## **PageObject Design Pattern**

The *PageObject Design Pattern* models areas of an UI as objects within test code which can also be considered as *Object Repository* for *Web UI Elements.* The functionality classes (*PageObjects*) in this design represent a logical relationship between the pages of the application. Each class is referred to as a *PageObjects* and returns other *PageObjects* to facilitate the flow between pages. Page Object class is responsible to find the WebElements of that page and also hold methods that perform operations on those WebElements.

Because PageObjects are returned, it becomes necessary to model both successful and unsuccessful events that can occur when interacted with a page.

For example, consider logging into Gmail. After entering the user details, the step either passes and navigates to the Inbox page or stays on the Login page possibly due to invalid input parameters. A pass would then return the Inbox *PageObject* whereas a fail would return the *Login PageObject.*

This means better tests, exception handling and reporting. It may sound a little confusing, but its quite a simple yet an elegant approach to write your tests. PageFactory in C# helps us to follow the same approach in an easy way.

### ***Advantages of using Page Object Pattern:***

* *Easy to Maintain*
* *Easy Readability of scripts*
* *Reduce or Eliminate duplicacy*
* *Re-usability of code*
* *Reliability*

## **PageFactory in C#**

The *PageFactory in C#* Class is an extension to the *Page Object Design Pattern.* It is an inbuilt POM concept for Selenium WebDriver but it is very optimized. It is used to initialize the elements of the Page Object or instantiate the Page Objects itself. Annotations for elements can also be created (*and recommended*) as the describing properties may not always be descriptive enough to tell one object from the other.

It is used to *Initialize Elements* of a Page class without having to use '*FindElement*' or '*FindElements*'. Annotations can be used to supply descriptive names of target objects to improve code readability.

### ***@FindBy Annotation***

As the name suggests, it helps to *find* the elements in the page using *By* strategy. *@FindBy* can accept *TagName, PartialLinkText, Name, LinkText, Id, Css, ClassName, XPath* as attributes. An alternative mechanism for locating the element or a list of elements. This allows users to quickly and easily create PageObjects.

*[FindsBy(How = How.Id, Using = "username")] private IWebElement UserName { get; set; }*

The above code will create a *PageObject* and name it as *UserName* by finding it using its *ID* locator.

his *Instantiate an Instance* of the given class. This method will attempt to instantiate the class given to it, preferably using a constructor that takes a WebDriver instance as its only argument or falling back on a no-arg constructor. An exception will be thrown if the class cannot be instantiated.

*PageFactory.InitElements(WebDriver, PageObject);*

*Parameters:*

* *WebDriver - The driver that will be used to look up the elements*
* *PageObjects - A class which will be initialised*

*Returns: An instantiated instance of the class with WebElement and List<WebElement> fields proxied*

### ***PageFactory NameSpace***

PageFactory functionality resides in *OpenQA.Selenium.Support.PageObjects.*

## **How to Implement PageFactory in Selenium Framework**

In the last chapter of the Selenium Automation Framework, we chose one Login Test Case to automate. Let's see how PageFactory in C# works in the real world:

### ***Create Login Page Objects***

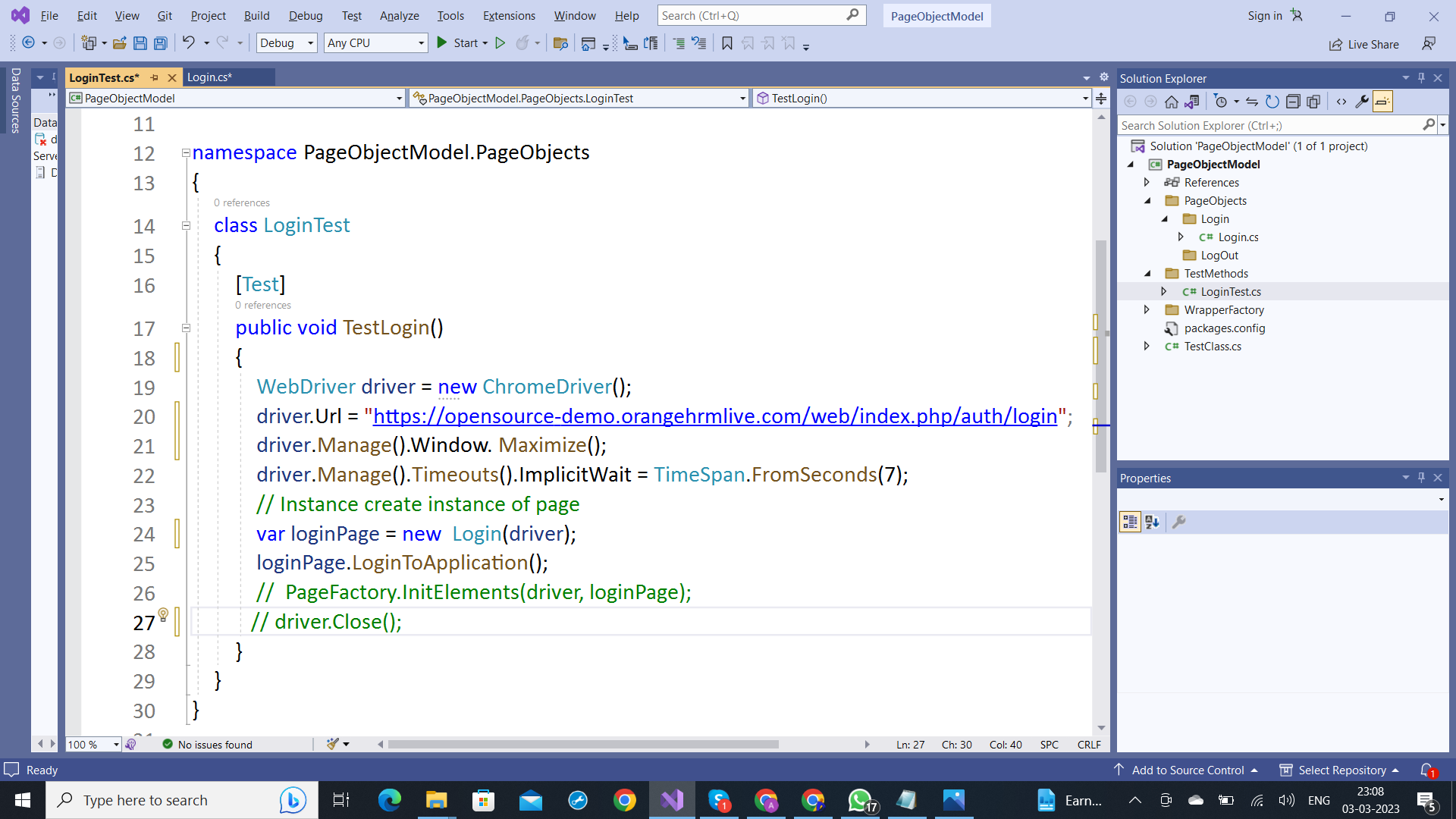
1. Create a New Folder and name it as *PageObjects*. This can be done by doing *Right Click* on *PageObjectModel* in the *Project Solution* and select *Add >> New Folder*.
2. Create a *New Class* file and refer it as *LoginPage.* This can be done by doing *Right Click* on *PageObjects* folder in the *Project Solution* and select *Add >> New Item...*

## 

*Note: HOW can be used with not just ID, it can be used with other attributes like TagName, PartialLinkText, Name, LinkText, Css, ClassName & XPath.*

### ***Write Test for LogIN Functionality***

1. *Create a New Class file in the TestMethod folder and refer it as LogInTest. Then create a Test method in it with [Test] annotation of NUnit Framework.*
2. *Write the test using LoginPage page objects.*

**

*Run the test from the Test Explorer and see how beautifully PageFactory initializes all the page objects and handle those objects to you test to carried on the execution.*

# ***PageFactory CacheLookup***

*PageFactory instantiates all the elements of the web page at the start when we Initialized any page class objects. But think of the elements which will display on the web page after some action, say Ajax action. In PageFactory, every time when we call a method on the WebElement, the driver will go and find it on the current page once again. In an AJAX-heavy application, this is what we would like to happen. So, in that case, we are good to use the PageFactory as if the element appears later, the pagefactory will still look for the element on its turn.*

