Principios SOLID en la Automatización de Pruebas de Software para Interfaces de Usuario Web con Selenium WebDriver y Java.

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

SOLID es un acrónimo que encapsula cinco principios de diseño de software orientado a objetos (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation y Dependency Inversion), promulgado por Robert C. Martin, cuyo enfoque es crear sistemas más comprensibles, flexibles y mantenibles. La adhesión a los principios SOLID durante el diseño de frameworks de pruebas en Selenium y Java no solo optimiza la calidad del código, sino que también facilita la detección de errores en las fases tempranas, reduce los costos de mantenimiento y mejora la eficiencia en el ciclo de vida del desarrollo de software al proporcionar un enfoque sistemático y escalable para la construcción de sistemas de pruebas automatizadas. En el presente trabajo, se muestra cómo poder implementar los principios anteriormente mencionados y como estos pueden ayudar a mejorar nuestro código, aportando un mejor entendimiento, mantenibilidad y escalabilidad.

***Palabras clave*— Automatización de Pruebas, Marcos de Trabajo, Modelo de Objeto Página, Principios SOLID, Refactorización.**

Abstract

SOLID is an acronym that encapsulates five principles of object-oriented software design (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion), promulgated by Robert C. Martin, which aims to create systems that are more understandable, flexible, and maintainable. Adhering to the SOLID principles during the design of testing frameworks in Selenium and Java not only optimizes code quality but also facilitates the detection of errors in early stages, reduces maintenance costs, and enhances efficiency in the software development lifecycle by providing a systematic and scalable approach for building automated testing systems. The present work demonstrates how to implement the principles mentioned above and how these can assist in improving our code, contributing to better understanding, maintainability, and scalability..

***Keywords*—Frameworks, Page Object Model, Refactoring, SOLID Principles, Test Automation.**

I. INTRODUCCIÓN

En el evolutivo contexto de la ingeniería de software, la automatización de pruebas emerge como un pilar crucial para asegurar la fiabilidad, eficiencia y robustez de las aplicaciones. Este ámbito, rico en técnica y estrategia, ha visto la adopción y adaptación de diversas metodologías para potenciar la calidad y eficacia de los procesos de validación automatizada [1]. Entre estos, los principios SOLID, que se muestra su definición en la Tabla 1, originariamente concebidos para el diseño y desarrollo de software orientado a objetos, han establecido un marco que potencia la creación de software más comprensible, flexible y mantenible [2]. La adaptación de estos principios en la automatización de pruebas de software, con un enfoque particular en pruebas sobre Interfaces Gráficas de Usuario Web, empleando Selenium WebDriver y Java como herramientas cardinalmente representativas en la esfera de testing automatizado.

TABLA I
PRINCIPIOS SOLID

Inicial	Principio
S	Single Responsibility (Responsabilidad Única)
O	Open-Close (Abierto-Cerrado)
L	Liskov Substitution (Sustitución de Liskov)
I	Interface Segregation (Segregación de Interfaces)
D	Dependency Inversion (Inversión de Dependencia)

La investigación busca dilucidar cómo estos principios, aun estando firmemente arraigados en el desarrollo de software, pueden ser hábilmente aplicados en la arquitectura de marcos de trabajo (*frameworks* por su traducción al inglés) de pruebas automatizadas para fortalecer su cohesión, reducir su acoplamiento y facilitar su mantenimiento y escalabilidad. En este camino, se exploran casos prácticos, desafíos y soluciones, proporcionando un recurso integral para profesionales y académicos que buscan amalgamar las virtudes del diseño de software y las pruebas automatizadas en un entorno tan dinámico y desafiante como lo es el desarrollo web contemporáneo.

II. MARCO TEÓRICO

A. Single Responsibility Principle (SRP)

Este principio se enfoca en asignar a cada clase un propósito claro y simple. En diversas situaciones, se sitúa un método reutilizable que no guarda relación alguna con la clase, ya que simplemente facilita un uso particular. La complicación aparece cuando es necesario emplear ese mismo método desde otra clase. Si no se realiza una refactorización en ese instante y se crea una clase dedicada para esa función específica que debería ser su responsabilidad, surgen problemas [2]. Robert C. Martin nos dice que: *"Una clase debe tener una y solo una razón para*

cambiar." [3].

B. Open-Closed Principle (OCP)

Entre todos los fundamentos del diseño basado en objetos, este prevalece como el más crucial. Se deriva de las obras de Bertrand Meyer [3] y establece lo siguiente: *"Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas a la extensión, pero cerradas a la modificación".*

Esta es una entidad que puede permitir que se cambie su comportamiento sin alterar su código fuente.

Aunque pueda sonar paradójico, hay múltiples estrategias para implementar el OCP de manera efectiva. Todas ellas se fundamentan en la abstracción, que, de hecho, se constituye como el pilar del OCP.

C. Liskov Substitution Principle (LSP)

Este principio fue creado por Barbara Liskov [4] en su conferencia magistral *"Data Abstraction"* en 1987, la cual menciona que "Los *tipos derivados deben ser completamente sustituibles por sus tipos base"*. Establece que los objetos de una superclase deben ser reemplazables por objetos de sus subclases sin romper la aplicación. Amplía el OCP y permite reemplazar objetos de una clase principal con objetos de una subclase sin interrumpir la aplicación.

