Guía de Estudio Para Certificación

Automatizador Profesional de Pruebas con Selenium y Cucumber (CJS-71)

Versión 1.2 junio 2025

Contenido

1. Introducción a Selenium WebDriver

1.1 ¿Qué es Selenium WebDriver?

Selenium WebDriver es una herramienta de automatización de pruebas que permite interactuar directamente con navegadores web de la misma forma que lo haría un usuario humano. Es un componente del conjunto de herramientas de Selenium que facilita la automatización a nivel de navegador, permitiendo simular acciones del usuario, como hacer clic, ingresar texto, navegar entre páginas, entre otros.

WebDriver utiliza los controladores específicos de cada navegador para ejecutar comandos y recuperar información sobre el estado de los elementos de la página, asegurando una interacción más precisa y fiel al comportamiento del usuario.

1.2 Diferencias entre Selenium IDE, WebDriver y Selenium Grid

- **Selenium IDE:** Es un complemento para navegadores que permite grabar, editar y depurar pruebas. Está diseñado principalmente para usuarios principiantes y se utiliza para crear pruebas rápidas sin necesidad de escribir código.
- **Selenium WebDriver:** Es una API que permite escribir scripts de automatización en varios lenguajes de programación, como Java, Python, C#, entre otros. A diferencia de Selenium IDE, WebDriver interactúa directamente con el navegador, proporcionando mayor control y precisión en la automatización.

- **Selenium Grid:** Es una herramienta que permite ejecutar pruebas en múltiples navegadores y sistemas operativos en paralelo, distribuyendo las ejecuciones en diferentes nodos. Se utiliza principalmente para pruebas en múltiples entornos y para realizar pruebas de regresión más rápidamente.

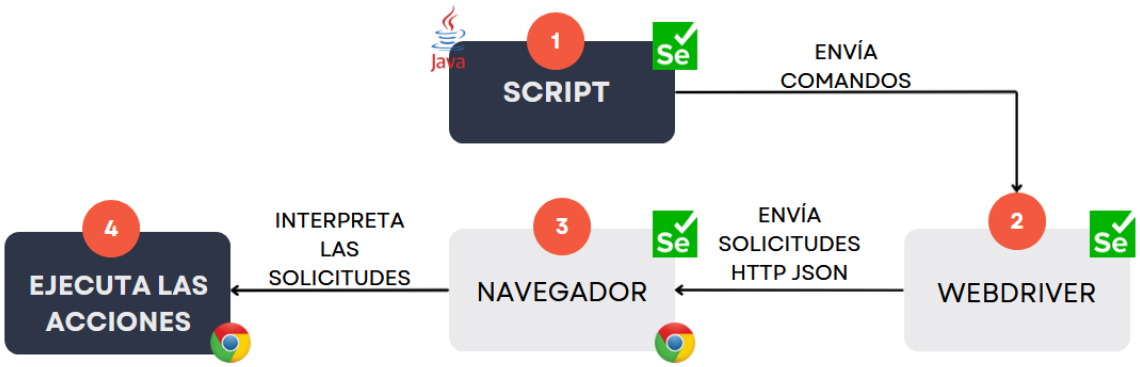
1.3 Arquitectura de Selenium WebDriver

La arquitectura de Selenium WebDriver está compuesta por los siguientes componentes:

1. **Client Library:** Biblioteca que contiene los comandos de WebDriver en distintos lenguajes de programación.
2. **JSON Wire Protocol:** Protocolo que permite la comunicación entre el cliente (WebDriver) y el servidor (controlador del navegador).
3. **Browser Drivers:** Drivers específicos para cada navegador (ChromeDriver, GeckoDriver, EdgeDriver, etc.) que actúan como intermediarios entre WebDriver y el navegador.
4. **Browsers:** Los navegadores web donde se ejecutan las pruebas.

El flujo de comunicación es el siguiente:

- El script de prueba envía comandos a WebDriver.
- WebDriver convierte los comandos en solicitudes HTTP JSON.
- El controlador del navegador recibe las solicitudes y las convierte en acciones específicas del navegador.
- El navegador ejecuta las acciones y devuelve el resultado a WebDriver.



1.4 Métodos esenciales de Selenium WebDriver

Algunos de los métodos más utilizados en Selenium WebDriver incluyen:

1. **get(String URL):** Abre una URL específica en el navegador.
2. **findElement(By locator):** Localiza un elemento en la página.
3. **findElements(By locator):** Localiza múltiples elementos en la página.
4. **click():** Hace clic en un elemento.
5. **sendKeys(String text):** Ingresa texto en un campo de entrada.
6. **getText():** Obtiene el texto visible de un elemento.
7. **getTitle():** Obtiene el título de la página actual.
8. **navigate().to(String URL):** Navega a una nueva URL.
9. **navigate().back():** Navega hacia atrás en el historial del navegador.
10. **navigate().forward():** Navega hacia adelante en el historial del navegador.
11. **navigate().refresh():** Recarga la página actual.

Estos métodos forman la base para la mayoría de los scripts de automatización y permiten interactuar con elementos del DOM de manera efectiva.