*But in the case of regular application where elements are stable and when we know that the element is always going to be there and won't change. It would be handy if we could 'cache' the element once we'd looked it up and save some time of execution by commanding PageFactory to not search the WebElements on the page again.*

## ***PageFactory CacheLookup***

*PageFactory also provide CacheLookup attribute to cache the WebElements. This attribute helps scripts to instruct the InitElements method to cache the element once it's located. In other words, any attribute marked [CacheLookup] will not be searched over and over again – this is especially useful for elements that are always going to be there (not always true for AJAX apps). It means the elements of the page will be cached once searched. All elements used in the HomePage & LoginPage class are static and are always present. So it is better to cache objects and save execution time of the test run.*

### ***How to use PageFactory CacheLookup***

*If we know that element is always present on the page, it is best to use the following declaration:*

*[FindsBy(How = How.Id, Using = "account")][CacheLookup] public IWebElement MyAccount { get; set; }*

### ***PageFactory CacheLookup NameSpace***

*PageFactory functionality resides in OpenQA.Selenium.Support.PageObjects.CacheLookupAttribute.*

## ***How to Implement PageFactory CacheLookup in Framework***

*We will be using the same LogIn Scenario for this. Moving further as well we will keep on enhancing this scenario and soon will reach to the end goal of our Selenium Automation Framework.*

### ***Change Page Objects***

*All we need to do here is to add PageFactory CacheLookup attribute for every element.*

### ***Notes***

1. *If you use the PageFactory, you can assume that the fields are initialised. If you don't use the PageFactory, then NullPointerExceptions will be thrown if you make the assumption that the fields are already initialised.*
2. *List<WebElement> fields are decorated if and only if they have @FindBy or @FindBys annotation. Default search strategy "by id or name" that works for WebElement fields is hardly suitable for lists because it is rare to have several elements with the same id or name on a page.*
3. *WebElements are evaluated lazily. That is, if you never use a WebElement field in a PageObject, there will never be a call to "findElement" for it.*

### ***Advantages***

1. *When the PageFactory is initialised the proxies are configured, but the WebElements are not found at that point (so you won't get a NoSuchElementException).*
2. *Every time you use a WebElement it will go and find it again so you shouldn't see StaleElementException's.*
3. *But when you use the @CacheLookup annotation, which is losing you the second benefit as it will find the element once and then keep a reference to it, you are now far more likely to see StaleElementExceptions.*

### ***Differences between C# and Java Implementation***

1. *In C# the PageFactory.InitElements returns void, whereas in Java it returns the Page Objects.*
2. *The PageFactory implementation for C# only searches for elements using the ID. It does not locate the elements using the NAME property. Whereas in Java it also tries to find the element with Name property, if it is not able to find it with ID.*
3. *The Java implementation can locate the element even without the FindsBy attribute. This isn’t the case for C#.*

*As we have learned in the last chapter of Page Object model is an object design pattern in Selenium, where web pages are represented as classes. The various elements on the page are defined as variables on the class. All possible user interactions can then be implemented as methods on the class. Since well-named methods in classes are easy to read, this works as an elegant way to implement test routines that are both readable and easier to maintain or update in the future.*

*In order to support Page Object model, we used* [*Page Factory*](https://toolsqa.com/selenium-webdriver/c-sharp/pagefactory-in-c/)*. Page Factory is an extension to Page Object and can be used in various ways. As we know that web page classes or Page Objects containing web elements need to be initialized using Page Factory before the web element variables can be used. This can be done simply through the use of initElements function on PageFactory. Still, this is not enough in the PageFactory, it can be utilized much efficiently by Optimizing Page Object Model :*

* *Initialize Elements within the Constructor*
* *Binding methods within the PageObject class*

## ***Initialize Elements within the Constructor***

*In the first chapter of Selenium* [*PageObjects and PageFactory*](https://toolsqa.com/selenium-webdriver/c-sharp/pagefactory-in-c/) *, we did this to initialize elements of the page:*

*var loginPage = new LoginPage();*

*PageFactory.InitElements(driver, loginPage);*

*But the duty of the Framework is to minimize the code where ever it can. Let say if the application has ten pages to automate. To do the task, it is required to create the object of ten pages first and then initialize ten pages to use their objects. This will make the test very lengthy and filthy. It is a good idea to abstract this task from the test. By placing the Init statement in the Page Class Constructor, half of the code from the test can be reduced.*