Esto requiere que todas las subclases se comporten de la misma manera que la clase principal. Para lograrlo, sus subclases deben seguir estas reglas [5]:

1. No implementar reglas de validación más estrictas en los parámetros de entrada que las implementadas por la clase principal.
2. Aplicar al menos las mismas reglas a todos los parámetros de salida que aplica la clase principal.

En términos sencillos, se aspira a que los objetos de las subclases actúen de la misma forma que los objetos de la superclase. Así, la próxima vez que, por un error, se cree un objeto de la subclase en vez de la superclase, todavía debería estar bien desde la perspectiva del LSP. Un método sobrescrito de una subclase debe aceptar los mismos valores de parámetros de entrada que el método de la superclase. A los valores de retorno del método se les aplican normas parecidas.

D. Interface Segregation Principle (ISP)

Este principio fue definido por primera vez por Robert C. Martin como: *"No se debe* obligar *a los clientes a depender de interfaces que no utilizan".* El objetivo de este principio es reducir los efectos secundarios del uso de interfaces más grandes al dividir las interfaces de las aplicaciones en otras más pequeñas. Es similar al principio de responsabilidad única, donde cada clase o interfaz tiene un único propósito [6].

La creación detallada del diseño de la aplicación y una adecuada abstracción son fundamentales para el principio de segregación de interfaces. Aunque involucre más tiempo y esfuerzo en la etapa de diseño de una aplicación y posiblemente incremente la complejidad del código, finalmente resulta en un código flexible.

E. Dependency Inversion Principle (DIP)

R. C. Martín define este principio como *"Depende de las abstracciones. No dependas de concreciones"* [3]. El propósito de una arquitectura Orientada a Objetos es minimizar la cantidad de modificaciones en un módulo, idealmente, no incurrir en cambios. Una fuente primaria de cambio es una implementación concreta de un módulo, esto es, un tipo específico o una ejecución particular [7]. Comúnmente, un módulo de nivel superior poseerá dependencias de implementaciones de nivel inferior sin que sea estrictamente necesario conocer.

Cuando una clase tiene conocimiento explícito del diseño y ejecución de otra, las modificaciones en una clase amplifican el riesgo de afectar a la otra. Estos cambios pueden provocar repercusiones en cascada a través de toda la aplicación, volviéndola frágil [8]. Para esquivar este tipo de inconvenientes, se debe redactar un "código sólido" que esté ligeramente acoplado y, para apoyarlo, es posible emplear el principio de inversión de dependencia.

F. Selenium WebDriver

En la página oficial de Selenium, encontramos que ellos lo definen como: *"Selenium automatiza los navegadores. ¡Eso es todo! Lo que hagas con ese poder depende totalmente de ti"* [9]. Principalmente es para automatizar aplicaciones web con fines de prueba, pero ciertamente no se limita solo a eso. Las aburridas tareas de administración basadas en web también pueden (y deben) automatizarse.

La suite de Selenium se conforma de tres herramientas, las cuales se muestran en la Fig. 1. Entre estas herramientas, WebDriver es la más popular debido a su soporte y que se puede codificar con varios lenguajes de programación, dando una versatilidad a la hora de crear pruebas automatizadas.

Fig. 1. Selenium suite.

WebDriver controla un navegador de forma nativa, como lo haría un usuario, ya sea localmente o en una máquina remota utilizando el servidor de Selenium, y marca un salto adelante en términos de automatización del navegador.

Selenium WebDriver se refiere tanto a los lenguajes de programación como a las implementaciones del código de control del navegador individual. Esto se conoce comúnmente simplemente como WebDriver [9].

Selenium WebDriver es una recomendación del W3C:

- WebDriver está diseñado como una interfaz de programación sencilla y más concisa.
- WebDriver es una API compacta orientada a objetos.
- Maneja el navegador de manera efectiva.

G. Java

Java es un lenguaje de programación y una plataforma informática lanzado por primera vez por Sun Microsystems en

1995, su logo se muestra en la Fig. 2. Ha evolucionado desde sus humildes comienzos hasta impulsar una gran parte del mundo digital actual, proporcionando una plataforma confiable sobre la cual se construyen muchos servicios y aplicaciones. Los productos nuevos e innovadores y los servicios digitales diseñados para el futuro también siguen confiando en Java.

Si bien la mayoría de las aplicaciones Java modernas combinan el tiempo de ejecución de Java y la aplicación, todavía hay muchas aplicaciones e incluso algunos sitios web que no funcionarán a menos que tenga instalado un Java de escritorio [10].

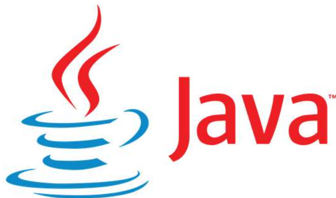

Fig. 2. Logo de Java.