2. Interacción con Elementos Web

2.1 Localizadores en Selenium (ID, Name, XPath, CSS Selector)

Los localizadores son identificadores utilizados para encontrar elementos en la página web. Selenium proporciona varios métodos de localización, cada uno con sus ventajas y limitaciones:

1. **ID:** Es el localizador más rápido y preciso, ya que los IDs son únicos por definición. Ejemplo:

```
WebElement element = driver.findElement(By.id("submit"));
```

2. **Name:** Similar al ID, pero no garantiza unicidad. Ejemplo:

```
WebElement element = driver.findElement(By.name("username"));
```

3. **XPath:** Permite localizar elementos basados en su estructura y relación con otros elementos. Es muy potente pero más lento que ID o Name. Ejemplo:

```
WebElement element =
        driver.findElement(By.xpath( xpathExpression: "//button[text()='Submit']"));
```

4. **CSS Selector:** Similar a XPath en términos de versatilidad, pero generalmente más rápido. Ejemplo:

```
WebElement element
        = driver.findElement(By.cssSelector("button.submit"));
```

2.2 Métodos para interactuar con elementos: click(), sendKeys(), getText()

- **click():** Realiza un clic sobre el elemento.

```
element.click();
```

- **sendKeys():** Envía texto al elemento, como un campo de entrada.

```
element.sendKeys("miTexto");
```

- **getText():** Obtiene el texto visible del elemento.

```
String texto = element.getText();
```

2.3 Validación de elementos: isDisplayed(), isSelected(), isEnabled()

Estos métodos permiten verificar el estado de los elementos:

- **isDisplayed():** Verifica si el elemento es visible en la interfaz de usuario.

```
boolean visible = element.isDisplayed();
```

- **isSelected():** Verifica si un elemento, como un checkbox o radio button, está seleccionado.

```
boolean seleccionado = checkbox.isSelected();
```

- **isEnabled():** Verifica si un elemento está habilitado para la interacción.

```
boolean habilitado = button.isEnabled();
```

2.4 Manejo de elementos dinámicos y técnicas recomendadas

Los elementos dinámicos cambian sus atributos o ubicación con cada carga de página. Para manejarlos de forma efectiva, se recomienda:

- Uso de **XPath dinámico**:

```
WebElement dynamicElement
  = driver.findElement(By.xpath("//button[contains(@id, 'submit_')]"));
```

- Implementar **esperas explícitas (WebDriverWait)** para esperar hasta que los elementos sean visibles, clicables o presentes en el DOM:

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement element =
        wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("dynamicElement")));
```

- Uso de **CSS Selectors flexibles**, combinando atributos:

```
WebElement element
        = driver.findElement(By.cssSelector("button[id^='submit_']"));
```

- Uso de **acciones avanzadas (Actions Class)** para interactuar con elementos dinámicos, arrastrar y soltar, o realizar desplazamientos controlados:

```
Actions actions = new Actions(driver);
actions.moveToElement(button).click().perform();
```

3. Manejo de Ventanas y Alertas

3.1 Cambiar entre ventanas: switchTo().window()

En Selenium WebDriver, se puede interactuar con múltiples ventanas o pestañas utilizando el método switchTo().window(). Este método permite cambiar el contexto de la ejecución a una ventana específica mediante su identificador (Window Handle).

- **Obtener el identificador de la ventana actual:**

```
String mainWindow = driver.getWindowHandle();
```

- **Obtener todos los identificadores de ventana abiertos:**

```
Set<String> allWindows = driver.getWindowHandles();
```

- **Cerrar una ventana específica y volver a la ventana principal:**

```
driver.close();
driver.switchTo().window(mainWindow);
```

- **Ejemplo completo**:

```
driver.get("https://example.com");
String mainWindow = driver.getWindowHandle();

driver.findElement(By.linkText("Open new window")).click();
Set<String> windows = driver.getWindowHandles();

for (String window : windows) {
    if (!window.equals(mainWindow)) {
        driver.switchTo().window(window);
        System.out.println("New window title: " + driver.getTitle());
        driver.close();
    }
}

driver.switchTo().window(mainWindow);
System.out.println("Back to main window: " + driver.getTitle());
```

3.2 Manejo de alertas: switchTo().alert() y sus métodos (getText(), accept(), dismiss())

Selenium WebDriver permite manejar alertas emergentes mediante el método switchTo().alert(). Los métodos más comunes para interactuar con alertas son:

- **Aceptar la alerta (accept()):**

```
Alert alert = driver.switchTo().alert();
alert.accept();
```

- **Cancelar la alerta (dismiss()):**

```
Alert alert = driver.switchTo().alert();
alert.dismiss();
```

- **Obtener el texto de la alerta (getText()):**

```
Alert alert = driver.switchTo().alert();
String alertText = alert.getText();
System.out.println("Alert Text: " + alertText);
```

- **Enviar texto a la alerta (sendKeys())** *(Solo para alertas tipo prompt)*:

```
Alert alert = driver.switchTo().alert();
alert.sendKeys("Texto enviado");
alert.accept();
```

3.3 Captura de pantallas (TakesScreenshot)

Selenium proporciona la interfaz TakesScreenshot para capturar imágenes del navegador. Es útil para registrar evidencias de fallos o validar visualmente el estado de una página.

- **Convertir el controlador del navegador a TakesScreenshot:**

```
TakesScreenshot screenshot = (TakesScreenshot) driver;
```

- **Capturar la pantalla y guardarla como un archivo:**

```
TakesScreenshot screenshot = (TakesScreenshot) driver;
File srcFile = screenshot.getScreenshotAs(OutputType.FILE);
File destFile = new File("screenshot.png");
Files.copy(srcFile.toPath(), destFile.toPath(),
        StandardCopyOption.REPLACE_EXISTING);
```

- **Capturar la pantalla y convertirla en base64:**

```
String base64Screenshot
        = screenshot.getScreenshotAs(OutputType.BASE64);
System.out.println(base64Screenshot);
```

Estos métodos son esenciales para interactuar con ventanas emergentes, alertas y para capturar evidencias visuales durante la ejecución de pruebas automatizadas.

4. Listas Desplegables (Dropdowns)

En Selenium WebDriver, la clase Select proporciona métodos para interactuar con listas desplegables (<select>). Esta clase facilita la selección y de-selección de opciones mediante distintos criterios.

4.1 Clase Select y sus métodos (selectByVisibleText(), selectByValue(), selectByIndex())

Para utilizar la clase Select, primero se debe identificar el elemento <select> y crear una instancia de Select.

Crear una instancia de Select:

```
WebElement dropdown = driver.findElement(By.id("miDropdown"));
Select select = new Select(dropdown);
```

1. selectByVisibleText(String text)

Este método selecciona una opción basada en el texto visible dentro de la opción (<option>). Es útil cuando se quiere seleccionar una opción específica conocida por su texto.

Sintaxis:

```
select.selectByVisibleText("Opción 2");
```

Ejemplo completo:

```
WebElement dropdown = driver.findElement(By.id("miDropdown"));
Select select = new Select(dropdown);
select.selectByVisibleText("Opción 2");
```

HTML de ejemplo:

```
<select id="miDropdown">
    <option value="1">Opción 1</option>
    <option value="2">Opción 2</option>
    <option value="3">Opción 3</option>
</select>
```

2. selectByValue(String value)

Este método selecciona una opción basada en el valor del atributo value del <option>. Es ideal cuando los valores son únicos y constantes.

Sintaxis:

```
select.selectByValue("2");
```

Ejemplo completo:

```
WebElement dropdown = driver.findElement(By.id("miDropdown"));
Select select = new Select(dropdown);
select.selectByValue("2");
```

3. selectByIndex(int index)

Este método selecciona una opción basada en su posición indexada, comenzando desde 0. Es útil cuando se necesita seleccionar opciones dinámicas o basadas en posición.

Sintaxis:

```
select.selectByIndex(1);//Selecciona "Opción 2"
```

Ejemplo completo:

```
WebElement dropdown = driver.findElement(By.id("miDropdown"));
Select select = new Select(dropdown);
select.selectByIndex(1);
```

4.2 Métodos Adicionales de la Clase Select

Además de los métodos de selección, Select también proporciona métodos para obtener información sobre las opciones seleccionadas:

Obtener todas las opciones del dropdown:

```
WebElement dropdown = driver.findElement(By.id("miDropdown"));
Select select = new Select(dropdown);

List<WebElement> opciones = select.getOptions();
for (WebElement opcion : opciones) {
    System.out.println(opcion.getText());
}
```

Obtener la opción seleccionada actualmente:

```
WebElement selectedOption = select.getFirstSelectedOption();
System.out.println("Opción seleccionada: " + selectedOption.getText());
```

Deseleccionar opciones (solo para dropdowns de selección múltiple):

```
select.deselectAll();
select.deselectByIndex(1);
```

5. Interacciones Avanzadas con Actions

La clase Actions en Selenium WebDriver permite realizar interacciones avanzadas que simulan acciones del usuario como hacer clic sostenido, doble clic, arrastrar y soltar, entre otras. Es especialmente útil para manejar elementos dinámicos o complejos en la interfaz del navegador.

5.1 Actions Class y sus Métodos (clickAndHold(), release(), doubleClick())

La clase Actions se debe instanciar utilizando el WebDriver y permite encadenar múltiples acciones en una secuencia.

Instanciación de la clase Actions:

```
Actions actions = new Actions(driver);
```

1. clickAndHold()

clickAndHold() mantiene el clic del ratón sin soltarlo. Se utiliza para arrastrar elementos o para interacciones complejas donde se requiere un clic sostenido.

Sintaxis:

```
actions.clickAndHold(elemento).perform();
```

2. release()

release() suelta el clic del ratón previamente mantenido con clickAndHold(). Se utiliza en combinación con clickAndHold() para completar la acción.

Sintaxis:

```
actions.release().perform();
```

Ejemplo Completo:

```
WebElement slider = driver.findElement(By.id("miSlider"));
Actions actions = new Actions(driver);
actions.clickAndHold(slider).moveByOffset(100, 0).release().perform();
```

3. doubleClick()

doubleClick() realiza un doble clic sobre el elemento especificado. Es útil para activar elementos o abrir opciones contextuales.

Sintaxis:

```
actions.doubleClick(elemento).perform();
```

Ejemplo Completo:

```
WebElement button = driver.findElement(By.id("doubleClickButton"));
Actions actions = new Actions(driver);
actions.doubleClick(button).perform();
```

5.2 Manejo de Arrastrar y Soltar (dragAndDrop() y dragAndDropBy())

En Selenium, la clase Actions proporciona métodos específicos para arrastrar y soltar elementos. Esto es fundamental en interfaces dinámicas, como sliders, paneles, o elementos reorganizables.

1. dragAndDrop()

dragAndDrop() permite arrastrar un elemento y soltarlo en otro.

Sintaxis y Ejemplo Completo:

```
WebElement source = driver.findElement(By.id("arrastrar"));
WebElement target = driver.findElement(By.id("soltar"));
Actions actions = new Actions(driver);
actions.dragAndDrop(source, target).perform();
```

2. dragAndDropBy()

dragAndDropBy() permite arrastrar un elemento a una posición específica mediante coordenadas x e y.

Sintaxis y Ejemplo Completo:

```
WebElement source = driver.findElement(By.id("arrastrar"));
Actions actions = new Actions(driver);
actions.dragAndDropBy(source, 150, 0).perform();
```

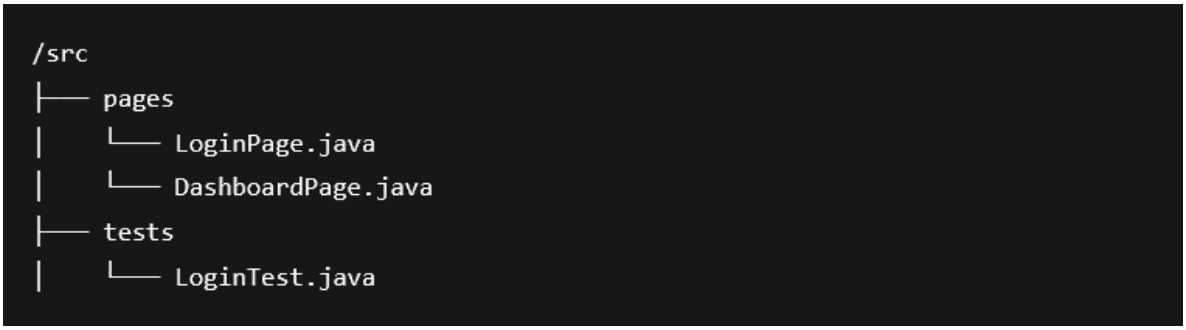
6. Estructura de Páginas - Page Object Model (POM)

El modelo **Page Object Model (POM)** es un patrón de diseño estructural utilizado en la automatización de pruebas con Selenium. En POM, cada página de la aplicación web se representa como una clase, y los elementos de la página se definen como variables. Los métodos de la clase implementan acciones que el usuario puede realizar en esa página.

6.1 ¿Qué es POM?

El **Page Object Model** es un patrón que separa la lógica de pruebas (Test Cases) de la lógica de la interfaz de usuario (Pages). En lugar de definir los localizadores y métodos de interacción directamente en los casos de prueba, se crean clases independientes para cada página de la aplicación.

Estructura básica del POM:



En este modelo, cada página tiene su propia clase (LoginPage.java, DashboardPage.java), que contiene:

1. **Localizadores:** Definidos como variables WebElement.
2. **Métodos de acción:** Métodos que encapsulan interacciones con los elementos.
3. **Constructor:** Inicializa los elementos utilizando PageFactory.

Ventajas del modelo POM

1. **Reutilización del Código:** Los métodos definidos en las páginas pueden ser reutilizados en múltiples casos de prueba.
2. **Mantenimiento Simplificado:** Si cambia la interfaz, solo se actualizan los localizadores en las clases de página.
3. **Legibilidad del Código:** Los casos de prueba se centran solo en la lógica de prueba, sin incluir detalles de la interfaz.
4. **Abstracción:** Se oculta la complejidad de los localizadores, proporcionando métodos más claros (login(), clickButton()).

5. **Modularidad:** Cada página se convierte en un módulo independiente, facilitando la actualización y depuración del código.

Consideraciones Importantes en POM

Uso de PageFactory:

- Facilita la inicialización de elementos mediante anotaciones @FindBy.
- Mejora la legibilidad y reduce el código repetitivo.

  PageFactory.initElements(driver, this);

Encapsulamiento:

- Los elementos deben ser privados para proteger su acceso y solo se accede a ellos mediante métodos públicos.

  private WebElement usernameField;

Manejo de Esperas:

- Es recomendable implementar esperas explícitas (WebDriverWait) para gestionar elementos dinámicos.

