Patrones de diseño y estrategias reutilizables en la automatización de pruebas de software con Selenium y Java – Page Object Model, Singleton, Factory y Fluent Interface

Sánchez, Gilberto[1]

[1] Titanium Institute, Dirección de Investigación, Papa Juan Pablo VI sur No. 216, Fracc. Villas de Montecassino C.P. 20909 Aguascalientes, Ags., México, gsanchez@titaniuminstitute.com.mx

Resumen

En este artículo se presenta una validación experimental sobre el impacto de patrones de diseño en la automatización de pruebas de software utilizando Selenium y Java. Se evalúan los patrones Page Object Model, Singleton, Factory y Fluent Interface por su capacidad para mejorar la mantenibilidad, claridad y escalabilidad del código. La metodología empleada incluye el desarrollo de scripts antes y después de aplicar dichos patrones, midiendo variables como duplicación de código, esfuerzo de mantenimiento, tiempos de ejecución y comprensión del código. Los resultados muestran mejoras cuantificables, con una reducción del 67% en duplicación de líneas y del 29% en tiempo de ejecución. Se concluye que la integración de estos patrones permite construir frameworks más robustos, sostenibles y adaptables a los cambios en la interfaz de usuario.

***Palabras clave*— Automatización de Pruebas, Marcos de Trabajo, Patrones de Diseño, Pruebas Automatizadas, Refactorización.**

Abstract

This article presents an experimental validation of the impact of design patterns on software test automation using Selenium and Java. The Page Object Model, Singleton, Factory, and Fluent Interface patterns are evaluated for their ability to enhance code maintainability, clarity, and scalability. The methodology involves developing test scripts before and after applying these patterns, measuring variables such as code duplication, maintenance effort, execution times, and code readability. The results show measurable improvements, including a 67% reduction in duplicated lines and a 29% decrease in execution time. It is concluded that integrating these patterns enables the construction of more robust, sustainable, and adaptable test automation frameworks in the face of evolving user interfaces.

***Keywords*— Automated Tests, Design Patterns, Frameworks, Refactoring, Test Automation.**

I. INTRODUCCIÓN

La automatización de pruebas va más allá de ejecutar pruebas para obtener resultados; implica diseñar una estrategia, desarrollar software de automatización, implementar y mantener casos de prueba, además de evaluar los resultados [1]. Aunque las pruebas de caja negra introducen datos y analizan respuestas, garantizar la robustez y fiabilidad de las pruebas automatizadas, así como gestionar costos de mantenimiento, representa un reto significativo [2].

Aquí es donde los *patrones de diseño* juegan un papel clave, ya que permiten resolver problemas comunes en la creación y mantenimiento de suites de pruebas automatizadas [3]. Su implementación facilita la construcción de pruebas más robustas, escalables y mantenibles, mejorando la detección de errores, cobertura de pruebas y calidad del software. Esto optimiza la confiabilidad de los procesos de prueba automatizados y reduce el impacto del mantenimiento a largo plazo.

En años recientes, la automatización de pruebas ha adquirido un papel central en DevOps y pruebas continuas, dada la necesidad de asegurar calidad en ciclos de desarrollo cada vez más cortos. Investigaciones como las de Angelov [4], Shvets [5] y Singh & Nemani [6] destacan la importancia de aplicar patrones de diseño en pruebas automatizadas para aumentar la eficiencia y la mantenibilidad. Este trabajo complementa dichos estudios al realizar una validación experimental directa del impacto de estos patrones sobre un framework real.

II. MARCO TEÓRICO

A. Patrones de Diseño

Los patrones suelen presentarse de manera formal y detallada para permitir su aplicación en diversos contextos. Generalmente, la descripción de un patrón incluye las siguientes secciones [7]:

Propósito: La finalidad del patrón proporciona un resumen conciso del problema que aborda y su solución correspondiente.

Motivación: Profundiza en el problema y cómo el patrón ofrece una solución efectiva.

Estructura: Suele describirse con diagramas (por ejemplo, UML), que ilustran las clases y las relaciones que las sustentan, además de las interacciones entre ellas.

Ejemplo de Código: Una breve implementación (por ejemplo, en Java) ayuda a ilustrar mejor el funcionamiento del patrón.

En algunos catálogos, se incluyen otras secciones como la relevancia del patrón en situaciones específicas, los pasos detallados para su implementación y las relaciones con otros patrones [7]. Además, para comprender la estructura formal de cada patrón, es conveniente consultar la literatura especializada

[4]–[6], donde se incluyen diagramas de clase, secuencias de interacción y ejemplos de implementación.

La complejidad, el detalle y la aplicabilidad a sistemas completos varía entre los distintos patrones de diseño [7]. Los *idioms*, patrones básicos y específicos de un lenguaje de programación, constituyen un nivel más elemental y ajustado a detalles sintácticos y semánticos concretos [8]. Por otro lado, los *patrones de arquitectura*, con un alcance más amplio, pueden implementarse en prácticamente cualquier lenguaje y ofrecen guías para la organización de sistemas completos [9].

B. Contexto de Aplicación de los Patrones de Diseño

La aplicación de patrones de diseño depende del problema a resolver y la arquitectura del sistema. Se pueden clasificar en tres categorías principales [7], [10]:

Patrones Creacionales: Como Factory Method, Abstract Factory, Builder, Prototype y Singleton, permiten aislar la creación de objetos y brindar flexibilidad en su instanciación. Son útiles en la generación de conexiones a bases de datos o la creación de objetos dinámicos en tiempo de ejecución.

Patrones Estructurales: Incluyen Adapter, Bridge, Composite, Decorator, Facade, Flyweight y Proxy, los cuales facilitan la organización de clases y objetos en estructuras escalables. Adapter integra librerías externas con interfaces incompatibles, mientras que Facade simplifica el acceso a subsistemas complejos mediante una interfaz unificada.