H. Modelo de Objeto Página (POM)

Modelo de Objeto Página (POM por sus siglas en inglés) es un patrón de diseño que se ha vuelto popular en la automatización de pruebas para mejorar el mantenimiento de las pruebas y reducir la duplicación de código. Un objeto página es una clase orientada a objetos que sirve como interfaz para una página de la *Aplicación Bajo Prueba*. Luego, las pruebas utilizan los métodos de esta clase objeto página siempre que necesitan interactuar con la interfaz de usuario de esa página como se muestra en la Fig. 3 . El beneficio es que si la interfaz de usuario cambia, no es necesario cambiar las pruebas en sí, solo es necesario cambiar el código dentro del objeto página. Posteriormente, todos los cambios para admitir esa nueva interfaz de usuario se ubican en un solo lugar.

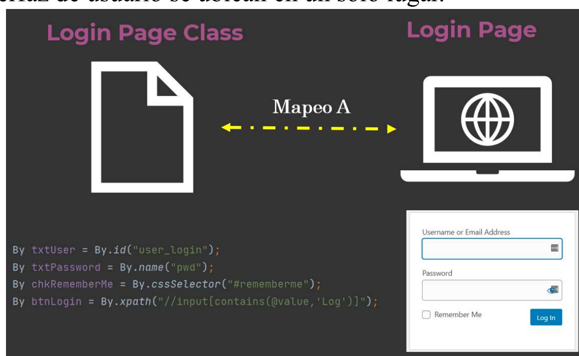

Fig. 3. Page Object Model.

Dentro de las ventajas que se encuentran, existe una separación clara entre el código de prueba y el código específico de la página, como los localizadores (o su uso si usa un mapa de interfaz de usuario) y el diseño.

Existe un repositorio único para los servicios u operaciones que ofrece la página en lugar de tener estos servicios dispersos a lo largo de las pruebas.

En ambos casos, esto permite que cualquier modificación requerida debido a los cambios en la interfaz de usuario se realice en un solo lugar [11].

I. Page Factory

Page Factory es una clase proporcionada por Selenium WebDriver para admitir patrones de diseño de objetos de página, el cual se puede ver su implementación en la Fig. 4. En Page Factory, los ingenieros de prueba utilizan la anotación *@FindBy* [12]. Se utiliza un método llamado *initElements()* para inicializar elementos web, el cual debe ir en el constructor de la clase página.

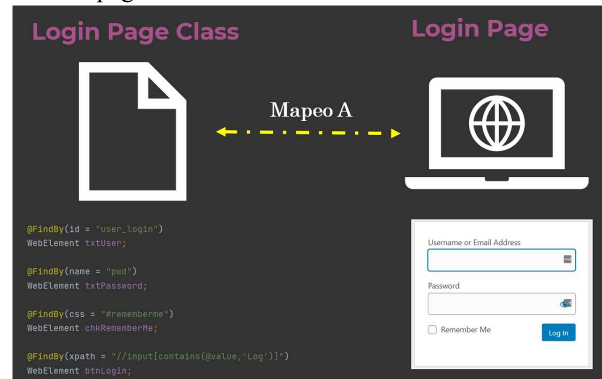

Fig. 4. Page Factory.

III. METODOLOGÍA

Cuando se habla de frameworks (marcos de trabajo por su traducción al español), es muy común tener clases, métodos, patrónes de diseño, etc. que son utilizados para una implementación de las pruebas automatizadas, sin embargo estas implementaciones no siempre otorgan beneficios en cuestión de mantenibilidad y reusabilidad. Para poder identificar en que áreas se puede implementar los principios, se tiene que revisar las clases, métodos, patrones de diseño que pueden ser mejorados, desde la llamada de un navegados hasta le mejor implementación de POM, a continuación se ve en que escenarios se podría detectar el uso de cada principio para posteriormente, refactorizar el código para hacerlo más mantenible, reutilizable, rápido y eficiente.

A. Page Factory o POM

Si se utiliza POM o Page Factory, probablemente se puede encontrar con clases página como la que se muestra en la Fig. 5, las cuales carecen de mantenibilidad y escalabilidad, mientras elementos web existan, dichas clases páginas seguirán creciendo.