```
public void waitForLoginVisibility(WebElement elemento) {
    new WebDriverWait(driver, Duration.ofSeconds(10))
            .until(ExpectedConditions.visibilityOf(elemento));
}
```

7. Introducción a Cucumber y Gherkin

Cucumber es una herramienta de automatización de pruebas que facilita la escritura de casos de prueba en un formato legible para todos los interesados, utilizando un lenguaje llamado **Gherkin**. Este enfoque permite crear pruebas basadas en el comportamiento (BDD - Behavior Driven Development), alineando la comunicación entre desarrolladores, testers y stakeholders.

7.1 ¿Qué es Cucumber?

Cucumber es una herramienta open source que permite la creación de pruebas de aceptación automatizadas basadas en BDD. Estas pruebas están escritas en Gherkin, un lenguaje simple y natural que describe el comportamiento del sistema a través de escenarios de prueba.

Beneficios de Cucumber:

- Mejora la colaboración entre equipos técnicos y no técnicos.
- Documenta los requisitos del sistema de manera estructurada.
- Facilita la automatización de pruebas basadas en el comportamiento.

7.2 Sintaxis de Gherkin (Feature, Scenario, Given, When, Then)

Gherkin es un lenguaje basado en texto estructurado que permite escribir pruebas de manera legible. Los archivos Gherkin tienen la extensión .feature y se componen de los siguientes elementos:

Feature: Define la funcionalidad que se va a probar.

Scenario: Representa un caso de prueba específico.

Given: Establece el contexto inicial.

When: Define la acción que desencadena un evento.

Then: Especifica el resultado esperado.

Estructura Básica de un Archivo Feature:

```
Feature: Login Functionality

  Scenario: Login con credenciales válidas
    Given el usuario está en la página de inicio de sesión
    When ingresa el nombre de usuario "usuarioValido"
    And ingresa la contraseña "contraseña123"
    And hace clic en el botón de login
    Then debería ver el mensaje "Login exitoso"
```

7.3 Uso de Scenario Outline y Examples

El **Scenario Outline** permite ejecutar el mismo escenario con diferentes conjuntos de datos. Esto se logra mediante el uso de la tabla **Examples**.

```
Scenario Outline: Login con diferentes credenciales
  Given el usuario está en la página de inicio de sesión
  When ingresa el nombre de usuario "<username>"
  And ingresa la contraseña "<password>"
  And hace clic en el botón de login
  Then debería ver el mensaje "<mensaje>"

  Examples:
    | username      | password     | mensaje               |
    | usuarioValido | pass123      | Login exitoso         |
    | usuarioInvalido | incorrecta  | Credenciales inválidas |
```

8. Data Tables en Cucumber

Data Tables en Cucumber permiten estructurar datos en forma de tabla dentro de un escenario Gherkin. Esta estructura facilita la representación de múltiples entradas de datos sin necesidad de crear escenarios adicionales. Los datos pueden ser convertidos a listas o mapas en los Step Definitions para su procesamiento.

8.1 Definición de Data Table

En Gherkin, una Data Table se define utilizando líneas con | para separar los valores en formato tabular.

Sintaxis Básica:

```
Scenario: Ingreso de múltiples credenciales
  Given los siguientes usuarios:
    | usuario     | contraseña |
    | admin       | admin123   |
    | user1       | pass1      |
    | user2       | pass2      |
```

En este ejemplo, la tabla contiene dos columnas (usuario y contraseña) y tres filas de datos. Los datos se pueden acceder y manipular en el Step Definition correspondiente.

8.2 Conversión a Listas y Mapas (asList(), asMaps())

Cucumber proporciona métodos para convertir las tablas en estructuras de datos Java, facilitando su manipulación.

1. Conversión a Lista (asList())

asList() convierte la tabla en una lista de listas. Este método es útil cuando se necesita acceder a los datos en forma de filas.

Ejemplo:

```
Scenario: Ingreso de datos en formulario
  Given los siguientes campos:
    | campo      |
    | nombre     |
    | email      |
    | contraseña |
```

2. Conversión a Mapa (asMaps())

asMaps() convierte la tabla en una lista de mapas (List<Map<String, String>>), donde cada fila se convierte en un mapa clave-valor basado en los encabezados.

Ejemplo:

```
Scenario: Registro de múltiples usuarios
  Given los siguientes usuarios:
    | usuario | contraseña |
    | admin   | admin123   |
    | user1   | pass1      |
    | user2   | pass2      |
```

8.3 Consejos para el Uso de Data Tables:

Mantener la coherencia de los encabezados: Los nombres de las columnas deben ser únicos y representativos.

Uso de listas para datos simples: Cuando se trata de datos simples sin encabezados, asList() es más conveniente.

Uso de mapas para estructuras complejas: asMaps() es ideal cuando se necesita una estructura clave-valor.

Conversión a objetos personalizados: Es posible convertir los datos en objetos utilizando asList(MyObject.class), lo que permite trabajar con POJOs directamente.

9. Manejo de Hooks en Cucumber

Los **Hooks** en Cucumber son bloques de código que se ejecutan antes o después de un escenario. Son útiles para configurar el entorno de prueba, inicializar datos, capturar capturas de pantalla o limpiar los datos generados durante el test.

9.1 Definición y Uso de @Before, @After y Otros Hooks

Cucumber proporciona varios tipos de hooks:

- **@Before**: Se ejecuta antes de cada escenario.
- **@After**: Se ejecuta después de cada escenario.
- **@BeforeStep**: Se ejecuta antes de cada paso (step).
- **@AfterStep**: Se ejecuta después de cada paso.

1. Hook @Before

@Before se utiliza para inicializar el entorno, abrir el navegador, establecer conexiones a bases de datos, etc.

2. Hook @After

@After se utiliza para realizar tareas de limpieza, como cerrar el navegador, eliminar datos de prueba, etc.

3. Hook @BeforeStep

@BeforeStep se ejecuta **antes de cada paso** del escenario. Es útil para agregar verificaciones o tomar capturas de pantalla antes de cada acción.

4. Hook @AfterStep

@AfterStep se ejecuta **después de cada paso**. Se utiliza para realizar acciones de seguimiento o captura de evidencias.

Implementación de Hooks Condicionales Mediante Tags

En Cucumber, se pueden definir hooks específicos para determinados escenarios utilizando **tags**. Esto permite ejecutar ciertos hooks solo cuando el escenario contiene un tag específico.

Ejemplo de Hooks Condicionales con Tags:

```
@SmokeTest
Scenario: Login con credenciales válidas
  Given el usuario está en la página de inicio de sesión
  When ingresa el usuario "admin"
  And ingresa la contraseña "admin123"
  Then debería ver el mensaje "Login exitoso"

@RegressionTest
Scenario: Login con credenciales inválidas
  Given el usuario está en la página de inicio de sesión
  When ingresa el usuario "user1"
  And ingresa la contraseña "incorrecta"
  Then debería ver el mensaje "Credenciales inválidas"
```

Hook Condicional para @SmokeTest:

En este ejemplo, el hook @Before solo se ejecutará para escenarios etiquetados con @SmokeTest.

```
import io.cucumber.java.Before;

public class Hooks {

    @Before("@SmokeTest")
    public void beforeSmokeTest() {
        System.out.println("Preparando entorno para Smoke Test...");
    }
}
```

9.2 Uso Avanzado: Prioridad de Hooks

Cucumber permite establecer prioridades en los hooks mediante el parámetro (order). Los hooks con menor orden se ejecutan primero.

```
@Before(order = 1)
public void firstHook() {
    System.out.println("Primer Hook");
}

@Before(order = 2)
public void secondHook() {
    System.out.println("Segundo Hook");
}
```

10. Pruebas de API con Selenium y Cucumber

Selenium WebDriver no está diseñado específicamente para pruebas de API, pero se puede integrar con librerías como RestAssured o HttpClient para realizar solicitudes HTTP y validar respuestas JSON. Cucumber se utiliza para estructurar las pruebas en formato Gherkin, permitiendo que los casos de prueba sean legibles y reutilizables.

10.1 Conceptos Clave de Pruebas de API

Las pruebas de API consisten en verificar el comportamiento y la funcionalidad de los endpoints mediante solicitudes HTTP (GET, POST, PUT, DELETE). Las pruebas de API deben incluir:

1. **Verificación de Códigos de Estado (HTTP Status Codes):**
    - 200 - Éxito (OK)
    - 201 - Recurso creado
    - 404 - No encontrado
    - 500 - Error interno del servidor
2. **Validación del Cuerpo de la Respuesta (Response Body):**

    Verificar estructuras JSON.

    Validar valores específicos en el JSON.
3. **Encabezados y Autenticación:**

    Verificar headers (Content-Type, Authorization).

    Manejar tokens o claves API

11. Pruebas No Funcionales

Las pruebas no funcionales se centran en evaluar el rendimiento, la estabilidad y la capacidad de una aplicación web bajo diferentes condiciones. En el contexto de Selenium, este tipo de pruebas se implementa comúnmente en conjunto con herramientas especializadas como **JMeter**, **Gatling**, **Locust** o **Selenium Grid**

11.1 Pruebas de Carga, Rendimiento y Estrés

1. **Pruebas de Carga (Load Testing):**
    - Evalúan cómo responde la aplicación bajo un número específico de usuarios simultáneos.
    - Objetivo: Identificar el punto en el que el sistema comienza a degradar su rendimiento.
2. **Pruebas de Rendimiento (Performance Testing):**
    - Miden el tiempo de respuesta, velocidad de procesamiento y utilización de recursos bajo condiciones normales.
3. **Pruebas de Estrés (Stress Testing):**
    - Evalúan el comportamiento del sistema bajo condiciones extremas o pico de carga.
    - Objetivo: Identificar el punto de quiebre y la recuperación del sistema.

12. Aserciones y Validaciones en Pruebas Automatizadas

Las **aserciones** en pruebas automatizadas permiten comparar el resultado esperado con el resultado real, asegurando que el comportamiento de la aplicación sea el esperado. En **JUnit**, se utilizan métodos como assertEquals(), assertTrue() y assertFalse() para realizar estas validaciones.

12.1 Uso de diferentes tipos de Asserts

1. **assertEquals(expected, actual)**

   Compara dos valores y valida si son iguales. Si no coinciden, la prueba falla.