*public LoginPage(IWebDriver driver){*

*this.driver = driver;*

*PageFactory.InitElements(driver, this);*

*}*

*Every time the object for the LogIN page is created, it will first go into the constructor and initialize all the object of the page. This is a nice way of Optimizing Page Object Model. To read more about* [*Constructors*](https://toolsqa.com/java/class-constructors/)*, please refer the article in java.*

## ***Binding Methods within the Page Object Model Class***

*The second way to Optimizing Page Object Model or improving the test code is to wrap re-usable actions in the PageObject class itself. Now think of the reusable function in the example, that we have used in this Selenium Framework C# series. The answer is log into the application. As of now we just have one username and password, but what if there are more than thirty users for the application. Then the code for log into the application will be called again and again in every test script. Isn't a nice idea to wrap that action into a function and use it as many time as required.*

*Doing this is not a difficult task, as above we already initialized the elements of the page within the class constructor. Now the elements are ready to use in the PageObject class itself.*

*UserName.SendKeys("Admin");*

*Password.SendKeys("Test@123");*

*Submit.Submit();*

*But the above code has to be bind in the Function or a Method. We will be using the same LogIn Scenario for this. Let's just see how the Page classes and the test class will look like after binding login method within the LoginPage class.*

*LoginPage PageObject Class*

*using OpenQA.Selenium;*

*using OpenQA.Selenium.Support.PageObjects;*

*namespace OnlineStore.PageObjects*

*{*

*public class LoginPage*

*{*

*private IWebDriver driver;*

*[FindsBy(How = How.Id, Using = "log")]*

*[CacheLookup]*

*public IWebElement UserName { get; set; }*

*[FindsBy(How = How.Id, Using = "pwd")]*

*[CacheLookup]*

*public IWebElement Password { get; set; }*

*[FindsBy(How = How.Id, Using = "login")]*

*[CacheLookup]*

*public IWebElement Submit { get; set; }*

*public LoginPage(IWebDriver driver)*

*{*

*this.driver = driver;*

*PageFactory.InitElements(driver, this);*

*}*

*public void LoginToApplication()*

*{*

*UserName.SendKeys("TestUser\_1");*

*Password.SendKeys("Test@123");*

*Submit.Submit();*

*}*

*namespace PageObjectModel.PageObjects*

*{*

*class LoginTest*

*{*

*[Test]*

*public void TestLogin()*

*{*

*WebDriver driver = new ChromeDriver();*

*driver.Url = "https://opensource-demo.orangehrmlive.com/web/index.php/auth/login";*

*driver.Manage().Window. Maximize();*

*driver.Manage().Timeouts().ImplicitWait = TimeSpan.FromSeconds(7);*

*// Instance create instance of page*

*var loginPage = new Login(driver);*

*loginPage.LoginToApplication();*

*// PageFactory.InitElements(driver, loginPage);*

*// driver.Close();*

*}*

*}*

*}*

### ***Conclusion***

*Page Object and Page Factory make it easy to model web pages in Selenium and test them automatically and make the life of both developers and QAs much more simpler. When done right, these Page Object classes can be reused across your entire test suite and to give yourself the opportunity to implement automated Selenium tests for your projects early on, without compromising agile development. By abstracting away user interactions in your page object models and keeping your test routines light and simple, you can adapt your test suite to changing requirements with little effort.*

# ***Browser Factory or WebDriver Factory***

## ***Factory Design Pattern***

*Factory Design Pattern is one of the most useful patterns. You can see the usage of this pattern everywhere. One such usage of Factory Design Principle in Frameworks is in the PageFactory. As we all know PageFactory is the brilliant page object pattern provided by Selenium. A typical usage of Selenium PageFactory is something like this.*

*PageFactory.InitElements(driver, PageObjectClass);*

*Here you will notice that we are not directly creating a PageObject by using the new keyword. We are calling another class called PageFactory and on that class, we have a static method called InitElements(). This InitElements() method is responsible for initializing all the page objects. PageFactory is based on the Factory pattern design principle, it is basically a factory that stores and provides different page objects for usage.*