Pudiéramos decir que tanto POM como Page Factory, hacen una representación sobre el principio SRP, ya que cada clase página, tiene la única responsabilidad de representar los objetos y las interacciones que se necesitan realizar en cada página. La figura 5 puede no representar muchas líneas de código, sin embargo, cuando una página tiene muchos componentes web como cuadros de texto, links, etiquetas, botones, etc. estas clases se vuelven poco mantenibles ya que se llenan tanto de objetos web como de sus métodos para interactuar con cada uno de ellos. Es por esta razón, que cada clase página, en realidad no cumple con SRP. Para poder cumplir fielmente con SRP se puede realizar una refactorización del código donde se separen los objetos y las interacciones en diferentes clases.

```
Menu Page Class

public class MenuPage extends BasePage {
  @FindBy(xpath = "(//*[contains(text(),'Dashboard')])[4]")
  protected WebElement navDashboard;

  @FindBy(xpath = "(//*[contains(text(),'Teachers')])[2]")
  protected WebElement navTeachers;

  @FindBy(xpath = "(//*[contains(text(),'Students')])[2]")
  protected WebElement navStudents;

  @FindBy(xpath = "(//*[contains(text(),'Parents')])[2]")
  protected WebElement navParents;

  @FindBy(xpath = "(//*[contains(text(),'Classes')])[2]")
  protected WebElement navClasses;

  @FindBy(xpath = "(//*[contains(text(),'Attendance')])[3]")
  protected WebElement navAttendance;

  @FindBy(xpath = "(//*[contains(text(),'Events')])[2]")
  protected WebElement navEvents;

  @FindBy(xpath = "(//*[contains(text(),'Notify')])[2]")
  protected WebElement navNotify;

   @FindBy(xpath = "(//*[contains(text(),'Transport')])[2]")
   protected WebElement navTransport;

   @FindBy(xpath = "(//*[contains(text(),'General Settings')])[2]")
   protected WebElement navGeneralSettings;

   public void goToDashboard(){navDashboard.click();}

   public void goToTeachers(){navTeachers.click();}

   public void goToStudent(){navStudents.click();}

   public void goToParents(){navParents.click();}

   public void goToClasses(){navClasses.click();}

   public void goToAttendance(){navAttendance.click();}

   public void goToEvents(){navEvents.click();}

   public void goToNotify(){navNotify.click();}

   public void goToTransport(){navTransport.click();}

   public void goToGeneralSettings(){navGeneralSettings.click();}
}
```

Fig. 5. Clase para almacenar objetos y acciones de un menu.

B. *Selección de Navegadores*

Cuando realizamos automatización web, una parte clave es poder seleccionar el navegador en el que necesitamos ejecutar nuestra suite de pruebas, esto puede ser resuelto con una declaración interruptor como *switch* muy conocida en los lenguajes de programación, dentro de cada bifurcación, poder indicarle qué navegador queremos instanciar y que se utilice como base para la ejecución completa, tal como se ve en la Fig. 6.

```
Browser Driver Selection

public enum BrowserType {
    CHROME,
    EDGE,
    FIREFOX,
    SAFARI
}

public void setDriver(BrowserType browserType) {
  switch (browserType) {
    case CHROME -> driver.set(new ChromeDriver());
    case EDGE -> driver.set(new EdgeDriver());
    case FIREFOX -> driver.set(new FirefoxDriver());
    case SAFARI-> driver.set(new SafariDriver());
  }
  driver.get().manage().window().maximize();
}
```

Fig. 6. Método y enumerable para seleccionar navegadores.

Al revisar a detalle, el método *setDriver()*, al necesitar de un *switch* para la selección del navegador, da pauta a que no se cumpla con el OCP, ya que si quisiera agregar otro navegador como Brave u Opera o remover alguno de los ya existentes en el enumerable *BrowserType*, se tendría que modificar el método ya mencionado.

C. *Diferentes Roles*

Es muy común que las aplicaciones cuenten con diferenes roles, cada rol tiene acceso a ciertas funcionalidades, en la página de prueba de Titanium Institute® la cual se puede visitar en https://demosite. titaniuminstitute. com.mx/wp- admin/ admin. php?page =sch-dashboard, es una aplicación web para la gestión de una institución educativa que cuenta con los roles de administrador, maestro y estudiante. La Fig. 7 muestra la interfaz gráfica del administrador; la interfaz del maestro la podemos encontrar en la Fig. 8 y la del estudiante en Fig. 9.

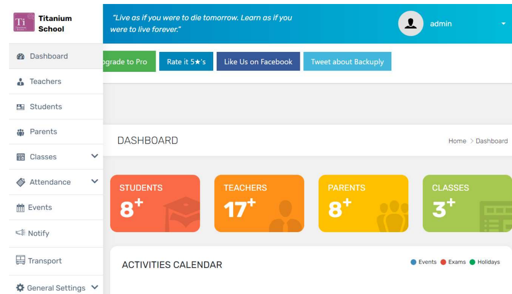

Fig. 7. Interfaz de usuario del administrador.

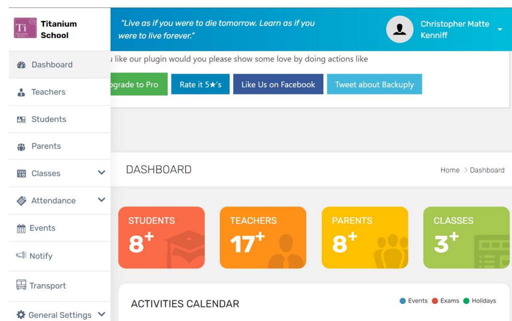

Fig. 8. Interfaz de usuario del maestro.

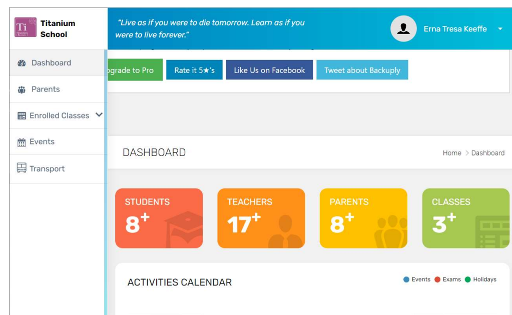

Fig. 9. Interfaz de usuario del estudiante.

