Documento: Estrategia de Automatización – Tienda OpenCart

Proyecto Transversal: Calidad de Software Aplicada a Sistemas de Dominio Específico

Equipo: Alejandro Salazar – Tomás Mancera – Sebastián López – Santiago Cardona

Fecha: 3 de Junio de 2025

1. Introducción

El presente documento detalla la estrategia de automatización implementada para el proyecto transversal, cuyo objetivo principal es "desarrollar un conjunto de pruebas automatizadas sobre la tienda de demostración de OpenCart (https://opencart.abstracta.us/) utilizando buenas prácticas de automatización". Esta estrategia se ha diseñado para "evaluar la capacidad técnica en el diseño de pruebas, estructura de framework, selección de selectores adecuados, sincronización, validaciones, y manejo de datos externos".

El framework de automatización se ha construido siguiendo los lineamientos y requisitos técnicos especificados, buscando siempre la legibilidad, mantenibilidad y robustez del código.

2. Herramientas y Tecnologías Utilizadas

Para el desarrollo del proyecto, se emplearon las siguientes tecnologías requeridas:

- **Java:** Como lenguaje de programación principal.
- **Selenium WebDriver:** Para la interacción y automatización del navegador web.
- **Maven:** Como herramienta para la gestión del proyecto, dependencias y ciclo de vida de la construcción (build).
- **Apache POI:** Para la lectura de datos de prueba desde archivos Excel y la escritura de resultados, como se evidencia en nuestras clases ExcelReader.java y ExcelWriter.java.
- **JUnit Jupiter (JUnit 5):** Como framework para la escritura y ejecución de las pruebas unitarias y funcionales (visible en nuestras clases de test como RegisterTest.java).
- **WebDriverManager:** Utilizada en la clase BaseTest.java para la gestión automática de los binarios de los drivers del navegador.

3. Arquitectura del Framework de Automatización

La arquitectura del framework se ha centrado en la modularidad y la reutilización de código, siguiendo patrones de diseño estándar en la automatización de pruebas.

3.1. Patrón de Diseño Page Object Model (POM) Se ha implementado rigurosamente el patrón POM para la creación de las pruebas.

- Todas las interacciones con las páginas web se encapsulan en clases específicas ubicadas en el paquete com.opencart.pages (ej. HomePage.java, RegisterPage.java, ProductPage.java, CartPage.java).
- Cada clase Page Object contiene los localizadores de los elementos de su página respectiva y los métodos que representan las acciones que un usuario puede realizar sobre ella. Esto centraliza la lógica de interacción y facilita el mantenimiento, ya que si un selector cambia, solo se modifica en un lugar.

3.2. Estructura del Proyecto El proyecto Maven se ha organizado con la siguiente estructura de paquetes principal:

- com.opencart.pages: Contiene las clases del Page Object Model.
- com.opencart.test: Contiene las clases de prueba que utilizan los Page Objects para ejecutar los casos de prueba funcionales.
- com.opencart.utils: Alberga clases de utilidad reusables como ExcelReader.java, ExcelWriter.java, Constants.java (para valores constantes como la URL base), y Verify.java (para manejo de aserciones).
- src/main/resources: Directorio destinado a los archivos de datos externos, como inputData.xlsx y donde se genera outputData.xlsx.

3.3. Clase Base para Tests (BaseTest.java) Se ha creado una clase BaseTest.java de la cual heredan todas las clases de prueba. Esta clase se encarga de:

- **Configuración Inicial (@BeforeEach):**
    - Inicializa WebDriver (ChromeDriver, con capacidad para Brave Browser) utilizando WebDriverManager.
    - Maximiza la ventana del navegador.
    - Establece una espera implícita global.
    - Navega a la URL base de la aplicación obtenida desde Constants.java.
- **Finalización (@AfterEach):**

- Cierra el navegador (driver.quit()) después de cada prueba para asegurar la independencia entre tests y liberar recursos.

3.4. Clase Base para Páginas (BasePage.java) Algunas clases Page Object heredan de BasePage.java. Esta clase base común:

- Inicializa las instancias de WebDriver y WebDriverWait (explícita) para ser utilizadas por las clases Page Object hijas.
- Proporciona métodos comunes como MapsTo(String url) y obtenerTitulo().

4. Estrategia de Identificación de Elementos (Selectores)

La selección de localizadores se ha realizado buscando la mayor estabilidad y claridad posible:

- Se ha priorizado el uso de selectores como By.id y By.name cuando los elementos disponían de estos atributos únicos y estables (ej. en RegisterPage.java para los campos del formulario).
- Se ha utilizado By.cssSelector para selectores más complejos pero concisos (ej. en ProductPage.java).
- By.xpath se ha empleado cuando era necesario para localizar elementos basados en su texto (ej. HomePage.goToRegisterPage para los enlaces "My Account", "Logout", "Register") o relaciones jerárquicas complejas no fácilmente expresables con CSS.
- Se han evitado rutas absolutas en los XPaths para mejorar la mantenibilidad. Los XPaths se construyen de forma relativa y, en algunos casos, de manera dinámica (ej. HomePage.category(String category)).

