Stale Element Reference Exception

What is StaleElementReferenceException?

- Stale means old, decayed, and no longer fresh. Stale Element means an old element or no longer available element.

When do we get this exception?

- After finding the element and before performing the action on that element, if the page is refreshed then we get the stale element reference exception.

  Example: Assume there is an element that is found on a web page referenced as a WebElement in WebDriver. If the DOM changes then the WebElement goes stale. If we try to interact with an element which is staled then the **StaleElementReferenceException** is thrown.

Program to raise the stale element reference exception?

```python
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome()
driver.get('https://demo.actitime.com/')
driver.implicitly_wait(30)

username = driver.find_element_by_css_selector("input#username")
password = driver.find_element_by_css_selector("input.textField.pwdfield")
loginButton = driver.find_element_by_id("loginButton")

#Entering the wrong user name and wrong password
username.send_keys("admin")
password.send_keys("admin")
loginButton.click()
sleep(5)

#Entering the correct usernam and password
username.send_keys("admin")
password.send_keys("manager")
loginButton.click()

driver.quit()
```

If you run the above script, you will get the following exception.

```
selenium.common.exceptions.StaleElementReferenceException: Message: stale element reference: element is not attached to the page document
  (Session info: chrome=70.0.3538.67)
  (Driver info: chromedriver=2.42.591088 (7b2b2dca23cca0862f674758c9a3933e685c27d5),platform=Windows NT 10.0.17134 x86_64)
```

Pramod K S

Common Causes

A stale element reference exception is thrown in one of two cases, the first being more common than the second:

- The element has been deleted entirely.
- The element is no longer attached to the DOM.

The Element has been deleted

- The most frequent cause of this is that page that the element was part of has been refreshed, or
- The user has navigated away to another page.

Note: We face this stale element reference exception when the element we are interacting is destroyed and then recreated again. When this happens the reference of the element in the DOM becomes stale. Hence we are not able to get the reference to the element.

The Element is not attached to the DOM

A common technique used for simulating a tabbed UI in a web app is to prepare DIVs for each tab, but only attach one at a time, storing the rest in variables. In this case, it's entirely possible that your code might have a reference to an element that is no longer attached to the DOM (that is, that has an ancestor which is "document.documentElement").

How to solve this issue! Or how do you handle stale element reference exception

Solution 1: Page object model - (POM).

Data Encapsulation

- **Encapsulation** is a mechanism of wrapping the data and code acting on the data together as a single unit.
- Declare the variables of a class as private. Provide setter and getter methods to modify and view the variables values.
- Hiding internal state and requiring all interaction to be performed through on objects methods is known as data encapsulation.

Here, data refers to variable and for any given variable we perform 3 steps.

1. Declaration.
2. Initialization.
3. Utilization.

Pramod K S

Example:

```python
class A:
    #Declaration
    __i = ""

    def __init__(self, j):
        #Initialization
        self.__i = j

    def print_value(self):
        print(self.__i)
```

Encapsulation in **Selenium**:

What is the advantage?

- **Creating the object repository:** To avoid identifying the same element multiple times and to provide the protection for the web elements.
- **Rising Flexibility:** Based on our requirement, we can only read or write only the variables of the class.
- **Reusability:** improvisation with improvisation and new requirements is improved easily with encapsulation.
- **The test code is simple:** it is easy to check for unit testing.

Sample scripts to implement the encapsulation in selenium.

Page Object Model

What is POM?

- Page Object model is an object design pattern in Python.
- Where web **pages** are represented as **classes**, and the various **elements** on the page are defined as **variables** of the class. All possible user **interactions** can then be implemented as **functions** on the class.

Pramod K S

What are the uses of page object model?

- Using POM we can develop and test web pages.
- It is used to handle the stale element reference exception.
- Code reusability.
- Code maintainability.
- Code Readability.
- Object Repository.

Why POM?

Starting an UI Automation in Selenium WebDriver is NOT a tough task. You just need to find elements, perform operations on it.