Se puede ver que tando administrador como maestro, cuentan con los mismos menús, sin embargo el estudiante solo cuenta con algunos de ellos (Dashboard, Parents, Enrolled Classes, Events y Transport). Se puede llegar a pensar, que crear una clase página que almacene al usuario que tenga todos los componentes del menú es una buena opción tal como se mostró en la Fig. 5, sin embargo, varios de esos métodos no son necesarios para el estudiante, esta situación nos lleva a replantear, la implementación de las páginas para poder cumplir con LSP.

D. *Transformar a Listas*

La lectura de diferentes fuentes de información, es crucial para poder crear casos de prueba que realicen múltiples combinaciones para asi, asegurar mayor cobertura de los escenarios que ya existen. Es muy común generar clases que nos ayuden a leer estas fuentes de información como archivos

de excel, Notación de Objetos JavaScript (*JSON*, por sus siglas en inglés), de queries (consultas por su traducción al español) en el caso de las bases de datos.

Se creó la interfaz de la Fig. 10, con la cual se puede obtener la información y convertirla a LinkedHashMap, la cual contiene métodos, uno para obtener la tabla y otro para convertir una *LinkedHashMap* a un objeto bidimensional.

```
Get Table Array Interface

public interface IGetTableArray {
    Object[][] getTableArray(String file, String iterableItem);
    Object[][] asTwoDimensionalArray(List<LinkedHashMap<Object,Object>> results);
}
```

Fig. 10. Clase página de menú.

Para poder leer archivos de Excel, se creó la clase de la Fig. 11.

```
ExcelArrayData Class

public class ExcelArrayData implements IGetTableArray{
  @Override
  public Object[][] asTwoDimensionalArray(List<LinkedHashMap<Object,Object>> interimResults)
  {
    // lógica para convertir un LinkedHashMap a Object[][]
    return results;
   }

   @Override
   public Object[][] getTableArray(String excelWorkbook, String excelWorkSheet) {
     // lógica para leer un archivo .xlsx y convertirlo en un Object[][]
      return asTwoDimensionalArray(results);
    }
}
```

Fig. 11. Clase para leer archivos .xlsx.

Para leer archivos del tipo .sql, se diseño la clase que se muestra en la Fig. 12.

```
SQLArrayData Class

public class SQLArrayData implements IGetTableArray{
  @Override
  public Object[][] asTwoDimensionalArray(List<LinkedHashMap<Object,Object>> interimResults)
  {
    // lógica para convertir un LinkedHashMap a Object[][]
    return results;
   }

   @Override
   public Object[][] getTableArray(String sqlFile, String dbName) {
     // lógica para leer un archivo .sqly convertirlo en un Object[][]
      return asTwoDimensionalArray(results);
    }
}
```

Fig. 12. Clase para leer archivos .sql.

Por último, es necesario leer archivos .json, y la Fig. 13 muestra el código generado.

```
JSONArrayData Class

public class JSONArrayData implements IGetTableArray{
  @Override
  public Object[][] asTwoDimensionalArray(List<LinkedHashMap<Object,Object>> interimResults)
  {
    throw new UnsupportedOperationException();;
   }

    @Override
    public  Object[][] getTableArray(String filename, String jsonObj){
        // lógica para leer un archivo .json y convertirlo en un Object[][]
        return testData;
    }

}
```

Fig. 13. Clase para leer archivos .json.

Se observa que la clase que se encarga de leer archivos JSON, no necesita de la implementación de *asTwoDimentionaArray()*, dando paso a poder utilizar ISP para mejorar la interfaz *IGetTableArray*.

E. Cambio de Fuente de Datos

En ocasiones, el cliente necesita que se cambie la fuente donde se estás extrayendo los datos. *TestNG* [13] tiene una herramienta muy útil llamada *DataProvider*, la cual nos permite utilizar los datos y sustituirlo en variables para poder recorrer varias combinaciones y generar permutaciones de nuestros casos de prueba. En el punto anterior, creamos las clases para extraer nuestros datos (archivos .xlsx, .json o .sql), los cuales se pueden implementar como se ve en la Fig. 14.

```
JSONArrayData Class

Object[][] tableArray = null;

@DataProvider
public Object[][] getJSONProviderData(){
  tableArray = new JSONArrayData();
  return tableArray.getJSONTableArray("webusers.json", "invalidUsers");
}

@DataProvider
public Object[][] getExcelProviderData(){
  tableArray = new ExcelArrayData();
  return tableArray.getWorkBookTableArray("webusers.xlsx", "invalidUsers");
}

@DataProvider
public Object[][] getSQLProviderData(){
  tableArray = new SQLArrayData();
  return tableArray.getSQLTableArray("webusers.sql", "invalidUsers");
}
```

Fig. 14. DataProviders para extraer datos.

Si se incia la extracción de datos desde un archivo .json (*getJSONProviderData()*), pero tiempo después, el cliente decide cambiar la fuente de datos a una conexión con la base de datos que se haya creado (*getSQLProviderData()*), podemos llegar a la conclusión de que será necesario modificar nuestro *DataProvider* cada vez que la fuente de información se cambie, ya que, si más adelante se desea cambiar a un archivo .xlsx tendríamos que realizar el mismo proceso (*getExcelProviderData()*), esta es una clara violación al DIP.