5. Manejo de Sincronización (Esperas)

La correcta sincronización es crucial para la estabilidad de las pruebas automatizadas. En este proyecto, se han aplicado los tres tipos de esperas disponibles en Selenium:

- **Espera Implícita:** Se configura una única vez en BaseTest.java (y también en el constructor de BasePage.java). Esta espera se aplica globalmente mientras Selenium intenta encontrar cualquier elemento.
- **Espera Explícita (WebDriverWait):** Es la estrategia de espera principal y más recomendada. Se utiliza sistemáticamente en las clases Page Object antes de interactuar con los elementos. Se espera por condiciones específicas (ExpectedConditions) como la visibilidad del elemento

(visibilityOfElementLocated), que sea clickeable (elementToBeClickable), o cambios en la URL (urlContains). Ejemplos se encuentran en HomePage.goToRegisterPage(), ProductPage.searchForProduct(), y CartPage.isProductInCart(). La instancia de WebDriverWait se inicializa en BasePage.java.

- **Fluent Wait:** Se ha implementado un ejemplo de FluentWait en el método HomePage.selectSubCategory() para demostrar su uso, configurando el tiempo máximo de espera, la frecuencia de sondeo y las excepciones a ignorar durante la espera.

6. Manejo de Datos Externos (Apache POI y Excel)

Para facilitar la parametrización de las pruebas y el manejo de grandes conjuntos de datos, se ha utilizado Apache POI:

- **Lectura de Datos (ExcelReader.java):**
    - Esta clase de utilidad permite leer datos de archivos .xlsx.
    - Se utiliza en las clases de prueba como RegisterTest.registerUserFromExcel() y AddAndSaveProductsTest.searchAddAndSaveProducts() para obtener los datos de entrada (nombres de usuario, contraseñas, productos a buscar, etc.) desde diferentes hojas del archivo inputData.xlsx ubicado en src/main/resources/.
- **Escritura de Resultados (ExcelWriter.java):**
    - La clase ExcelWriter.java permite escribir datos en un nuevo archivo Excel.
    - Se utiliza en AddAndSaveProductsTest.searchAddAndSaveProducts() para cumplir con el requisito de "registrar en un nuevo archivo Excel (outputData.xlsx) los productos que fueron agregados exitosamente al carrito".

7. Estrategia de Validaciones y Aserciones

Las validaciones son fundamentales para determinar el resultado de las pruebas:

- Se utilizan las aserciones proporcionadas por JUnit Jupiter, principalmente Assertions.assertTrue(), al final de los métodos de prueba para verificar que se cumplen las condiciones esperadas (ej. RegisterTest.isRegistrationSuccessful(), ProductPage.isProductFound()).
- **Utilidad Verify.java:** Se ha desarrollado una clase Verify.java que implementa un mecanismo de "soft assertions". Esto permite ejecutar múltiples verificaciones

dentro de un mismo test y, en lugar de detenerse en el primer fallo, recopila todos los errores de aserción. Al final del test, si se encontraron errores, se lanza una única AssertionError combinada que los agrupa. Esto es útil para obtener un panorama más completo de los fallos en tests con múltiples puntos de validación sin una ejecución prematura.

8. Ejecución de Pruebas y Generación de Reportes

- **Ejecución:** Las pruebas se ejecutan utilizando Maven con el comando mvn clean test. El plugin maven-surefire-plugin es el encargado de ejecutar los tests. Es importante asegurar que Surefire esté configurado para JUnit 5 para que los tests escritos con anotaciones de JUnit Jupiter se ejecuten correctamente.
- **Reportes:** Tras la ejecución, Surefire genera reportes detallados en formato XML y HTML en el directorio target/surefire-reports/. Estos reportes sirven como evidencia de la ejecución y los resultados de las pruebas (adaptado de "reportes TestNG" a JUnit, que es lo que usa el proyecto).

9. Distribución de Tareas y Colaboración

El proyecto se abordó de manera colaborativa, distribuyendo el desarrollo de los casos de prueba y los componentes del framework entre los 4 miembros del equipo.

- Desarrollo del flujo de "Registro de Usuario" (RegisterTest.java), incluyendo la lógica de HomePage.goToRegisterPage() y la clase RegisterPage.java. Colaboración en ExcelReader.java.
- Implementación del flujo de "Búsqueda y Agregado al Carrito" y "Escritura de Resultados en Excel" (AddAndSaveProductsTest.java), desarrollando ProductPage.java (parcialmente) y ExcelWriter.java.
- Desarrollo de los flujos "Verificación de Productos en el Carrito" (VerifyCartTest.java) y un segundo escenario de búsqueda/agregado (SelectProductTest.java), implementando CartPage.java y colaborando en ProductPage.java.
- Desarrollo del flujo de "Inicio de Sesión" (creando LoginTest.java y LoginPage.java). Responsable de la creación y mantenimiento de clases base (BaseTest.java, BasePage.java) y utilidades clave como Verify.java y Constants.java.

10. Conclusión

La estrategia de automatización implementada se enfoca en crear un framework robusto, mantenible y escalable utilizando Java, Selenium WebDriver, y el patrón Page Object Model. El uso de Maven facilita la gestión del proyecto, y Apache POI permite un manejo eficiente de datos externos. Las esperas explícitas y las validaciones claras aseguran la fiabilidad de las pruebas. Esta aproximación nos ha permitido cubrir los casos de prueba requeridos de manera efectiva y sienta una base sólida para futuras expansiones de la cobertura de pruebas. Se han seguido las recomendaciones de priorizar legibilidad, mantenibilidad y el uso de aserciones claras.