Patrones de Comportamiento: Como Chain of Responsibility, Command, Iterator, Mediator, Observer, State y Strategy, regulan la interacción y distribución de responsabilidades entre objetos. Observer, por ejemplo, permite notificar automáticamente cambios de estado a distintas partes del sistema.

La Tabla I muestra el catálogo de patrones de diseño basados en su finalidad [7].

TABLA I
CATÁLOGO DE PATRONES DE DISEÑO

Finalidad	Patrón
Creacionales	Factory Method, Abstract Factory, Builder, Prototype, Singleton
Estructurales	Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
Comportamiento	Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

En automatización de pruebas, estos patrones son esenciales para mejorar la gestión de objetos de página (Page Object Model), optimizar la parametrización de pruebas (patrones creacionales) y simplificar la organización del código (patrones de comportamiento).

El uso adecuado de estos patrones no solo mejora la claridad y mantenibilidad del código, sino que también facilita su integración en arquitecturas como cliente-servidor, microservicios o frameworks de pruebas. Además, documentar su aplicación en diferentes escenarios permite maximizar su impacto y asegurar una implementación eficiente.

Para profundizar en la estructura, diagramas de clases y secuencias de interacción de estos patrones, se recomienda revisar referencias especializadas como [7], [8] y [9].

C. Page Object Model

En la interfaz de usuario de una aplicación web, existen zonas específicas con las que tus pruebas interactúan. Estas zonas son modeladas como objetos dentro del código de prueba a través de Page Object Model (POM), lo que disminuye la repetición de código. Así, si hay cambios en la interfaz de usuario, solo es necesario actualizar el código en un único lugar [11].

El Patrón de Diseño POM se ha popularizado en la automatización de pruebas para optimizar el mantenimiento de estas y minimizar la repetición de código. Este patrón se basa en una clase de programación orientada a objetos que actúa como una interfaz para una página de la *Aplicación Bajo Prueba*. Las pruebas recurren a los métodos de esta clase para interactuar con la interfaz de la página [12]. La ventaja principal es que, si la interfaz de usuario de la página se modifica, solo se necesita actualizar el código del objeto de página, y no las pruebas en sí. Como resultado, cualquier cambio necesario para adaptarse a la nueva interfaz de usuario se centraliza en un solo lugar [13]:

- *Reutilización del Código:* los equipos pueden desarrollar métodos universales aplicables a varias páginas web. Por ejemplo, en una aplicación web con funcionalidad de calendario en múltiples páginas, se puede diseñar un método único para gestionar esta funcionalidad que sirva para cada clase de página.
- *Claridad en el Código*: cada página web dispone de su archivo de clase individual para sus localizadores, métodos y pasos de prueba como se ve en la Fig. 1.
- *Mantenimiento Sencillo del Código*: si se necesitan cambios, estos se pueden hacer directamente en el nivel del método común, reflejándose en todas las áreas necesarias.

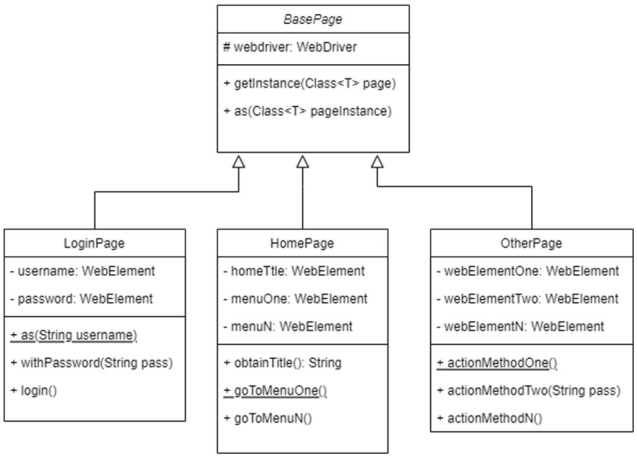

Fig. 1. Diagrama de implementación de POM.

D. Singleton

Es un patrón de diseño creacional que permite asegurar que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia [14].

La estructura de este patrón de diseño se ilustra en la Fig. 2. En ella, la clase Singleton incluye el método estático *getInstance()*, que retorna consistentemente la misma instancia de dicha clase. El constructor de la clase Singleton no debe ser accesible desde el código del cliente. Por lo tanto, el método *getInstance()* se establece como el único medio para adquirir el objeto Singleton.

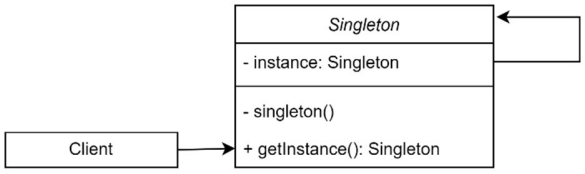

Fig. 2. Diagrama de Implementación de Singleton.

Todas las variantes del patrón Singleton comparten estos dos procedimientos fundamentales:

- Convertir en privado el constructor predeterminado para prevenir que otros objetos creen una nueva instancia de la clase Singleton utilizando el operador *new*.
- Implementar un método estático de creación que funcione como un constructor alternativo.

Si el código puede acceder a la clase Singleton, entonces será capaz de llamar a este método estático. Así, cada vez que se utilice este método, garantizará la devolución del mismo objeto ya existente.