IV. RESULTADOS

A. Single Responsibility Principle (SRP)

Para poder cumplir con SRP, la Fig. 5 se puede dividir en dos clases, una que solo tenga la responsabilidad de mantener los objetos de las páginas como se ve en la Fig. 15 y otra que almacene las acciones que se pueden realizar en cada página como la Fig. 16 y herede de la clase página creada.

```
POM Class

public class MenuPage extends BasePage {
  @FindBy(xpath = "(//*[contains(text(),'Dashboard')])[4]")
  protected WebElement navDashboard;

  @FindBy(xpath = "(//*[contains(text(),'Teachers')])[2]")
  protected WebElement navTeachers;

  @FindBy(xpath = "(//*[contains(text(),'Students')])[2]")
  protected WebElement navStudents;

  @FindBy(xpath = "(//*[contains(text(),'Parents')])[2]")
  protected WebElement navParents;

  @FindBy(xpath = "(//*[contains(text(),'Classes')])[2]")
  protected WebElement navClasses;

  @FindBy(xpath = "(//*[contains(text(),'Attendance')])[3]")
  protected WebElement navAttendance;

  @FindBy(xpath = "(//*[contains(text(),'Events')])[2]")
  protected WebElement navEvents;

  @FindBy(xpath = "(//*[contains(text(),'Notify')])[2]")
  protected WebElement navNotify;

   @FindBy(xpath = "(//*[contains(text(),'Transport')])[2]")
   protected WebElement navTransport;

   @FindBy(xpath = "(//*[contains(text(),'General Settings')])[2]")
   protected WebElement navGeneralSettings;
 }
```

Fig. 15. Page Factory con solo objetos web.

```
Task Class

public class MenuTask extends MenuPage {
  public void goToDashboard(){navDashboard.click();}

   public void goToTeachers(){navTeachers.click();}

   public void goToStudent(){navStudents.click();}

   public void goToParents(){navParents.click();}

   public void goToClasses(){navClasses.click();}

   public void goToAttendance(){navAttendance.click();}

   public void goToEvents(){navEvents.click();}

   public void goToNotify(){navNotify.click();}

   public void goToTransport(){navTransport.click();}

   public void goToGeneralSettings(){navGeneralSettings.click();}
 }
```

Fig. 16. Clase donde se almacenan las acciones de cada página.

B. Open-Closed Principle (OCP)

La Fig. 6 representa la forma de llamar al navegador que se necesite elegir, para cumplir el OCP, se hizo uso de la dependencia *Reflections*, la cual se puede encontrar en https://mvnrepository.com/artifact/org.reflections/reflections. Se creó una interfaz para poder crear la instancia del navegador, el cual se puede ver en la Fig. 17.

```
IDriver Interface

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

Fig. 17. Interfaz para crear el driver de cada navegador.

Después se crearon clases de la Fig. 18, *ChromeDriverManager y EdgeDriverManager*, que hacen referencia a los navegadores Chrome y Edge, dichas clases implementan la interfas *IDriver*.

```
Driver Manager classes

public class ChromeDriverManager implements IDriver{

    public ChromeDriverManager(){}
    @Override
    public WebDriver createDriver() { return new ChromeDriver();}
}

public class EdgeDriverManager implements IDriver{

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

Fig. 18. Clases para abrir los navegadores Chrome y Edge.

Por último, se modificó el método *setDriver()* como muestra la Fig. 19, para que, dependiendo del navegador que se elija, se llame la clase respectiva a la ejecución de dicho navegador. Además, el uso del enumerable *BrowserType* se puede remover, ya que el parámetro de entrada necesita de que se indique el nombre del navegador e igualarlo al nombre que se tiene en el constructor. De esta forma, si se quiere agregar otro navegador, se crearía otra clase como la que se muestra en la Fig. 18, implementando la interfaz *IDriver*, dando como paso a que se sigue abierto a la extension de dicha clases, pero al no necesitar cambiar algo dentro del método *setDriver()*, se indica que se está cerrado a la modificación.

```
Set Driver Method

public void setDriver(String browser) {
        Set<Class<? extends IDriver>> driverInterfaces =
        new Reflections(IDriver.class).getSubTypesOf(IDriver.class);

        for (var driverManager:driverInterfaces){
            if (driverManager.getName().contains(browser)){
                try {
                    driver.set((WebDriver) driverManager
                    .getMethod("createDriver")
                    .invoke(Class.forName(driverManager.getName())
                    .getConstructor()
                    .newInstance()));
                    } catch (Exception e) {
                     new FrameworkException(e.getMessage());
                }
            }
        }
        driver.get().manage().window().maximize();
    }
```

Fig. 19. Método para llamar las clases que inicializan los navegadores.

C. Liskov Substitution Principle (LSP)

Teniendo en cuenta las Fig. 15 y Fig. 16, se puede modificar esta última para poder cumplir con LSP. Primero se generó una clase abstracta llamada *MenuAbstractClass* (Fig. 20), la cual servirá para mantener los menús que existen para los tres roles (administrador, maestro, estudiante).

```
Menu Abstract Class

public abstract class MenuAbstractClass extends MenuPage {

    protected abstract MenuAbstractClass goToDashboard();
    protected abstract MenuAbstractClass goToParent();
    protected abstract MenuAbstractClass goToEvents();
    protected abstract MenuAbstractClass goToTransport();

}
```

Fig. 20. Clase abstracta que almacena los menús comúnes.