```
Assert.assertEquals("Mensaje de error", experado, actual);
```

2. **assertTrue(condition)**

   Verifica si una condición es verdadera. Si es falsa, la prueba falla. Por ejemplo, la condición puede ser algo como; 10 > 5 o 10 / 2 == 5, etc. En general, cualquier condición booleana.

```
Assert.assertTrue("Mensaje de error", condicion);
```

3. **assertFalse(condition)**

   Verifica si una condición es falsa. Si es verdadera, la prueba falla. Mismo caso de assetTrue, pero para verificar que el resultado siempre sea False, si el resultado de la comprobación booleana es True, la prueba falla.

```
Assert.assertFalse("Mensaje de error", condición);
```

Verificar Múltiples Condiciones con assertTrue()

En lugar de utilizar múltiples assertTrue(), se pueden combinar condiciones lógicas:

```
@Test
public void testMultipleConditions() {
    int userAge = 25;
    String userRole = "admin";

    Assert.assertTrue("El usuario debe ser mayor de edad y administrador",
                      userAge > 18 && userRole.equals("admin"));
}
```

Aserciones Condicionales en Funciones Personalizadas

En algunos casos, se pueden crear métodos que encapsulen las aserciones para mejorar la legibilidad del código.

```
public void verifyLoginStatus(boolean isLoggedIn, String userRole) {
    Assert.assertTrue("El usuario debe estar logueado", isLoggedIn);
    Assert.assertEquals("El rol debe ser administrador", "admin", userRole);
}

@Test
public void testLoginVerification() {
    verifyLoginStatus(true, "admin");
}
```

Validación de Valores Nulos o Vacíos

Es importante validar que los objetos no sean null antes de realizar comparaciones o aserciones.

```
@Test
public void testNullOrEmpty() {
    String response = null;

    Assert.assertNull("La respuesta debería ser nula", response);

    response = "Data";
    Assert.assertNotNull("La respuesta no debería ser nula", response);
}
```

12.2 Resumen y Buenas Prácticas:

- Utilizar assertEquals() para comparar valores exactos.
- Emplear assertTrue() y assertFalse() para validar condiciones booleanas.
- Combinar condiciones lógicas para reducir la cantidad de aserciones en un mismo test.
- Validar valores null o vacíos antes de realizar otras aserciones.

13. Estrategias para el Manejo de Excepciones en Selenium

El manejo adecuado de excepciones es fundamental en la automatización de pruebas con Selenium. Las excepciones permiten detectar errores durante la ejecución del script y proporcionar mensajes claros sobre el estado del sistema. En Selenium, las excepciones más comunes incluyen **NoSuchElementException**, **TimeoutException**, entre otras.

13.1 Excepciones Comunes en Selenium

1. **NoSuchElementException:**

   Se lanza cuando un elemento no se encuentra en el DOM.

2. **TimeoutException:**

   Se lanza cuando se excede el tiempo de espera para encontrar un elemento o cargar una página

3. **StaleElementReferenceException**

   Se lanza cuando el elemento ya no está presente en el DOM o ha cambiado.

4. **ElementClickInterceptedException:**

   Se lanza cuando un elemento está cubierto por otro elemento, impidiendo el clic.

13.2 Estrategias para Manejar Excepciones en Flujos de Prueba

1. Uso de Try-Catch para Capturar Excepciones Específicas

Es una de las estrategias más utilizadas. Permite capturar excepciones específicas y ejecutar acciones de recuperación.

```
public void safeClick(WebElement element) {
    try {
        element.click();
    } catch (NoSuchElementException e) {
        System.out.println("Elemento no encontrado. Reintentando...");
        // Reintentar o tomar una captura de pantalla
    } catch (ElementClickInterceptedException e) {
        System.out.println("Elemento bloqueado. Reintentando...");
    }
}
```

2. Uso de Waits Inteligentes para Evitar Excepciones

Las excepciones como **NoSuchElementException** y **TimeoutException** se pueden prevenir utilizando esperas explícitas (WebDriverWait).

```
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.support.ui.ExpectedConditions;

public WebElement waitForElement(By locator, int timeout) {
    WebDriverWait wait = new WebDriverWait(driver, timeout);
    return wait.until(ExpectedConditions.visibilityOfElementLocated(locator));
}
```

3. Estrategia de Reintento (Retry)

En casos donde los elementos son dinámicos o tardan en aparecer, se puede implementar una estrategia de reintento.

```
public void testWithFinally() {
    WebDriver driver = new ChromeDriver();
    try {
        driver.get("https://example.com");
        WebElement element = driver.findElement(By.id("test"));
        element.click();
    } catch (Exception e) {
        System.out.println("Error al interactuar con el elemento: " + e.getMessage());
    } finally {
        driver.quit(); // Garantiza el cierre del navegador
    }
}
```

4. Uso de Finally para Cerrar Recursos

El bloque finally se utiliza para garantizar que se ejecuten ciertas acciones, independientemente de si ocurre una excepción o no.