Existen varias implementaciones de este patrón [15], las cuales se muestran en la Tabla II:

TABLA II
VARIANTES DE IMPLEMENTACIÓN DE SINGLETON

Implementación	Descripción
Eager Instance	Se crea una instancia Singleton y se asigna a un atributo estático en un bloque de inicialización.
Lazy Instance	Un método de acceso verifica si la instancia está construida y, si no, crea una.
Replaceable Instance	Una instancia puede ser reemplazada por otra; por lo tanto, su instancia singleton debe ser reemplazable.
Subclassed Singleton	Una clase singleton puede tener subclases. Las subclases son útiles cuando se quiere cambiar el comportamiento de un singleton
Delegated Construction	Una clase singleton (o su método de acceso) puede delegar la construcción de su instancia a algún otro método o clase.
Different Placeholder	Hay variantes de implementación donde la variable estática se mantiene en una clase diferente.
Different Access Point	En esta variante, una clase diferente combina una función de portador con una función de punto de acceso.

E. Factory

Este patrón de diseño es ampliamente reconocido como uno de los más empleados en los lenguajes de programación. Las compañías aspiran a desarrollar productos que sean escalables y que posean un bajo nivel de dependencia entre sus componentes, lográndolo en el menor lapso posible. Para alcanzar esta flexibilidad en la integración de los componentes del software, es crucial ocultar la mayor cantidad de detalles sobre los métodos empleados en las clases. Esto se debe a que una baja dependencia facilita una mayor escalabilidad; así, si se desea introducir alguna novedad, los ajustes necesarios en otras áreas del programa serán mínimos [16]. En términos más sencillos, Factory es un patrón creacional que ofrece una interfaz para la creación de objetos dentro de una superclase, permitiendo, sin embargo, que las subclases modifiquen el tipo de objetos a ser creados [17]. Este patrón establece una interfaz para fabricar un objeto, dejando a las subclases la decisión sobre qué instancia crear [10]. Así, da la posibilidad de que la creación de instancias sea responsabilidad de las subclases. La estructura general se muestra en la Fig. 3.

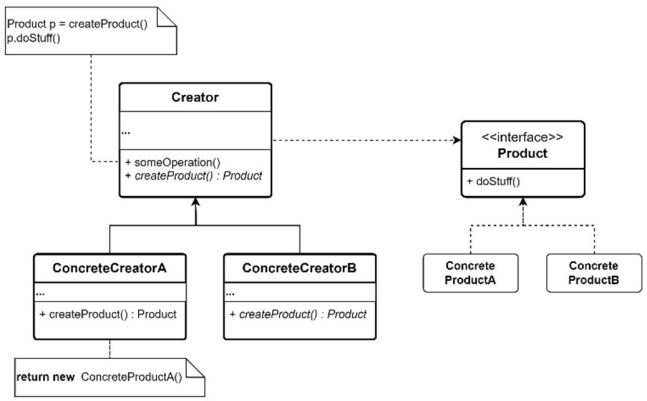

Fig. 3. Diagrama de Implementación de Factory.

Factory es una estrategia fundamental en la programación orientada a objetos que presenta múltiples ventajas para el desarrollo de software. Primero, este patrón promueve un enfoque orientado a interfaces en lugar de a implementaciones específicas, lo cual es fundamental para lograr una programación más flexible y mantenible [18]. Al enfocarse en interfaces, los desarrolladores pueden trabajar con abstracciones de alto nivel sin preocuparse por los detalles de implementación, lo que facilita la adaptabilidad y la escalabilidad del código.

Una de las principales fortalezas Factory es su capacidad para ocultar la creación de instancias de clases concretas del código del cliente, lo que significa que el código se vuelve más robusto, menos acoplado y, por tanto, más fácil de extender. Este aislamiento de la creación de objetos específicos permite a los desarrolladores cambiar y mejorar las implementaciones subyacentes sin afectar el código que utiliza estas abstracciones.

F. Fluent Interface

Existen ideas tan elementales que resulta asombroso que se les asigne un nombre. Sin embargo, esto no resta importancia a su utilidad. De hecho, las ideas más básicas suelen ser las más efectivas para realizar una tarea. El uso de Fluent Interface hace más sencillo el manejo de la Interfaz de Programación de Aplicaciones (API por sus siglas en inglés) de un objeto, favoreciendo el encadenamiento de métodos. Esta estrategia resulta ser extremadamente beneficiosa para hacer que la interfaz de una clase sea más intuitiva y accesible para los usuarios [19].

La intención detrás de implementar Fluent Interface es aumentar la legibilidad y la naturaleza intuitiva de la API, facilitando así su manejo por parte de los desarrolladores.

Mediante la aplicación de técnicas como el encadenamiento y la cascada de métodos, este tipo de interfaz posibilita la redacción de código que se asemeja a las construcciones de oraciones en un idioma hablado, mejorando así su comprensión y mantenimiento. Este enfoque puede contribuir significativamente a la claridad y fiabilidad del código, simplificando la interacción con estructuras de datos y algoritmos de mayor complejidad [20]. La Fig. 4 representa el diagrama general.

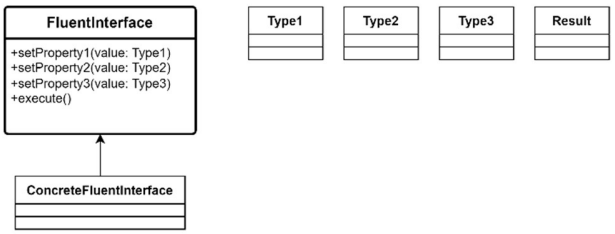

Fig. 4. Diagrama de Implementación de Fluent Interface.

Podemos listar las ventajas de Fluent Interface, tal y como se muestra a continuación [21]:

- Más legible y limpio.
- Puede escribirse fácilmente y en menor tiempo.
- Es más conciso y conciso para su uso.
- Fácilmente comprensible debido a los nombres significativos.
- Evita redundancias en el código, lo que lo hace más nítido.