Se desarrollaron las clases de *AbstractMenuClass* que se observa en la Fig. 21 y la de *StudentAbstractClass* de la Fig. 22, las cuales tienen como super clase a *MenuAbstractClass*. Dando la opción de que solo los estudiantes, puedan acceder al menú de *Enrolled Classes* y tanto el administrador como el maestro, puedan acceder a los demás menús.

```
Menu Abstract Class

public abstract class AdminAbstractClass extends MenuAbstractClass{
    protected abstract MenuAbstractClass goToTeachers();
    protected abstract AddStudent goToStudents();
    protected abstract MenuAbstractClass goToClasses();
    protected abstract MenuAbstractClass goToAttendance();
    protected abstract MenuAbstractClass goToNotify();
    protected abstract MenuAbstractClass goToGeneralSettings();
}
```

Fig. 21. Clase abstracta para los menús de administrador y maestro.

```
Menu Abstract Class

public abstract class StudentAbstractClass extends MenuAbstractClass{
    protected abstract MenuAbstractClass goToEnrolledClasses();

}
```

Fig. 22. Clase abstracta para el menú de estudiante.

Ahora, se pueden crear las clases de acción que encontramos en la Fig. 23 y Fig. 24, que implementarán las clases abstractas previamente creadas, en la cual se puede agregar la lógica necesaria para poder acceder a los menús que le corresponden a cada rol.

```
Admin Menu Task Class

public class AdminMenuTask extends AdminAbstractClass {

    @Override
    public AdminMenuTask goToTeachers() {
        navTeachers.click();
        return (AdminMenuTask) (actualPage = getInstance(AdminMenuTask.class));
    }

    @Override
    public AddStudent goToStudents() {
        navStudents.click();
        return (AddStudent) (actualPage = getInstance(AddStudent.class));
    }

    @Override
    public AdminMenuTask goToClasses() {
        navClasses.click();
        return (AdminMenuTask) (actualPage = getInstance(AdminMenuTask.class));
    }

    // ... se implementan los métodos restantes
}
```

Fig. 23. Clase abstracta para el menú del administrador y maestro.

```
Student Menu Task Class

public class studentMenuTask extends studentAbstractClass {

    @Override
    public StudentMenuTask goToEnrolledClasses() {
        navEnrolledClasses.click();
        return (StudentMenuTask) (actualPage = getInstance(StudentMenuTask.class));
    }

    // ... se implementan los métodos restantes de MenuAbstractClass
}
```

Fig. 24. Clase abstracta para el menú de estudiante.

Finalmente, se pueden implementar las clases de acciones como se muestra en la Fig. 25, sin tener dependencia de métodos que no existen o no tienen una función que en verdad ayude a la creación y ejecución de las mismas.

```
Test MEthods

@Test
void teacherLoginWithRightCredentials() {
  // pasos para ingresar a la aplicación

   actualPage = getInstance(AdminMenuTask.class);
   actualPage.as(AdminMenuTask.class).goToTeachers();

   actualPage = getInstance(AdminMenuTask.class);
    ctualPage.as(AdminMenuTask.class).goToParent();
}

@Test
void studentLoginWithRightCredentials() {
  // pasos para ingresar a la aplicación

   actualPage = getInstance(StudentMenuTask.class);
   actualPage.as(StudentMenuTask.class).goToEnrolledClasses();
 }
```

Fig. 25. Métodos de prueba donde se utiliza LSP.

D. *Interface Segregation Principle (ISP)*

El código de la Fig. 10 se refactorizó, dando paso a la creación de la interfaz mostrada en la Fig. 26 y la interfaz de la Fig. 27.

```
IGetTableArray Interface

public interface IGetTableArray {
    Object[][] getTableArray(String file, String iterableItem);
}
```

Fig. 26. Interfaz refactorizada para cumplir con ISP.

```
IGetTwoDimensionalArray Interface

public interface IGetTwoDimensionalArray {
    Object[][] asTwoDimensionalArray(List<LinkedHashMap<Object,Object>> results);
}
```

Fig. 27. Nueva interfaz creada después de la refactorización.