Consider this simple script to login into a website,

```python
from selenium import webdriver
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as ec
'''
   @author PriyaPramod
   This test case will login in http://demo.actitime.com
   Login to application
   Verify the home page by getting the title of the Home Page
'''
from selenium.common.exceptions import TimeoutException

driver = webdriver.Chrome(executable_path="C:/Users/PriyaPramod/Desktop/Selenium/Driver/chromedriver")
driver.get('https://demo.actitime.com/')
driver.implicitly_wait(30)

username = driver.find_element_by_css_selector("input#username")
password = driver.find_element_by_css_selector("input.textField.pwdfield")
loginButton = driver.find_element_by_id("loginButton")

username.send_keys("admin")
password.send_keys("manager")
loginButton.click()


isTrue = False
wait = WebDriverWait(driver, 30)
try:
    isTrue = wait.until(ec.title_is("actiTIME - Enter Time-Track"), "Verifying the title of home page")
except TimeoutException:
    isTrue = False
    print("Time out Exception")

if isTrue:
    print("Login Successful: Home page is displayed")
else:
    print("Login Failed: Home page is not displaying")

driver.quit()
```

As you can observe, all we are doing is finding elements and filling values for those elements.

This is a small script. Script maintenance looks easy. But with time test suite will grow. As you add more and more lines to your code, things become tough.

The chief problem with script maintenance is that if 10 different scripts are using the same page element, with any change in that element, you need to change all 10 scripts. This is time consuming and error prone.

Pramod K S

A better approach to script maintenance is to create a separate class file which would find web elements, fill them or verify them. This class can be reused in all the scripts using that element. In future, if there is a change in the web element, we need to make the change in just 1 class file and not 10 different scripts.

- This approach is called **Page Object Model (POM)**.
- It helps make the code **more readable, maintainable**, and **reusable.**
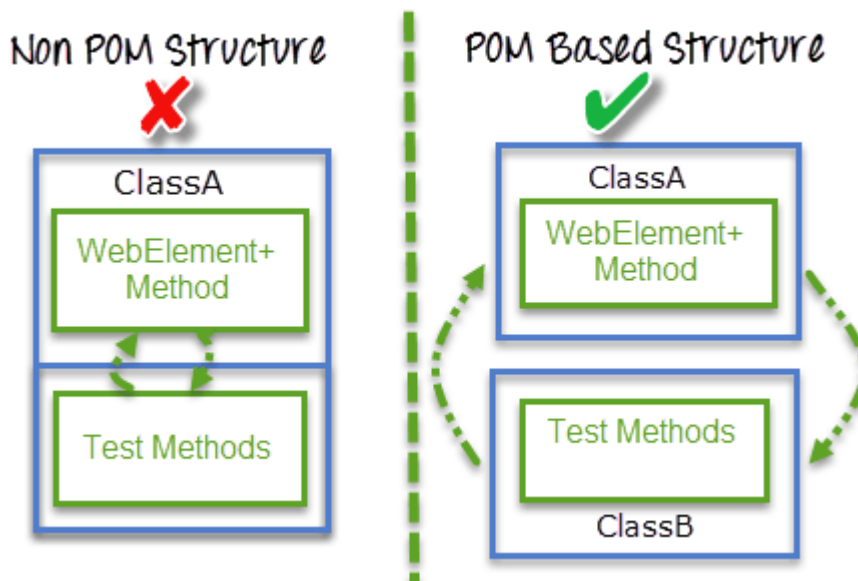
Note:

- Creating Selenium test cases can result in an unmaintainable project. One of the reasons is that too many duplicated codes is used. Duplicated code could be caused by duplicated functionality and this will result in duplicated usage of locators. The disadvantage of duplicated code is that the project is less maintainable. If some locator will change, you have to walk through the whole test code to adjust locators where necessary. By using the page object model, we can make non-brittle test code and reduce or eliminate duplicate test code. Beside of that it improves the readability and allows us to create interactive documentation. Last but not least, we can create tests with less keystroke. An implementation of the page object model can be achieved by separating the abstraction of the test object and the test scripts.

- Page Object Model Framework has now a days become very popular test automation framework in the industry and many companies are using it because of its easy test maintenance and reduces the duplication of code.