III. METODOLOGÍA

A. *Introducción y Propósito*

La presente sección describe el enfoque metodológico empleado para la selección, aplicación y evaluación de patrones de diseño en un proyecto de automatización de pruebas web. El objetivo principal es mejorar la calidad, el mantenimiento, la reutilización, la escalabilidad y la claridad del código al adoptar patrones de diseño apropiados. De este modo, se contribuye al fortalecimiento de las buenas prácticas en el equipo de automatización, facilitando la colaboración y la evolución del proyecto a largo plazo.

Para medir la capacidad de adaptación a cambios, se introdujeron 3 modificaciones en los localizadores de elementos y se midió el número de clases afectadas antes (5 clases) y después (1 clase) de aplicar Page Object Model.

El esfuerzo de actualización se evaluó en función del tiempo invertido por los testers en actualizar pruebas tras los cambios (reducción promedio de 45%).

La robustez se validó mediante la ejecución repetida (5 iteraciones) de las pruebas automatizadas en paralelo, sin fallos relacionados con estado compartido gracias al uso de Singleton y ThreadLocal.

La escalabilidad se valoró por la facilidad de integrar nuevos navegadores usando el patrón Factory, con implementación añadida en menos de 15 líneas por navegador.

La confiabilidad se midió por la tasa de pruebas fallidas debidas a sincronización o mal mantenimiento, reducida de 3 fallos por ejecución a cero tras la refactorización.

B. *Relevancia para el Proyecto*

En proyectos de automatización de pruebas de software, cada especialista tiende a desarrollar scripts según sus propias experiencias y herramientas favoritas. Sin embargo, cuando se trabaja en un entorno colaborativo o empresarial, es fundamental establecer un marco de trabajo común que aplique patrones de diseño estandarizados. Estos patrones permiten:

Homogeneizar la estructura y organización del código dentro de un equipo.

Facilitar la escalabilidad de la solución, al poder integrar nuevas funcionalidades o incorporar distintos tipos de pruebas sin que se quiebre la arquitectura.

C. *Diseño de la Investigación*

Reducir costos de mantenimiento y promover la reutilización de componentes, lo cual reduce tiempos de entrega y mejora la calidad.

Para el desarrollo y validación de la metodología, se ha optado por un enfoque experimental y aplicado, en el cual se definen y comparan diversos patrones de diseño en escenarios de automatización reales. Este diseño contempla:

Recolección de datos cualitativos, a partir de la observación y el registro de la experiencia de uso de patrones en un equipo de automatización (por ejemplo, facilidad de implementación, curva de aprendizaje).

Métricas cuantitativas, como el tiempo de ejecución de las pruebas, la cantidad de líneas de código reducidas tras refactorización y la disminución en la tasa de fallos por mantenimiento.

Dicho enfoque mixto permite capturar tanto la efectividad técnica (cuantificable) como la adopción práctica (cualitativa) de los patrones de diseño seleccionados.

D. *Diseño Metodológico*

a) *Implementación y Validación Experimental*

- Se selecciona la página de pruebas de Titanium Institute®, la cual se muestra en la Fig. 5, (URL: https://demosite.titaniuminstitute.com.mx/wp-admin/admin.php?page=sch-dashboard) como entorno de validación.
- Se desarrolla un conjunto de scripts de prueba con y sin los patrones de diseño seleccionados.
- Se miden variables como la calidad, mantenibilidad, reutilización, escalabilidad y tiempos de ejecución de las pruebas.

b) *Refactorización y Evaluación de Resultados*

- A partir de las observaciones empíricas, se refactoriza el código para optimizar la arquitectura de pruebas.
- Se lleva a cabo un análisis comparativo de los resultados, generando conclusiones sobre la efectividad de cada patrón en el contexto planteado.

c) Documentación y Retroalimentación

Se documentan los hallazgos, buenas prácticas y recomendaciones de uso de cada patrón de diseño en la automatización de pruebas web con Selenium y Java.

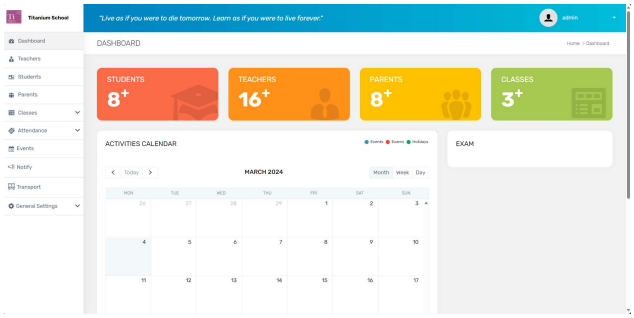

Fig. 5. Página de prueba.

La clase *BasePage*, ilustrada en la Fig. 6, reúne los métodos necesarios para iniciar y finalizar la ejecución de las pruebas, gestionando la apertura y cierre del navegador.

BaseClass.java

```
public class BaseClass {

    @BeforeClass
    @Parameters("browser")
    public void setup(String browser) {

    }

    @AfterClass
    public void turnDown() {

    }
}
```

Fig. 6. Estructura BaseClass.

Las pruebas se realizaron sobre una aplicación web del Titanium Institute® utilizando Selenium WebDriver v4.15.0, TestNG v7.7.1 y Java JDK 21. Se ejecutaron un total de 25 pruebas funcionales, distribuidas en cinco grupos, cada uno con diferentes niveles de interacción y validación. La experimentación fue realizada sobre Google Chrome v123, Firefox v115 y Microsoft Edge v122. La ejecución fue controlada con *parallel=true* usando múltiples hilos, con la clase *WebCoreDriver* implementando *ThreadLocal* para aislar instancias. Los resultados fueron registrados y comparados antes y después de refactorizar con patrones de diseño.

