Revisión de Puntos Clave de JUnit: Uso de Assertions en Pruebas Automatizadas

En el mundo del testing automatizado con Selenium, las **assertions** juegan un papel crucial. Son el método mediante el cual puedes verificar si el estado actual de la aplicación web coincide con el estado esperado. Las **assertions** son las que realmente determinan si una prueba ha pasado o ha fallado.

¿Qué son las Assertions?

Una **assertion** es una declaración en programación que permite verificar si una condición particular es verdadera o falsa. Si la assertion se evalúa como verdadera, el programa continúa su ejecución normalmente. Si se evalúa como falsa, el programa generalmente arroja una excepción y se detiene, o bien maneja el fallo de alguna otra manera.

En el contexto de pruebas automatizadas, las **assertions** se utilizan para validar si una operación en un navegador web se ha realizado correctamente. Por ejemplo, si estás comprobando que un texto específico está presente en una página web, puedes usar una assertion para comparar el texto esperado con el texto real en la página.

Tipos de Assertions

- **Soft Assertions**: Las **Soft Assertions** permiten que una prueba continúe ejecutándose a pesar de que haya assertions que fallen durante su ejecución. Estas assertions no interrumpen el flujo de la prueba inmediatamente, sino que se registran y se evalúan al final de la ejecución. Si alguna **Soft Assertion** no se cumple, se lanzará una excepción, reportando todos los fallos que ocurrieron durante la prueba.
  Este enfoque es útil cuando necesitas verificar múltiples condiciones y

deseas obtener un informe exhaustivo de todos los fallos al final de la ejecución, en lugar de detener la prueba en el primer fallo detectado.
Aunque JUnit no ofrece una clase integrada para las **Soft Assertions**, es posible lograr un comportamiento similar utilizando estrategias alternativas, como la recolección de errores en una lista para luego verificarlos todos al concluir la prueba.

- **Hard Assertions**: Las **Hard Assertions** son el tipo más común de assertions utilizadas en pruebas de software. Estas assertions provocan que una prueba falle de inmediato si no se cumplen las condiciones especificadas. Si una **Hard Assertion** falla, la ejecución de la prueba se detiene en ese punto exacto y no continúa con las líneas de código siguientes.
  Las **Hard Assertions** son una característica fundamental en la mayoría de los frameworks de pruebas, como JUnit, debido a su importancia en la validación precisa de resultados esperados durante la ejecución de las pruebas.

¿Qué es JUnit?

JUnit es una herramienta esencial para cualquier desarrollador que quiera asegurar la calidad y la robustez de su código a través de pruebas unitarias. Las **assertions** son el corazón de las pruebas, ya que permiten verificar si el código se comporta como se espera.

¿Cómo se instala JUnit en una aplicación de Maven?

Para agregar JUnit a tu proyecto Maven, necesitas incluir la dependencia de JUnit en tu archivo `pom.xml`. Sigue estos pasos:

- Busca la versión más reciente de JUnit en la documentación oficial de Maven.
- Copia las líneas de la dependencia.
- Abre el archivo `pom.xml` de tu proyecto Maven.
- Dentro del elemento `<dependencies>`, agrega la dependencia para JUnit.
- Guarda los cambios en el archivo `pom.xml`. Maven descargará e integrará JUnit automáticamente la próxima vez que construyas el proyecto.

¿Cómo se define un test con JUnit?

La anotación `@Test` de JUnit se utiliza para indicar que un método específico es un método de prueba en una clase de pruebas. JUnit ejecutará estos métodos cuando corras la prueba. Cada método de prueba generalmente verifica una funcionalidad específica del código que estás probando, y puede utilizar **assertions** (como `assertEquals`, `assertTrue`, etc.) para verificar que el código funcione como se espera.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;

public class MiClaseDePrueba {
    @Test
    public void pruebaSuma() {
        int a = 5;
        int b = 3;
        int resultado = a + b;
        Assertions.assertEquals(8, resultado, "La suma de a y b debe ser 8");
    }
}
```

Importación de assertions

En JUnit, hay dos formas de importar las afirmaciones (assertions) en tu clase de pruebas:

1. **Importar la clase `Assertions` completa:** Si importas la clase `Assertions` completa, puedes utilizar cualquier afirmación sin necesidad de especificar su nombre completo. La sintaxis sería:

```
import org.junit.jupiter.api.Assertions;
```

Luego, puedes utilizar las afirmaciones de la siguiente forma:

```
Assertions.assertEquals(expected, actual);
Assertions.assertTrue(condicion);
```

2. **Importar afirmaciones específicas:** Alternativamente, puedes importar solo las afirmaciones que necesites. De esta manera, ahorras espacio y puedes hacer que tu código sea más claro. Para hacerlo, simplemente importa las afirmaciones específicas, como por ejemplo:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;
```

Luego, puedes usar las afirmaciones directamente sin anteponer el nombre de la clase:

```
assertEquals(expected, actual);
assertTrue(condicion);
```

En ambas formas, las afirmaciones pueden aceptar un mensaje opcional que se muestra cuando la afirmación falla, lo cual es útil para proporcionar detalles sobre el error:

```
assertEquals(expected, actual, "El valor esperado no coincide con el valor
actual");
```

Uso de mensajes personalizados

En ambas formas de importación, puedes agregar un mensaje opcional que se muestra cuando una afirmación falla. Esto es útil para proporcionar detalles sobre el error y facilitar la depuración. Por ejemplo:

```
assertEquals(expected, actual, "El valor esperado no coincide con el valor
actual.");
```

Si no incluyes un mensaje, JUnit simplemente mostrará el resultado de la comparación en el reporte de la prueba. Sin embargo, agregar un mensaje descriptivo siempre es una buena práctica, ya que permite identificar rápidamente la causa del error.