```
public void testWithFinally() {
    WebDriver driver = new ChromeDriver();
    try {
        driver.get("https://example.com");
        WebElement element = driver.findElement(By.id("test"));
        element.click();
    } catch (Exception e) {
        System.out.println("Error al interactuar con el elemento: " + e.getMessage());
    } finally {
        driver.quit(); // Garantiza el cierre del navegador
    }
}
```

13.2 Resumen y Buenas Prácticas:

- **Identificar Excepciones Comunes:** Comprender las causas de excepciones específicas como NoSuchElementException, TimeoutException, StaleElementReferenceException.
- **Implementar Estrategias de Reintento:** Reintentar operaciones críticas en caso de fallos temporales.
- **Centralizar el Manejo de Excepciones:** Crear métodos o clases dedicadas al manejo de excepciones para evitar código duplicado.
- **Aplicar Waits Inteligentes:** Emplear esperas explícitas (WebDriverWait) para evitar excepciones por tiempos de carga.
- **Capturar Evidencias de Error:** Implementar capturas de pantalla o logs detallados para facilitar la depuración.

14. Pruebas de Integración y Regresión Automatizadas

Las pruebas de integración y regresión son esenciales para garantizar que los componentes del sistema funcionen correctamente cuando se integran entre sí y que los cambios en el código no introduzcan nuevos defectos.

14.1 Pruebas de Integración

Las pruebas de integración tienen como objetivo verificar que los diferentes módulos o componentes del sistema funcionen correctamente cuando se integran entre sí. Estas pruebas aseguran que las interacciones entre módulos sean correctas y que los datos fluyan como se espera.

Objetivos de las pruebas de integración:

- Verificar la correcta comunicación entre componentes.
- Detectar defectos en interfaces y puntos de integración.
- Asegurar la integridad de los datos a través de los módulos.
- Identificar errores de compatibilidad o dependencias.

Ejemplo de Integración:

En una aplicación de comercio electrónico, se debe verificar que el módulo de pago:

(PaymentService) funcione correctamente con el módulo de inventario

(InventoryService) para asegurar que se actualice el stock tras un pago exitoso.

14.2 Pruebas de Regresión:

Las pruebas de regresión se enfocan en validar que los cambios o actualizaciones en el sistema no generen defectos en las funcionalidades existentes. Se ejecutan después de cada modificación, actualización o corrección de errores.

Objetivos de las pruebas de regresión:

- Verificar que las funcionalidades existentes no se vean afectadas por los cambios.
- Asegurar que los defectos corregidos no vuelvan a aparecer.
- Identificar efectos colaterales no previstos.

Ejemplo de Regresión:

Si se actualiza la lógica de cálculo de precios en una aplicación de e-commerce, se deben reejecutar los casos de prueba relacionados con la visualización del precio, cálculo de impuestos, descuentos, etc.

14.3 Estrategias para la Implementación de Pruebas de Regresión

Implementar pruebas de regresión efectivas implica planificar cuidadosamente qué casos de prueba se deben incluir, cómo priorizarlos y cómo ejecutarlos de forma eficiente.

1. Priorización de Casos de Prueba:

En grandes aplicaciones, es inviable ejecutar todos los casos de prueba en cada ciclo. Se deben priorizar los casos según:

Casos Críticos: Funcionalidades esenciales para el sistema (login, pagos, envíos).

Casos Impactados por los Cambios: Funcionalidades directamente relacionadas con las modificaciones.

Casos de Defectos Recurrentes: Funcionalidades donde se han detectado defectos en el pasado

```
public enum Priority {
    CRITICAL, HIGH, MEDIUM, LOW;
}

@Test(priority = Priority.CRITICAL)
public void testLoginFunctionality() {
    // Caso de prueba crítico
}
```

14.4 Uso de Suites de Regresión:

Crear suites de regresión organizadas por módulos, funcionalidades o áreas del sistema:

- **Smoke Suite:** Casos críticos que deben ejecutarse siempre.
- **Sanity Suite:** Casos de prueba para verificar áreas impactadas.
- **Full Regression Suite:** Todos los casos de prueba, ejecutados en ciclos programados (e.g., cada semana).

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
        LoginTest.class,
        CheckoutTest.class,
        InventoryTest.class
})
public class RegressionSuite {
}
```

14.5 Resumen y Buenas Prácticas:

- **Priorización Estratégica:** No es necesario ejecutar todos los casos en cada ciclo; priorizar los casos críticos y los relacionados con las áreas modificadas.
- **Automatización Sostenible:** Implementar scripts automatizados para casos de regresión estables y repetitivos.
- **Gestión de Suites de Regresión:** Organizar los casos de prueba en suites lógicas (Smoke, Sanity, Full Regression).

Esta guía de estudio fue desarrollada para ayudarte a rendir exitosamente el examen de certificación CJS-71 como Automatizador Profesional de Pruebas con Selenium y Cucumber por el Centro Latinoamericano de Testing y Calidad del Software.

Puedes complementar estos conocimientos con nuestros cursos disponibles en **https://www.centyc.com.ar**