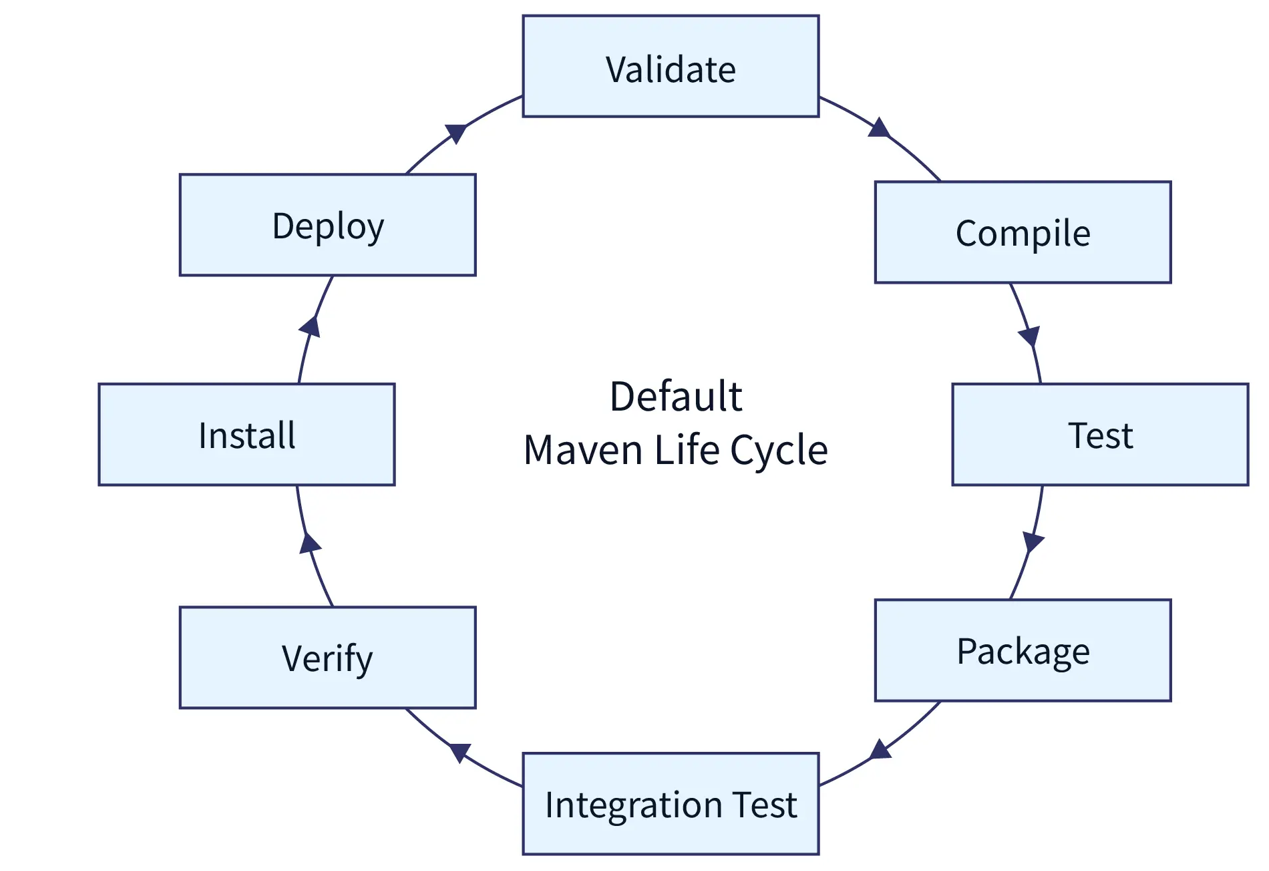
<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

</dependencies>

|  |  |
| --- | --- |
| **Level** | **Description** |
| **ALL** | All levels including custom levels. |
| **DEBUG** | Designates fine-grained informational events that are most useful to debug an application. |
| **INFO** | Designates informational messages that highlight the progress of the application at coarse-grained level. |
| **WARN** | Designates potentially harmful situations. |
| **ERROR** | Designates error events that might still allow the application to continue running. |
| **FATAL** | Designates very severe error events that will presumably lead the application to abort. |
| **OFF** | The highest possible rank and is intended to turn off logging. |
| **TRACE** | Designates finer-grained informational events than the DEBUG. |