Algunos métodos comunes de assertions en JUnit

JUnit proporciona varios métodos de assertion, cada uno diseñado para un tipo específico de comparación:

- **assertEquals(expected, actual, message)**: Verifica que dos valores sean iguales.

```
//Implementación importando la clase Assertions sin mostrar un mensaje de error
personalizado.
Assertions.assertEquals("automation", input.getAttribute("value"));
//Implementación sin importar Assertions completa y con un mensaje de error
personalizado usando assertEquals directamente.
assertEquals("automation", input.getAttribute("value"), "El valor del atributo
'value' no es el esperado.");
```

```

```

- **assertNotEquals(expected, actual, message)**: Verifica que dos valores no sean iguales.

```
Assertions.assertNotEquals("manual", input.getAttribute("value"));
```

- **assertTrue(condition, message)**: Verifica que una condición sea verdadera.

```
Assertions.assertTrue(checkbox.isSelected());
```

- **assertFalse(condition, message)**: Verifica que una condición sea falsa.

```
Assertions.assertFalse(checkbox.isSelected());
```

- assertNull(condition, message): Asegura que un objeto es nulo.

```
Assertions.assertNull(temp);
```

- **assertNotNull(object, message)**: Verifica que un objeto no sea nulo.

```
Assertions.assertNotNull(driver);
```

- **assertSame(expected, actual, message)**: Verifica que dos referencias apunten al mismo objeto.

```
Assertions.assertSame(driver1, driver2);
```

- **assertNotSame(expected, actual, message)**: Verifica que dos referencias no apunten al mismo objeto.

```
Assertions.assertNotSame(driver1, driver2);
```

- **assertArrayEquals(expected, actual, message)**: Verifica que dos arrays sean iguales, tanto en contenido como en orden.

```
int[] expected = new int[]{1, 2, 3};
int[] actual = new int[]{1, 2, 3};
Assertions.assertArrayEquals(expected, actual);
```

¿Cómo se integra Selenium con JUnit ?

La integración de **Selenium** con **JUnit** permite realizar pruebas más robustas y repetibles de interfaces de usuario web. Los pasos para automatizar un test unitario son:

- Configurar el driver.

- Localizar los elementos a testear.
- Realizar las pruebas (assertions).
- Evaluar los resultados de la consola.

Ejemplo de prueba automatizada con Selenium y JUnit:

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.Keys;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.support.ui.ExpectedConditions;

public class TestingGoogle {
    @Test
    public void testingInput() {

        // Configurar el driver
        System.setProperty("webdriver.chrome.driver", "/ruta/a/chromedriver");
        WebDriver driver = new ChromeDriver();

        // Abrir el navegador en una página web
        driver.get("http://www.google.com");

        // Localizar algún elemento
        WebElement input = driver.findElement(By.cssSelector("textarea#APjFqb"));

        // Realizar una prueba
        Assertions.assertEquals("", input.getAttribute("value"));

        // Escribir en el input y buscar al presionar enter
        input.sendKeys("automation");
        input.sendKeys(Keys.ENTER);

        // Esperar hasta que el título cumpla cierta condición
        WebDriverWait waitSearch = new WebDriverWait(driver,
Duration.ofSeconds(5));
        waitSearch.until(ExpectedConditions.titleContains("automation"));

        // Realizar una prueba
        Assertions.assertTrue(driver.findElements(By.cssSelector("div#search
div.g")).size() > 0);

        // Cerrar el navegador
        driver.quit();
    }
}
```

Este script de prueba automatiza el proceso de búsqueda en Google y verifica tanto el estado inicial del input de búsqueda como la presencia de resultados después de realizar la búsqueda. Si alguna de las **assertions** no se cumple, la prueba fallará y te informará cómo y dónde está fallando.

Ajustes en las versiones más recientes de Selenium: Cambios en la implementación de búsqueda y métodos específicos

En la constante evolución de herramientas como Selenium, es normal que, con cada nueva versión, se realicen cambios en la implementación de ciertos métodos de búsqueda o en la forma en que interactuamos con los elementos en la página. Estos cambios buscan mejorar la fiabilidad, la eficiencia y la compatibilidad con las versiones más recientes de los navegadores.

Un ejemplo de esto es la forma en que obtenemos los atributos de los elementos en una página web. En versiones más antiguas de Selenium, podíamos utilizar el siguiente código para obtener la clase de un elemento:

```
String buttonClass = signInButton.getAttribute("class");
```

Este método sigue siendo válido y seguirá funcionando en muchas versiones de Selenium. Sin embargo, en las versiones más recientes, se recomienda utilizar el siguiente enfoque:

```
String logoClass2 = logo.getDomAttribute("class");
```

¿Por qué este cambio? El cambio de `getAttribute()` a `getDomAttribute()` forma parte de la evolución de la herramienta para adaptarse a las necesidades del desarrollo y la mejora del manejo de los atributos. Este tipo de ajustes es común a medida que se optimizan las tecnologías para ofrecer mejores resultados y mayor compatibilidad con los navegadores.

Recomendación: Aunque el método anterior sigue siendo funcional, se recomienda que, si estás utilizando una versión más reciente de Selenium, te adaptes al nuevo enfoque (`getDomAttribute()`) para estar alineado con las actualizaciones futuras de la herramienta. Sin embargo, si aún necesitas usar el método anterior por alguna razón, puedes seguir haciéndolo, pero es importante tener en cuenta que, en el futuro, este podría dejar de ser soportado o ser menos eficiente.

Siempre es fundamental consultar la **documentación oficial** y las actualizaciones proporcionadas por la herramienta. Esto te permitirá mantener tu código actualizado, asegurando su compatibilidad con las últimas versiones y optimizando su rendimiento.