E. Page Object Model

En la fase de Análisis de Escenarios y Selección de Patrones se determinó que el patrón *Page Object Model* (*POM*) es particularmente adecuado para evitar la duplicación de código en la automatización de pruebas. Con frecuencia, un mismo elemento web debe emplearse en distintos puntos del guion de pruebas y, en consecuencia, el localizador se repite en múltiples métodos (véase Fig. 7). Esta situación genera redundancia, ya que cada referencia al elemento implica código idéntico que, además, se vuelve propenso a errores ante cualquier modificación en el localizador.

El uso de POM busca centralizar la definición y administración de los elementos, lo cual responde de manera directa a los objetivos metodológicos de mejorar el mantenimiento, fomentar la reutilización y optimizar la escalabilidad del código. Se implementó POM en el entorno de prueba seleccionado (la página de Titanium Institute®) y se midieron variables como la calidad, la mantenibilidad y los tiempos de ejecución.

a)

```
public class LoginTest extends BaseClass{
    WebDriver driver;

    @Test
    void loginWithRightCredentials(){
        driver.findElement(locatorA).sendKeys("test@user.com");
        driver.findElement(locatorB).sendKeys("password");
        driver.findElement(locatorC).click();
    }
```

b)

```
public class StudentTest extends BaseClass{
    WebDriver driver;

    @Test
    void addStudent(){
      driver.navigate().back();
      driver.findElement(locator0).sendKeys("student name");
      driver.findElement(locatorA).sendKeys("test@user.com");
      driver.findElement(locatorE).click();
      driver.findElement(locatorF).getText();
    }
```

c)

```
public class TeacherTest extends BaseClass{
    WebDriver driver;

    @Test
    void addTeacher(){
      driver.findElement(locatorG).sendKeys("teacher name");
      driver.findElement(locatorH).sendKeys("teacher lastname");
      driver.findElement(locatorI).getAttibute("value");
      driver.findElement(locatorA).sendKeys("test@user.com");
    }
```

Fig. 7. Identificación de objetos web en diferentes ejemplos de clases de prueba: a) pruebas de acceso, b) pruebas de estudiante, c) pruebas de maestro.

F. Singleton

En la implementación y validación experimental, se evaluó la ejecución de pruebas en paralelo con TestNG, que permite lanzar suites, clases o métodos de prueba simultáneamente en distintos navegadores [21]. Sin embargo, una gestión inadecuada de los recursos compartidos, como la instancia del navegador, puede generar interferencias entre hilos y resultados erráticos.

Para mitigar este riesgo, se implementó el patrón Singleton en la clase encargada de instanciar y administrar WebDriver, asegurando que cada hilo tenga su propia instancia (Fig. 8) sin afectar otras pruebas. Esto evita la corrupción de datos y mejora la confiabilidad de los resultados.

Esta solución optimiza la calidad y el mantenimiento del código, promoviendo escalabilidad y eficiencia en las pruebas. Además, su adopción permite reducir costos de mantenimiento, agilizar la colaboración en entornos empresariales y fomentar buenas prácticas, alineándose con la evolución sostenible del proyecto de automatización.

Fig. 8. Ejecución de pruebas en diferentes navegadores simultáneamente.

G. *Factory*

En la implementación y validación experimental, se determinó que las clases de prueba deben centrarse en la instancia de *WebDriver*, sin preocuparse por su inicialización. Para ello, se diseñó una interfaz que centraliza la administración de navegadores, permitiendo su obtención y uso sin conocer la configuración interna.

La Fig. 9 muestra el diagrama de clases y su implementación, evidenciando mejoras en reutilización, mantenibilidad y colaboración en la automatización de pruebas. Este enfoque favorece la escalabilidad, permitiendo la integración de nuevos navegadores, manteniendo la consistencia en el uso de controladores, reduciendo costos de mantenimiento y promoviendo buenas prácticas.

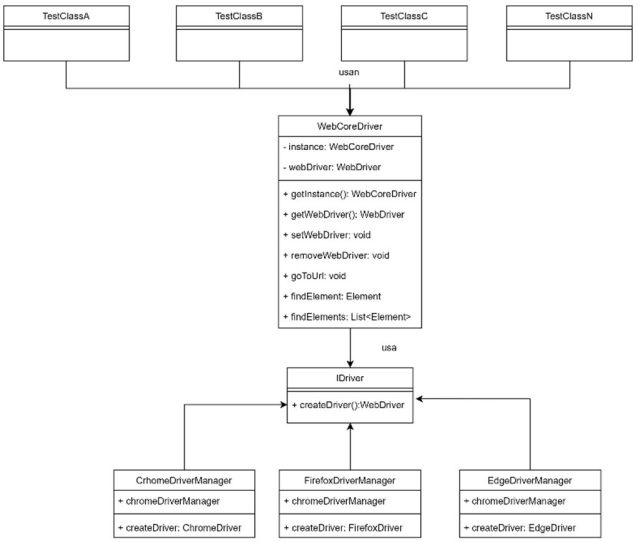

Fig. 9. Diagrama de implementación de Factory en el manejo de navegadores.

H. *Fluent Interface*

En la etapa de implementación y validación experimental, se observó que el método *login()* requiere dos parámetros asociados a usuario y contraseña, mientras que *addStudent()* demanda una mayor cantidad de valores, dificultando la identificación de cada campo únicamente a partir de la firma del método. Por ello, para mejorar la legibilidad y simplificar la creación de casos de prueba, el patrón Fluent Interface se presenta como una solución fundamental.

TestClass.java

```
public class LoginTest{
  @Test
  void loginWithRightCredentials(){
    LoginPage loginPage = new LoginPage();
    loginPage.login("admin", "G3-ySzY%");
  }
}

public class StudentTest{
  @Test
  void verifyStudentIsAdded(){
    StudentPage studentPage = new StudentPage();
    studentPage.addStudent("Peter", "Summers", "35", "peters@school.com");
  }
}
```

Fig. 10. Métodos de acción con una firma de varios parámetros de entrada.