**Sprint 4: Duration 30 days (20 Working Days )**

**Exectution-15 day Exectution-15 day**

**Execution Development**

**Sprint : 1 2 3 4**  **Manual Team**

**Mail ->** From Manual Team to Automation Team Lead for Automation TC Development.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Mail ->** Team lead **->**Team members **Sr\_NO** | **TC\_Name** | | **Status** | **Comments** | | **Priority** |
| 1  2  3  -  -  -  30 | | Pass  Fail  Completed  Hold  Pending | | | 1  2  3 | |

**How To Explain Test Automation Framework To The Interviewer**

We need to specify in and out of our Test Automation Framework such as programming **language** used, **Type of framework** used, Test Base Class (Initializing WebDriver, Implicit Waits), How we separate Element locators and tests (Page Objects, Page Factory), Utility functions file, Property files, TestNG annotations, How we parameterize tests using Excel files, How we capture error screenshots, Generating reports(Extent Reports), Emailing reports, Version Control System used and Continues Integration Tool used.

**Language:** In our Selenium Project we are using Java language. Even though Selenium supports multiple languages, we are using Java language is just because most of the automation developers have knowledge on Selenium with Java.

**Type of Framework:** In our project, we are using Data-driven Framework by using Page Object Model design pattern with Page Factory.

**POM:** As per the Page Object Model, we have maintained a class for every web page. Each web page has a separate class and that class holds the functionality and members of that web page. Separate classes for every individual test.

**Packages:** We have separate packages for Pages and Tests. All the web page related classes come under the **Pages** package and all the tests related classes come under **Tests** package.

For example, Home Page and Login Page have separate classes to store element locators. For the login test, there would be a separate class which calls the methods from the Home Page class and Login Page class.

**Test Base Class:** Test Base class (TestBase.java) deals with all the common functions used by all the pages. This class is responsible for loading the configurations from properties files, Initializing the WebDriver, Implicit Waits, Extent Reports, and also to create the object of FileInputStream which is responsible for pointing towards the file from which the data should be read.

**Utility Class (AKA Functions Class):** Utility class (TestUtil.java) stores and handles the functions (The code which is repetitive in nature such as waits, actions, capturing screenshots, accessing excels, sending email, etc.,) which can be commonly used across the entire framework. The reason behind creating a utility class is to achieve reusability. This class extends the TestBase class to inherit the properties of TestBase in TestUtil.

**Properties file:** This file (***config.properties***) stores the information that remains static throughout the framework such as browser-specific information, application URL, screenshots path, etc.

All the details which change as per the environment and authorization such as URL, Login Credentials are kept in the *config.properties* file. Keeping these details in a separate file makes it easy to maintain.

**Screenshots:** Screenshots will be captured and stored in a separate folder and also the screenshots of failed test cases will be added to the extent reports.

**Test Data:** All the historical test data will be kept in an excel sheet (*controller.xlsx*). By using *‘controller.xlsx’*, we pass test data and handle data-driven testing. We use Apache POI to handle excel sheets.

**TestNG:** Using TestNG for Assertions, Grouping, and Parallel execution.

**Maven:** Using Maven for build, execution, and dependency purpose. Integrating the TestNG dependency in the POM.xml file and running this POM.xml file using Jenkins.

**Version Control Tool:** We use Git as a repository to store our test scripts.

**Jenkins:** By using Jenkins CI (Continuous Integration) Tool, we execute test cases on a daily basis and also for nightly execution based on the schedule. Test Results will be sent to the peers using Jenkins.

**Extent Reports:** For the reporting purpose, we are using Extent Reports. It generates beautiful HTML reports. We use the extent reports for maintaining logs and also to include the screenshots of failed test cases in the Extent Report.

**\*Roles & Responsibilities of automation Tester**

1. Selenium Environment setup :

download & install eclipse, jav lang, Configrution Selenium jar files TESING, Maven project etc.

2. Inspeet elements / object: using firepath

3. Creating test case using element locators and Selenium webdriver Command

* element locater for identifying element
* Selenium webdriver for performing operation on elements.