- The main advantage of Page Object Model is that if the UI changes for any page, it don't require us to change any tests, we just need to change only the code within the

page objects (Only at one place). Many other tools which are using selenium, are following the page object model.

Advantages of POM

- **Code reusability** – We could achieve code reusability by writing the code once and use it in different tests.
- **Code maintainability** – There is a clean separation between test code and page specific code such as locators and layout which becomes very easy to maintain code. Code changes only on Page Object Classes when a UI change occurs. It enhances test maintenance and reduces code duplication.
- **Object Repository –** Each page will be defined as a python class. All the fields in the page will be defined in an interface as members. The class will then implement the interface.
- **Readability –** Improves readability due to clean separation between test code and page specific code
- **Low maintenance**: Any User Interface changes can swiftly be implemented into the interface as well as class.
- **Programmer Friendly**: Robust and more readable. The Object-oriented approach makes the framework programmer friendly.
- **Low Redundancy**: Helps reduce duplication of code. If the architecture is correctly and sufficiently defined, the POM gets more done in less code.
- **Efficient & Scalable**: Faster than other keyword-driven/data-driven approaches where Excel sheets are to be read/written.

Disadvantages
1. **High Setup Time & Effort**: Initial effort investment in development of Automation Framework is high. This is the biggest weight of POM in case of web applications with hundreds/thousands of pages. It is highly suggested that if this model is decided to be implemented, then it should be done parallel to development of the application. Refer V-Model for Software Development Life Cycle.
2. **Skilled labour**: Testers not technically sound or aware of programming best practices are a nightmare in this case. Perhaps this is the biggest mistake to make, employing unskilled labour in hopes of training them during implementation.
3. **Specific**: Not a generic model. Automation Framework developed using POM approach is specific to the application. Unlike keyword-driven/data-driven frameworks, it is not a generic framework.

Irrespective of the disadvantages, POM is perhaps the most efficient and highly recommended approach towards any web application. As the framework matures it is perhaps easier to modify it into a hybrid framework from a POM approach than from other keyword/data driven approaches.

How to implement POM?

Rules to Implement POM classes:

1. Develop a class for every web page in the application.

   ➢ For example: If we have a 10 web pages in an application, then we need to develop 10 classes.

2. Web Elements should be identified and assigned to the variable and all the variables should be declared as private.

3. Develop functions to perform the action on the web elements.

4. All the page classes should be initialized during run time.

Three ways we can implement the POM.

1. Simple POM
2. Optimized POM
3. Advance POM

Simple POM

- It's the basic structure of Page object model (POM) where all Web Elements of the Application under Test (**AUT)** and the method that operate on these Web Elements are maintained inside a class file.
- A task like **verification** should be **separate** as part of Test methods.

Create a project by naming as 'simple_pom'. And create a folder structure as below.
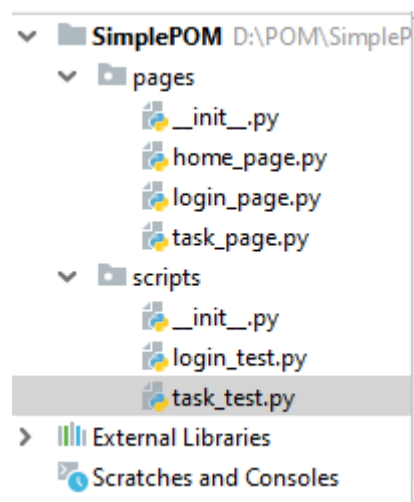
Page Classes:

- Creating the classes for every web page in the application.
- Identifying the required web elements.
- Developing the functions to perform the action using the web elements.

```python
from selenium.webdriver.common.by import By


class LoginPage:

    def __init__(self, driver):
        self.driver = driver
        self.driver.set_page_load_timeout(15)

    __username_tb = (By.ID, "username")
    __password_tb = (By.NAME, "pwd")
    __login_button = (By.ID, "loginButton")

    def set_user_name(self, username):
        self.driver.find_element(*self.__username_tb).send_keys(username)

    def set_password(self, password):
        self.driver.find_element(*self.__password_tb).send_keys(password)

    def click_login_button(self):
        self.driver.find_element(*self.__login_button).click()
```

```python
from selenium.webdriver.common.by import By


class HomePage:

    def __init__(self, driver):
        self.driver = driver
        self.driver.set_page_load_timeout(15)

    __logout_link = (By.ID, "logoutLink")
    __task_tab = (By.XPATH, "//div[text()='TASKS']")

    def click_on_logout_link(self):
        self.driver.find_element(*self.__logout_link).click()

    def click_on_task_tab(self):
        self.driver.find_element(*self.__task_tab).click()




from time import sleep

from selenium.webdriver.common.by import By


class TaskPage:

    __add_new_button = (By.XPATH, "//div[contains(@class,'title ellipsis')]")
    __new_task_button = (By.XPATH, "//div[contains(@class,'item createNewTasks')]")
    __close_button = (By.XPATH, "//div[@id='closeCreateTasksPopupButton']")

    def __init__(self, driver):
        self.driver = driver
        driver.set_page_load_timeout(15)

    def click_on_add_new_button(self):
        self.driver.find_element(*self.__add_new_button).click()

    def click_on_new_task(self):
        self.driver.find_element(*self.__new_task_button).click()

    def close_pop_up(self):
        sleep(5)
        self.driver.find_element(*self.__close_button).click()
```

Note: **set_page_load_timeout** function sets the amount of time to wait for a page load to complete.

Pramod K S

Test Scripts:
- A test script in software testing is a set of instructions that will be performed on the system under test to test the system functions as expected.
- A test scripts will be created for every manual test cases.

login_test.py ×

```python
from selenium import webdriver
from pages.home_page import HomePage
from pages.login_page import LoginPage

driver = webdriver.Chrome()
driver.implicitly_wait(30)
driver.maximize_window()
driver.get("https://demo.actitime.com")

login = LoginPage(driver)
login.set_user_name("admin")
login.set_password("manager")
login.click_login_button()

home = HomePage(driver)
home.click_on_logout_link()
driver.quit()
```

Pramod K S

task_test.py ×

```python
from pages.task_page import TaskPage

driver = webdriver.Chrome()
driver.implicitly_wait(30)
driver.maximize_window()
driver.get("https://demo.actitime.com")

login = LoginPage(driver)
login.set_user_name("admin")
login.set_password("manager")
login.click_login_button()

home = HomePage(driver)
home.click_on_task_tab()

task = TaskPage(driver)
task.click_on_add_new_button()
task.click_on_new_task()
task.close_pop_up()
home.click_on_logout_link()

driver.quit()
```

Optimized POM

- **Optimized POM** is an extension of basic page object model. We extend the POM to improve **code quality** and **efficiency**. A program may be optimized so that it becomes a **smaller size**, **consumes less memory, executes** more **rapidly**, or performs fewer input/output operations.

What is optimization?

- Optimization is an act, process, or methodology of making something (as a design, system, or decision) as fully perfect, functional, or effective as possible; specifically.

Now, what is code optimization?

- Code optimization is any method of code modification to improve code quality and efficiency.

- Code optimization in an on-going process which includes cutting down the unused resources, lines of codes, and functions without affecting the core functioning of the code.

Advantages of code optimization

1. **Managing the functions inline:**

Managing smaller size functions by the inline approach is a widely used practise in code optimization. For the complex functions, it's best to create a separate file but you can include smaller functions to save resources.

2. **Increasing the readability:**

Another phase of this process is improving the code readability to make the code more usable. Code readability is the process of making the code easier for everyone to understand and reuse it if needed.

3. **Avoiding complex nested structures:**

It is recommended to simplify the complex nested loop structure to improve the execution time and performance.

4. **Memory utilization:**

Memory utilization involves carefully selecting the variables, defining the scope and managing their resources to save precious memory and increase the performance of the written code.

5. **A program may be optimized so that it:**
   Becomes a smaller size
   Consumes less memory
   Executes more rapidly
   Performs fewer input/output operations

Pramod K S

How to optimize the page object model.

- Identify all the repeated and duplicate code in all the page classes.