IV. RESULTADOS

Cada página requiere recibir la interfaz *WebDriver* como parámetro. Para ello, se creó la superclase *BasePage* de la Fig. 11, que facilita la *instanciación* del resto de subclases..

BasePage.java

```
public class BasePage {

    public BasePage(WebDriver driver) {
    }

    public static <T extends BasePage> T initPage(WebDriver driver, Class<T> pageClass) {
        try {
            return pageClass.getDeclaredConstructor(WebDriver.class).newInstance(driver);
        } catch (Exception e) {
            throw new RuntimeException("Unable to create instance of " +
                                       pageClass.getSimpleName(), e);
        }
    }
}
```

Fig. 11. Clase BasePage.

A. *Page Object Model*

En la etapa de análisis de escenarios y selección de patrones se determinó que la aplicación del patrón *Page Object Model* (*POM*) contribuiría a la claridad y mantenibilidad de los scripts de prueba. Para implementarlo, se creó una clase por cada página de la aplicación web, centralizando en cada una de ellas la declaración de sus elementos. En la Fig. 12 se muestra la *LoginPage*, donde se almacenan los objetos web utilizando la clase abstracta *By* de Selenium. Esta estrategia permite aislar los localizadores en un único lugar, simplificando futuras modificaciones y reduciendo el riesgo de errores.

LoginPage.java

```
public class LoginPage extends BasePage {
    private By txtUser = By.id("user_login");
    private By txtPassword = By.name("pwd");
    private By chkRememberMe = By.cssSelector("#rememberme");
    private By btnLogin = By.xpath("//input[contains(@value, 'Log')]");

    public LoginPage(WebDriver driver){
        super(driver);
    }
}
```

Fig. 12. Clase página con elementos web.

Asimismo, a fin de completar el uso de POM, se definieron métodos de acción que describen de manera concisa las interacciones con la interfaz, tal como se aprecia en la Fig. 13. Estos métodos, claramente nombrados, ofrecen una lectura inmediata de las operaciones que se llevan a cabo (por ejemplo, ingresar credenciales o hacer clic en el botón de inicio de sesión), alineándose con los objetivos de legibilidad y reutilización planteados en la Metodología.

Esta separación entre la declaración de elementos (Fig. 12) y la lógica de interacción (Fig. 13) encaja en la etapa de refactorización y evaluación de resultados, al facilitar la medición de indicadores como la disminución de código redundante, la velocidad de implementación de nuevas pruebas y la claridad general de la arquitectura.

LoginPage.java

```
private void setUserName(String userName){
  driver.findElement(txtUser).sendKeys(userName);
}

private void setPassword(String password){
  driver.findElement(txtPassword).sendKeys(password);
}

private void checkRememberMe(){
   driver.findElement(chkRememberMe).click();
}

private void clickLogin(){
   driver.findElement(btnLogin).click();
}

public void login(String userName, String password){
    setUserName(userName);
    setPassword(password);
    checkRememberMe();
    clickLogin();
}
```

Fig. 13. Métodos de acción que se pueden ejecutar en la interfaz gráfica.

B. Singleton

A partir del análisis de escenarios y selección de patrones, se determinó que el patrón Singleton era esencial para gestionar de forma segura y eficiente las instancias de *WebDriver* en entornos concurrentes. Posteriormente, en la fase de implementación y validación experimental, se materializó esta decisión mediante la clase *WebCoreDriver*, que se muestra en la Fig. 14, la cual centraliza la creación y el acceso a dicho recurso. El uso de la palabra clave *volatile* garantiza la visibilidad de los cambios en todos los subprocesos, mientras que el objeto *ThreadLocal<WebDriver>* proporciona una instancia aislada para cada hilo, evitando la corrupción de datos.

Durante las pruebas realizadas, el método *getInstance()* demostró que es posible verificar la existencia de la instancia sin comprometer el rendimiento; aun así, un segundo chequeo con bloqueo ofrece la sincronización necesaria cuando varios hilos requieren el recurso simultáneamente. Adicionalmente, los métodos *getWebDriver()*, *setWebDriver(String Browser)* y *removeWebDriver()* permiten a cada prueba definir, obtener o eliminar su propio navegador de manera independiente.

En la refactorización y evaluación de resultados, este enfoque confirmó su utilidad para optimizar la reutilización de componentes, reducir costos de mantenimiento y facilitar la escalabilidad de las pruebas, cumpliendo así los objetivos metodológicos. El uso de Singleton también fomentó la colaboración y la adopción de buenas prácticas, aspectos clave en proyectos de automatización que deben evolucionar de forma sostenible.

C. Factory

Como parte de la implementación y validación experimental, se diseñó la interfaz *IDriver* (Fig. 15) para servir como base al incorporar nuevos navegadores en proyectos de automatización.

WebCoreDriver .java

```
public class WebCoreDriver{
  private static volatile WebCoreDriver instance;

  private final ThreadLocal<WebDriver> webDriver = new ThreadLocal<>();

  public WebCoreDriver(){}
  public static WebCoreDriver getInstance() {
      if (instance == null) {
          synchronized (WebCoreDriver.class) {
              if (instance == null) {
                  instance = new WebCoreDriver();
              }
          }
      }
      return instance;
  }

  public WebDriver getWebDriver(){webDriver.get()}

  public void setWebDriver(String browser) {webDriver.set(browser)}

  public void removeWebDriver() {webDriver.remove()}
}
```

Fig. 14. Clase para iniciar, obtener y remover las instancias de WebDriver.