4. Enhancing Test Cases & using prg. Feature

flow Control statements ,exception handling ,Conditional (Selenium SUPP 6 lang)

adding Comment, error handling verification etc

5. Grouping Test cases, prioritizing test cases, executing test batches & generating test reports Using testing framework

6. Data driven testing : DB testing, cross browser testing executing same functionality with multiple set of Data

7. Analysing test result & reporting defects.

8. Selecting test cases for regression testing, defect tracking

9. Regression testing on modify builds

10. Final regression testing

11. Maintainance of test automation resourses

**Steps Involved In Automation**

1] Selecting The Test Tool

2]Define The Scope Of Automation

3]Planning Design & Devl

4]Test Execution

5]Maintance

**Steps Involved In Planning Phase Of Automation**

1] Selecting the Right Automation Tool

2] Selecting Automation Framework If Any

3] List of Scope for Automation Test Environment

4] Preparing Grant Chart for Dev. And Executions

5] Identify the Test Deliverables

**The conditions where we can’t use automation in agile**

1] When agile testing always asks for changes or stack holder changes in requirements.

2] When exhaustive level of documentation is required in agile

**Primary features of good automation tool**

1] Test envoirment support & easy to use

2] Good debugging facility

3] Rebust object identification

4] Object & image testing ability

5] Testing of Database

6] Supports multiple frameworks

**List of Challenges faced in project.**

1) Domain knowledge of application

2) vast application, flows, functionalities & module integration was

3) use of dynamic Xpath

4) Handling multiple exceptions while test execution. Eg Stale Element Exception,

NullPointer Exception

5) Dependencies of test Cases on one another

6) Execution of simple text Case work but files in Suite

7) use of Before methed implemented to minimize dependencies

8) maintenance of test Data & pre conditions on multiple

9) multiple unexpected. Pop up handling.

10) Application freeze because of multiple user using single part.

11) many setting related test cases were tested an different part to minimize the effort on other test cases

**Use of OPPS concepts in Selenium**

**1. INTERFACE:**

WebDriver driver = new ChromeDriver();

In this statement WebDriver is nothing but interface in selenium.

**2. UPCASTING:**

WebDriver driver = new ChromeDriver();

above statement is nothing but UPCASTING in selenium.

**3. INHERITANCE:**

We create a Base Class in the Framework to initialize WebDriver interface, WebDriver waits, Property files etc., in the Base Class. We extend the Base Class in Tests Class. that is nothing but Inheritance in Selenium Framework.

**4. POLYMORPHISM:**

Combination of overloading and overriding is known as Polymorphism.

**5. METHOD OVERLOADING:**

We use implicit wait in Selenium. Implicit wait is an example of overloading. In Implicit wait we use different time stamps such as SECONDS, MINUTES, HOURS etc., A class having multiple methods with same name but different parameters is called Method Overloading.

eg. driver.switchTo().frame(): - String name, int index

**6. METHOD OVERRIDING:**

Declaring a method in child class which is already present in the parent class is called Method Overriding.

Examples are get and navigate methods of different drivers in Selenium.

**7. ENCAPSULATION:**

All the POM classes in a framework are an example of Encapsulation. In POM classes, we declare the data members using @FindBy and initialization of data members will be done using Constructor to utilize those in methods. Encapsulation is a mechanism of binding code and data together in a single unit. Encapsulation is the process of wrapping up code and data together in a single unit. It is used to hide the data of a class from another class. Encapsulation can be achieved when you declare all variables as private and a public method in a class to get the values of the variable.

**8. ABSTRACTION:**

In Page Object Model design pattern, we write locators (such as id, name, xpath etc.,) in a Page Class. We utilize these locators in pom class but we can’t see these locators in the tests. Literally we hide the locators from the tests. Abstraction is the methodology of hiding the implementation of internal details and showing the functionality to the users.

**Modules in Project: 10-15**

Profile

Fund

Orders

wishlist

Buy

sell

Position

Holding

Dashboard

Login

Total modules in a project: -- 15--------->15 to 20

webpages in each module:---> 20 ---> 10-30

Pom class in each module:---> 20 ---> 10-30

Total pom classes in project:15\*20: 300 -->200 to 400

1 features= 1 module = TC in each module 100 --> 50 to 200

Test cases in each module: 100 -- 50--150

Test cases in each Test class: 5 --3 to 10

Test classes in each module: 20 -- 10 to 30

Test classes in project : 15\* 20: 300 : 150 to 275

Total no of test script in project: 300\*5: 1500: 1200 to 2000 test scripts in project

Test script per day: depends on complexity of Test script (1 to 4) (avg 2)

test script per month (20 working days--> 10 days devp): avg (20 )

src/main/java- All pom classes

src/test/java- All test classes

src/main/java- library files/config files