*A Factory pattern is used to encapsulate the complexity of creating an object. Below are the two important ideas behind Factory pattern:*

* *Hide the logic of initializing an object inside the factory.*
* *Refer to the object using a common interface instead of the concrete class.*
* *Maintain/Cache the newly created object’s life time. This is not main responsibility of a factory pattern.*

*Let's understand how a Factory Pattern is beneficial while designing a Selenium Automation Framework.*

## ***Browser Factory or WebDriver Factory***

*From our experience with frameworks we know that maintaining and passing around a WebDriver object across different tests is a delicate process. Also, the complexity increases when we have to maintain only one instance of a WebDriver through out the test run. To overcome the problem on instantiation of WebDriver and maintaining the instance of browser we can use create a small class called Browser Factory or WebDriver Factory*

*IWebDriver driver = BrowserFactory.InitBrowser("Firefox");*

*or it can be like below, your choice of names :)*

*IWebDriver driver = WebDriverFactory.InitDriver("Firefox");*

*There has to be a InitBrowser() or InitDriver() method inside the BrowserFactory or WebDriverFactory class, which will initialize the requested browser for the test. The above code do the same. By using the InitBrowser() method of the BrowserFactory class, it is requesting the class to return the Firefox browser to the test.*

*Now, how would you handle this in the Browser/Driver factory. It is completely up to you. You can use IF ELSE condition on the browser parameter and return browser. But I would like to make the best use of the Advance C# feature, so I would use Dictionary object to do the same.*

*Dictionary<TKey, TValue>is a generic collection included in the System.Collection.Generics namespace. TKey denotes the type of key and TValue is the type of TValue. A Dictionary can be initialized with a variable of IDictionary<Tkey, TValue> interface as well as with a Dictionary<TKey, Tvalue> class. It is recommended to program to the interface rather than to the class. So, use IDictionary<TKey, TValue> type variable to initialize a dictionary object.*

*In the below example, we have specified types of key and value while declaring a dictionary object. A string is a type of key and IWebDriver is a type of value that will be stored into a dictionary object named Drivers. You can use any valid C# data type for keys and values.*

*private static readonly Dictionary<string, IWebDriver> Drivers = new Dictionary<string, IWebDriver>();*

*Note: Dictionary cannot include duplicate or null keys, whereas values can be duplicated or set as null. Keys must be unique otherwise it will throw a runtime exception.*

### ***Dictionary Object Methods***

* *Count : Gets the total number of elements exists in the Dictionary<TKey,TValue>*
* *IsReadOnly : Returns a boolean indicating whether the Dictionary<TKey,TValue> is read-only*
* *Item : Gets or sets the element with the specified key in the Dictionary<TKey,TValue>*
* *Keys: Returns collection of keys of Dictionary<TKey,TValue>*
* *Values : Returns collection of values in Dictionary<TKey,TValue>*

## ***Browser Factory or WebDriver Factory Implementation in Selenium C#***

### ***Create Browser Factory***

1. *Create a new folder and name it as WrapperFactory.*
2. *Create a new C# class in the WrapperFactory folder, name it as BrowserFactory or WebDriverFactory*
3. *Write the implementation of the BrowserFactory class.*

*using System;*

*using System.Collections.Generic;*

*using System.Linq;*

*using System.Text;*

*using System.Threading.Tasks;*

*using OpenQA.Selenium;*

*using OpenQA.Selenium.Chrome;*

*using OpenQA.Selenium.Firefox;*

*using OpenQA.Selenium.IE;*

*namespace PageObjectModel.WrapperFactory*

*{*

*class BrowserFactory*

*{*

*private static readonly IDictionary<string, WebDriver> Drivers = new Dictionary<string, WebDriver>();*

*private static WebDriver driver;*

*public static WebDriver Driver*

*{*

*get*

*{*

*return driver;*

*}*

*private set*

*{*

*driver = value;*

*}*

*}*

*public static void InitBrowser(string browserName)*

*{*

*switch (browserName)*

*{*

*case "Firefox":*

*if (Driver == null)*

*{*

*driver = new FirefoxDriver(@"D:\\Trupti\\Seleniumjars\\geckodriver-v0.32.2-win32\\geckodriver.exe");*