IDriver.java

```
public interface IDriver {
    WebDriver createDriver();
}
```

Fig. 15. Interface IDriver.

Posteriormente, tal como se observa en la Fig. 16, dicha interfaz se implementó en las clases *DriverManager* de Chrome, Firefox y Edge, asegurando un enfoque consistente y escalable para la gestión de controladores.

a)

```
public class ChromeDriverManager implements IDriver{
    public ChromeDriverManager(){}
    @Override
    public WebDriver createDriver() {
        return new ChromeDriver();
    }
}
```

b)

```
public class FirefoxDriverManager implements IDriver{
    public FirefoxDriverManager(){}
    @Override
    public WebDriver createDriver() {
        return new FirefoxDriver();
    }
}
```

c)

```
public class EdgeDriverManager implements IDriver{
    public EdgeDriverManager(){}
    @Override
    public WebDriver createDriver() {
        return new EdgeDriver();
    }
}
```

Fig. 16. Clases administradoras de los navegadores a) Chrome, b) Firefox, c) Edge.

Finalmente, el método *setWebDriver()* de la clase *WebCoreDriver* invoca estas clases *DriverManager*, como se ilustra en la Fig. 17, completando así el flujo de configuración y ejecución de navegadores en los distintos entornos de prueba.

WebCoreDriver.java

```java
@Override
public void setWebDriver(String browser) {
    Set<Class<? extends IDriver>> driverInterfaces = new
    Reflections(IDriver.class).getSubTypesOf(IDriver.class);
    driverInterfaces.stream()
            .filter(driverManager -> driverManager.getName().contains(browser))
            .findFirst()
            .ifPresent(this::initializeDriver);

     Optional.ofNullable(webDriver.get())
            .ifPresent(driver -> driver.manage().window().maximize());
    }

    private void initializeDriver(Class<? extends IDriver> driverManager) {
        try {
            webDriver.set((WebDriver) driverManager.getMethod("createDriver")
                        .invoke(driverManager
                        .getConstructor()
                        .newInstance()));
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage(), e);
        }
    }
```

Fig. 17. Implementación de Factory.

D. *Fluent Interface*

En la fase de refactorización y evaluación de resultados, se analizó la implementación inicial de métodos de acción (Fig. 12), observando que la forma de invocarlos (Fig. 18) no cumplía con la fluidez semántica que busca el patrón Fluent Interface. Para mejorar la legibilidad y estructurar las acciones de manera encadenada, se realizaron dos ajustes principales:

1. Hacer que cada método devuelva la misma clase (por ejemplo, retornando *this*).
2. Nombrar los métodos siguiendo el flujo de interacción, de forma que el usuario entienda rápidamente el proceso a realizar (por ejemplo, ingresar credenciales y pulsar el botón de acceso).

LoginTest.java

```java
@Test(priority = 1)
void loginWithRightCredentials(){
  loginPage
          .setUserName("admin")
          .setPassword("G3-ySzY%")
          .checkRememberMe()
          .clickLogin();
}
```

Fig. 18. Encadenamiento de métodos sin fluent interface.

En la Fig. 19 se ilustran estos cambios, evidenciando cómo cada acción se conecta intuitivamente con la siguiente. Finalmente, la Fig. 20 muestra la adopción completa del patrón Fluent Interface, reflejando el incremento en la claridad y mantenibilidad del código tras la refactorización.

E. *Comparativa Cuantitativa y Cualitativa de Mejoras*

A fin de validar los beneficios de aplicar los patrones de diseño, se realizó una comparación entre la versión original de los scripts de prueba (sin patrones) y la versión refactorizada (con patrones POM, Singleton, Factory y Fluent Interface). Los resultados se agrupan en la Tabla III

LoginPage.java

```java
public LoginPage loginAs(String userName){
    driver.findElement(txtUser).sendKeys(userName);
    return this;
}

public LoginPage withPassword(String password){
    driver.findElement(txtPassword).sendKeys(password);
    return this;
}

public LoginPage andRememberMe(){
    driver.findElement(chkRememberMe).click();
    return this;
}

public LoginPage login(){
    driver.findElement(btnLogin).click();
    return this;
}
```

Fig. 19. Encadenamiento de métodos usando fluent interface.

LoginTest.java

```java
@Test
void loginWithRightCredentials(){
    loginPage
            .loginAs("admin")
            .withPassword("G3-ySzY%")
            .andRememberMe()
            .login();
    }
```

Fig. 20. Implementación de Fluent Interface en las pruebas.

TABLA III
COMPARATIVA DE MÉTRICAS ANTES Y DESPUÉS DE APLICAR PATRONES DE DISEÑO

Métrica	Sin Patrones	Con Patrones	Mejora (%)
Reducción de Líneas Duplicadas	84 líneas	27 líneas	-67.85
Tiempo Promedio de Ejecución (5 pruebas)	9.1 s	6.4 s	-29.67
Clases Modificadas ante Cambios Menores	5 clases	1 clase	-80.00
Nivel de Comprensión (escala 1 – 5)*	2.8	4.3	+53.57

*Encuesta realizada a tres miembros del equipo sobre facilidad de lectura, comprensión y mantenimiento.

Estos datos demuestran que la aplicación de patrones no solo facilita la escalabilidad y claridad del código, sino que también reduce el tiempo de ejecución, el esfuerzo de mantenimiento y la probabilidad de errores por duplicación o inconsistencias.