  Example: In all the page classes, we are creating the parameterised constructor, creating the class variable and using the waiting method in all the classes.

```python
def __init__(self, driver):
    self.driver = driver
    self.driver.set_page_load_timeout(10)
```

- Identify all the common actions/functionalities in all the page classes.

  Example: find_element, click, send_keys etc.

- Identify the common elements which will be present in all the pages in application.

  Example: In ACTi-time application, there are many web elements which are common in all the web pages. You navigate to any pages, same web elements will be present in the web pages.
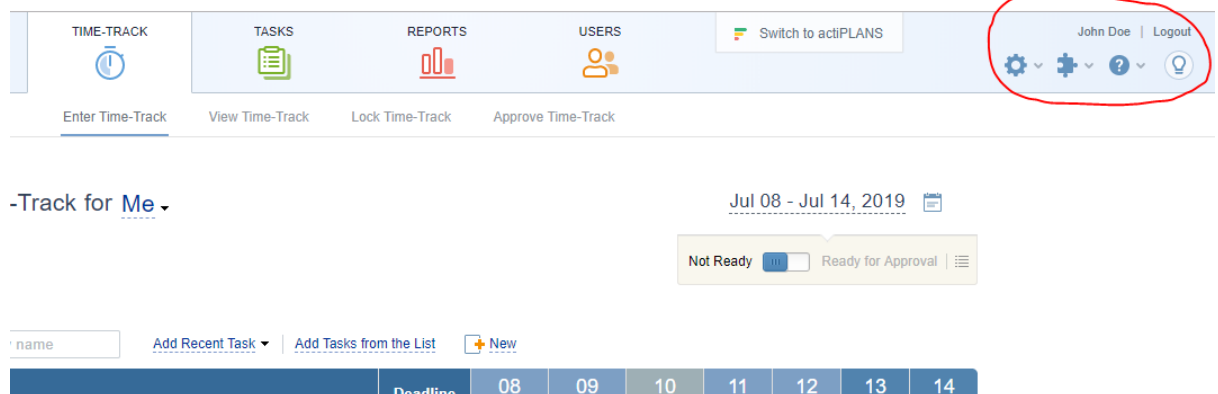
In the above image, no matter to which page you will navigate. The highlighted web elements will be present in all the web pages.

IMP

So, instead of using the duplicate code, common actions and common web elements in all the page classes.

Create a new class "**BASEPAGE**", and keep all the common behaviours, common web elements and anything which is common for all the pages should be developed in the base page class.

Pramod K S

To make all these actions to be available to the page classes, make sure, base page is inheriting to all the page classes.

By doing this, we can avoid the redundant/duplicate code, reduce the number of lines of code, code quality improves, performance of our project will improve and it will take less memory.

Note:

- In this case, we can use the "Inheritance" & "Constructor chaining" concepts.

Create the project "optimized_pom". And create the folder structure like below.

```
optimized_pom  D:\Weekend_Selenium\
    pages
        __init__.py
        base_page.py
        home_page.py
        login_page.py
        task_page.py
    test_scripts
        __init__.py
        base_test.py
        create_task_test.py
    External Libraries
    Scratches and Consoles
```
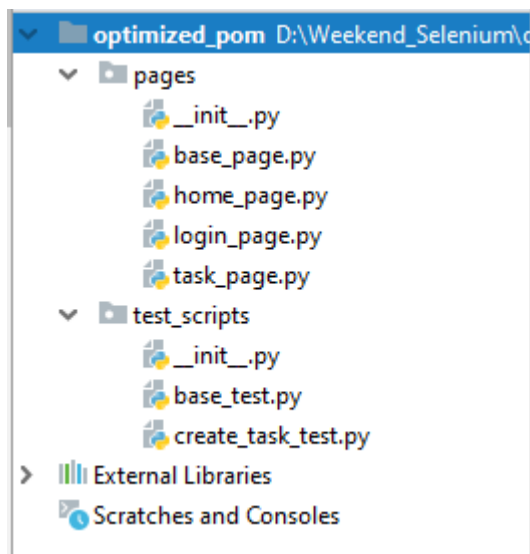
Base_page.py

```python
from selenium.common.exceptions import NoSuchElementException, TimeoutException
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as ec


class BasePage:

    def __init__(self, driver):
        self.driver = driver
        self.driver.set_page_load_timeout(15)
        self.wait = WebDriverWait(self.driver, 15)

    def find_element(self, locator, value):
        try:
            web_element = self.driver.find_element(locator, value)
            return web_element
        except NoSuchElementException:
            print("Web element is not available in the DOM")
```