*Drivers.Add("Firefox", Driver);*

*}*

*break;*

*case "Chrome":*

*if (Driver == null)*

*{*

*Driver = new ChromeDriver(@"D:\\Trupti\\Seleniumjars\\chromedriver\_win32\\chromedriver.exe");*

*Drivers.Add("Chrome", Driver);*

*}*

*break;*

*}*

*}*

*public static void LoadApplication(string url)*

*{*

*Driver.Url = url;*

*}*

*public static void CloseAllDrivers()*

*{*

*foreach (var key in Drivers.Keys)*

*{*

*Drivers[key].Close();*

*Drivers[key].Quit();*

*}*

*}*

*}*

*}*

*Key points in the above implementation:*

1. *A Dictionary Object of Browser name to Browser is stored as a private member. So nobody can directly access it.*
2. *In the InitBrowser() method a browser is created only once, that will be when the very first call to get the browser of a particular type will come. After that only the existing browser is returned. This means that the browser will be cached through the session.*
3. *Complexity of browser initialization and storage is abstracted out from tests. This means that tests will no longer have to worry about where the browser came from or where it is stored. All they have to do is call the BrowserFactory.InitBrowser(<BrowserName>) method.*
4. *This will enable tests to get any type of drivers without worrying of creating unnecessary driver instances.*
5. *Any of the Class now can get the instance of WebDriver by using BrowserFactory.Driver.*

*Cons of this approach*

* *As you can see that this is not thread safe, it means that it will not be able to work on parallel test environments. A few modifications in this class and it will become capable of handling parallel requests. We will learn about it in coming articles on design principles in test frameworks.*

*The Main Idea behind the BrowserFactory in the Framework*

* *We should not be using driver object directly in the test. Means there should not be any statement where we use driver and perform any action on it. At present we have one statement in the test to open the Url using driver object. Now instead of that we will use LoadApplication method to open the Url.*

*LoginPage will look like as below:*

*using System;*

*using System.Collections.Generic;*

*using System.Linq;*

*using System.Text;*

*using System.Threading.Tasks;*

*using OpenQA.Selenium;*

*using SeleniumExtras.PageObjects;*

*namespace PageObjectModel.PageObjects*

*{*

*public class LoginPage*

*{*

*//public IWebDriver driver;*

*private WebDriver driver;*

*[FindsBy(How = How.XPath, Using = "//input[@name='username']")][CacheLookup]*

*// public IWebElement UserName { get; set; }*

*private IWebElement UserName { get; set; }*

*[FindsBy(How = How.XPath, Using = "//input[@name='password']")][CacheLookup]*

*// public IWebElement Password { get; set; }*

*private IWebElement Password { get; set; }*

*[FindsBy(How = How.XPath, Using = "//button[@type='submit']")]*

*[CacheLookup]*

*// public IWebElement Submit { get; set; }*

*private IWebElement Submit { get; set; }*

*public LoginPage(IWebDriver driver) {*

*this.driver = driver as WebDriver;*

*PageFactory.InitElements(driver, this);*

*}*

*public void LoginToApplication() {*

*UserName.SendKeys("Admin");*

*Password.SendKeys("admin123");*

*Submit.Submit();*

*}*

*}*

*}*

*Look at the LoginTest1.cs*

*using NUnit.Framework;*

*using PageObjectModel.PageObjects;*

*using System.Configuration;*

*using System.Threading;*

*namespace PageObjectModel.WrapperFactory*

*{*

*[TestFixture]*

*class LoginTest1*

*{*

*[Test]*

*public void Test()*

*{*

*BrowserFactory.InitBrowser("Chrome");*

*BrowserFactory.LoadApplication("https://opensource-demo.orangehrmlive.com/web/index.php/auth/login");*

*// BrowserFactory.LoadApplication(ConfigurationManager.AppSettings["URL"]);*

*Thread.Sleep(3000);*

*var loginPage = new LoginPage(BrowserFactory.Driver);// Create instance of OR as an argument BrowserFactory Class and driver;*

*loginPage.LoginToApplication();*

*BrowserFactory.CloseAllDrivers();*

*}*

*}*

*}*