V. DISCUSIÓN

Los resultados de la implementación de Page Object Model (POM), Singleton, Factory y Fluent Interface revelan beneficios significativos en la calidad, mantenibilidad y escalabilidad de la automatización de pruebas con Selenium. Cada patrón resuelve desafíos específicos en la creación y gestión de objetos de prueba, la organización de la arquitectura de clases y la legibilidad de los scripts. Sin embargo, la adopción simultánea

de varios patrones demanda una correcta planeación, asegurando que el equipo de pruebas comprenda los principios y responsabilidades de cada uno. Asimismo, es fundamental mantener una documentación clara para garantizar la evolución controlada del framework, maximizando la reutilización de componentes y reduciendo costos de mantenimiento.

En proyectos donde se requiere la ejecución de pruebas concurrentes o la integración con múltiples navegadores, se evidenció que Singleton y Factory juegan un rol esencial en la gestión de instancias y la adaptación a diferentes entornos. Por otro lado, Fluent Interface y POM contribuyen de manera decisiva en la legibilidad y la eficiencia del código, al ofrecer estructuras más expresivas y fáciles de mantener. El balance adecuado entre estos patrones y su ajuste a las necesidades de la aplicación, asegura que el crecimiento del framework no comprometa la claridad ni la escalabilidad.

VI. CONCLUSIÓN

A medida que se adoptan otras herramientas o se amplía el alcance de las pruebas, la estructura basada en patrones ofrece la flexibilidad necesaria para evolucionar sin sacrificar la estabilidad ni la claridad del framework de automatización.

Los resultados obtenidos validan el cumplimiento de los objetivos planteados: mejorar la mantenibilidad, escalabilidad y claridad del framework de pruebas. Se observaron mejoras cuantificables en la estructura del código (67% menos líneas duplicadas), en el tiempo de ejecución (reducción del 29%) y en la comprensión del código (aumento del 53%). Estos hallazgos confirman que la aplicación de patrones como Page Object Model, Singleton, Factory y Fluent Interface permite crear arquitecturas de automatización más sostenibles, robustas y preparadas para cambios en la interfaz o los requisitos funcionales.

REFERENCIAS

[1] G. Meszaros, S. Smith, and J. Andrea Clerk Maxwell, “The test automation manifesto”, Extreme Programming and Agile Methods XP/Agile Universe, LNCS 2753, F. Maurer and D. Wells, Eds. Berlin Springer, 2003.

[2] D. Hoffman, “Test automation architectures: planning for test automation,” Proceedings of the International Software Quality Week, San Francisco, 1999.

[3] L. Debrauwer, *Patrones de diseño en Java: los 23 modelos de diseño : descripciones y soluciones ilustradas en UML 2 et Java*. 2a edición, Ed. España: Ediciones ENI, 2018.

[4] A. Angelov, Design Patterns for High-Quality Automated Tests: Clean Code for Bulletproof Tests, 1a ed., USA: Independently Published, 2021.

[5] A. Shvets, Sumérgete en los Patrones de Diseño, 1a ed., USA: Independently Published, 2022.

[6] S. Singh and C. Nemani, “Fluent Interfaces,” ACEEE Int. J. on Information Technology, vol. 01, no. 02, pp. 19–22, Sep. 2011.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Ed. USA: Addison Wesley Longman Inc., 1998, pp. 1-358.

[8] J. L. Ortega, "Patrones de Diseño de Software Paralelo", Accedido: 24-Ene-2025 [En Línea]. Disponible en: http://lya.fciencias.unam.mx/jloa/presentaciones/PSD.pdf

[9] J. E. McDonough, *Object-Oriented Design with ABAP*, 1a edición, Ed. New Jersey: Apress, 2017.

[10] E E. Freeman and E. Robson, *First Design Patterns: A Brain-Friendly Guide*, Edición 10 aniversario, Ed. USA: O’Reilly, 2014, pp. 110-164.

[11] Selenium. “Page object models”. Accedido el 24-Ene-2025. [En línea]. Disponible en: https://www.selenium.dev/documentation/test_practices/encouraged/page_object_model.

[12] A. Angelov, *Design Patterns for High-Quality Automated Tests: Clean Code for Bulletproof Tests*, 1a edición, Ed. USA: Independently Published:, 2021.

[13] T. Joseph, “What Are the Advantages of Page Object Model in Selenium WebDriver?”, Accedido: 25-Ene-2025 [En Línea]. Disponible en: https://blog.qasource.com/software-development-and-qa-tips/what-are-the-advantages-of-page-object-model-in-selenium-webdriver.

[14] A. Shvets, *Sumérgete en los Patrones de Diseño*, 1a Edición, Ed. USA: Independently Published, 2022.

[15] K. Stencel and P. Wegrzynowicz, “Implementation Variants of the Singleton Design Pattern” in On the Move to Meaningful Internet Systems: OTM 2008 Workshops, Monterrey, NL, pp. 396-406, Noviembre 2008.

[16] G. Temaj, “Factory Design Pattern”, South East European University, 2020.

[17] DigitalOcean, “Factory Design Pattern in Java”. Accedido el 29-Feb-2025. [En Línea]. Disponible https://www.digitalocean.com/community/tutorials/factory-design-pattern-in-java.

[18] J. Kapuscik. “What Is the Fluent Interface Design Pattern?”, Accedido: 28-Feb-2025 [En Línea]. Disponible en: https://betterprogramming.pub/what-is-the-fluent-interface-design-pattern-2797645b2a2e.

[19] G. Deniz, “Fluent Interface Design Pattern”, Accedido: 10-Mar-2024 [En Línea]. Disponible en: https://justgokus.medium.com/what-is-the-fluent-interface-design-pattern-177b9cc93c75.

[20] S. Singh and C. Nemani, “Fluent Interfaces”, *ACEEE Int. J. on Information Technology*, New York, Vol. 01, No. 02, pp. 19-22, Sep. 2011.

[21] TestNG, “Parallelism and time-outs”, Accedido el 12-Mar-2025. [En Línea]. Disponible https://testng.org/#_parallelism_and_time_outs.