Pramod K S

```
20    def wait_for_element_to_be_visible(self, web_element):
21        try:
22            self.wait.until(ec.visibility_of(web_element))
23        except TimeoutException:
24            print("Web element is not visible in the Application UI")
25
```

Login_page.py

```
login_page.py
1    from selenium.webdriver.common.by import By
2    from pages.base_page import BasePage
3
4
5    class LoginPage(BasePage):
6
7        def __username_tb(self): return self.find_element(By.ID, "username")
8        def __password_tb(self): return self.find_element(By.NAME, "pwd")
9        def __login_button(self): return self.find_element(By.ID, "loginButton")
10
11       def enter_user_name(self, user):
12           self.__username_tb().send_keys(user)
13
14       def enter_password(self, password):
15           self.__password_tb().send_keys(password)
16
17       def click_on_login_button(self):
18           self.__login_button().click()
19
```

Home_page.py

```
home_page.py
1    from selenium.webdriver.common.by import By
2    from pages.base_page import BasePage
3
4
5    class HomePage(BasePage):
6
7        def __tabs(self, tab_name): return self.find_element(By.XPATH,
8                                    "//div[text()='"+tab_name+"']/..")
9        def __log_out_link(self): return self.find_element(By.ID, "logoutLink")
10
11       def click_on_tab(self, tab_name):
12           self.__tabs(tab_name).click()
13
14       def click_on_log_out(self):
15           self.__log_out_link().click()
```

Task_page.py

Pramod K S

```python
from selenium.webdriver.common.by import By
from pages.base_page import BasePage


class TaskPage(BasePage):

    def __add_new_button(self): return self.find_element(By.XPATH,
                                        "//div[text()='Add New']")
    def __new_task_button(self): return self.find_element(By.XPATH,
                                        "//div[text()='+ New Tasks']")
    def __cancel_button(self): return self.find_element(By.ID,
                                        "closeCreateTasksPopupButton")

    def click_on_add_new_button(self):
        ele = self.__add_new_button()
        self.wait_for_element_to_be_visible(ele)
        ele.click()


    def click_on_add_new_task_button(self):
        self.__new_task_button().click()

    def click_on_cancel_button(self):
        ele = self.__cancel_button()
        self.wait_for_element_to_be_visible(ele)
        ele.click()
```

- Below files should be created under "test_scripts" package.

Base_test.py

```
base_test.py
1        from selenium import webdriver
2
3
4      class BaseTest:
5
6          def __init__(self):
7              self.driver = None
8
9          def start_browser(self):
10             self.driver = webdriver.Chrome(executable_path=
11                                            "C:/Drivers/chromedriver")
12             self.driver.implicitly_wait(30)
13             self.driver.maximize_window()
14             self.driver.get("https://demo.actitime.com/")
15             return self.driver
16
17         def stop_browser(self):
18             self.driver.quit()
19
20
```

Test_create_task.py

Pramod K S

create_task_test.py

```python
1   from pages.home_page import HomePage
2   from pages.login_page import LoginPage
3   from pages.task_page import TaskPage
4   from test_scripts.base_test import BaseTest
5
6   browser = BaseTest()
7   driver = browser.start_browser()
8   login = LoginPage(driver)
9   login.enter_user_name("admin")
10  login.enter_password("manager")
11  login.click_on_login_button()
12
13  home = HomePage(driver)
14  home.click_on_tab("Tasks")
15
16  task = TaskPage(driver)
17  task.click_on_add_new_button()
18  task.click_on_add_new_task_button()
19  task.click_on_cancel_button()
20  home.click_on_log_out()
21  browser.stop_browser()
```

Pramod K S