Esta refactorización ayuda a que las clases de lecturas de datos, no tuvieran implementaciones que no necesitan, dando como resultado el código de la Fig. 28.

```
Read Data Classes

public class ExcelArrayData implements IGetTableArray,IGetTwoDimensionalArray{
  @Override
  public Object[][] asTwoDimensionalArray(List<LinkedHashMap<Object,Object>> interimResults)
  {
    // lógica para convertir un LinkedHashMap a Object[][]
    return results;
   }

   @Override
   public Object[][] getTableArray(String excelWorkbook, String excelWorkSheet) {
     // lógica para leer un archivo .xlsx y convertirlo en un Object[][]
      return asTwoDimensionalArray(results);
    }
}

public class SQLArrayData implements IGetTableArray,IGetTwoDimensionalArray{
  @Override
  public Object[][] asTwoDimensionalArray(List<LinkedHashMap<Object,Object>> interimResults)
  {
    // lógica para convertir un LinkedHashMap a Object[][]
    return results;
   }

   @Override
   public Object[][] getTableArray(String sqlFile, String dbName) {
     // lógica para leer un archivo .sql y convertirlo en un Object[][]
      return asTwoDimensionalArray(results);
    }
}

public class jsonArrayData implements IGetTableArray{
   @Override
   public Object[][] getTableArray(String filename, String jsonOb) {
     // lógica para leer un archivo .json y convertirlo en un Object[][]
      return asTwoDimensionalArray(results);
    }
}
```

Fig. 28. Clases refactorizadas usando ISP.

E. *Dependency Inversion Principle (DIP)*

En la Fig. 14, se puede ver los *DataProviders* que servirán para extraer datos, dependiendo del requerimiento del client. Para implementar de manera correcta el DIP, es necesario utilizar la interfaz que se muestra en la Fig. 26, la cual utiliza el *DataProvider* de la Fig. 29, para extraer los datos dependiendo de si se quieren archivos .json, .xlsx o .sql.

```
Read JSON Data Class

@DataProvider
public Object[][] getProviderData(){
  IGetTableArray iGetTableArray = new JSONArrayData();
  return iGetTableArray.getTableArray("webusers.json", "invalidUsers");
}
```

Fig. 29. Implementación de DIP.

En el caso de que el cliente necesite de cambiar la fuente de donde se extraerán los datos, simplemente se cambiaría la implementación de *iGetTableArray* a cualquiera de las ya previamente creadas, sin depender de todas al mismo tiempo.

V. CONCLUSIÓN

La aplicación de los principios SOLID en la automatización de pruebas utilizando Selenium WebDriver y Java forma un pilar robusto en el área de ingeniería de software, proporcionando un esquema estructurado que permite a los ingenieros de pruebas desarrollar un código más limpio, escalable y mantenible. Al adherirse a estos principios, se puede desarrollar un framework de pruebas que no solo es resistente ante los cambios en las aplicaciones bajo prueba, sino que también facilita una adaptabilidad ágil a los nuevos requerimientos y funcionalidades.

Implementar los principios SOLID en la creación de marcos de pruebas con Selenium WebDriver y Java brinda un valor tangible al desarrollo de software mediante la optimización del proceso de pruebas automatizadas. El resultado es un código estructurado de manera lógica, con mayor reutilización y menos errores, que sustenta la calidad y la entrega continua del producto software, haciendo que las pruebas propicien su realización, en lugar de obstaculizar el ciclo de vida del desarrollo de software. Esto no solo mejora la efectividad de las pruebas, sino que también reduce los costos a largo plazo asociados con el mantenimiento del código de prueba y el manejo de la deuda técnica.

El código fuente se puede revisar en: https://dev.azure.com/gilsancheztsO649/SDET%20Bootcamp/_git/SDETJavaFramework.git

REFERENCIAS

[1] NextGenerationAutomation. “S.O.L.I.D Design Principles for Automation Developers”. NGAutomation. Accedido el 2 de octubre de 2023. [En línea]. Disponible: https://www.nextgenerationautomation.com/post/s-o-l-i-d-principles

[2] J. Rubira, (2019). ”SOLID, cinco principios básicos de diseño de clases”, Genbeta. [En Línea]. Disponible: https://www.genbeta.com/desarrollo/solid-cinco-principios-basicos-de-diseno-de-clases.

[3] R. C. Martin,*”Clean architecture: a craftsman's guide to software structure and design”*, Boston, Prentice Hall, 2018.

[4] Universidad de los Andes, ”Barbara Liskov, pionera en la computación moderna”, *Revista Foro ISIS.* No. 02, pp. 62-63, Sep. 2013.

[5] Codeonedigest cod. “Liskov Substitution Principle Tutorial with Java Coding Example for Beginners”. LinkedIn. Accedido el 5 de octubre de 2023. [En línea]. Disponible: https://www.linkedin.com/pulse/liskov-substitution-principle-tutorial-java-coding-example-digest/.

[6] D. Bhattacharjee. “Interface Segregation Principle in Java”. Baeldung. Accedido el 3 de octubre de 2023. [En línea]. Disponible: https://www.baeldung.com/java-interface-segregation.

[7] CodeCraft. “SOLID Design Principles-5: Dependency Inversion Principle”. Medium. Accedido el 3 de octubre de 2023. [En línea]. Disponible: https://codecraft.medium.com/depend-on-abstractions-and-not-on-modules-with-the-dependency-inversion-principle-f31b9d10789b.

[8] J. Thompson. “Dependency Inversion Principle - Spring Framework Guru”. Spring Framework Guru. Accedido el 4 de octubre de 2023. [En línea]. Disponible: https://springframework.guru/principles-of-object-oriented-design/dependency-inversion-principle.

[9] Selenium. “WebDriver”. Selenium. Accedido el 8 de agosto de 2023. [En línea]. Disponible: https://www.selenium.dev/documentation/webdriver/

[10] Oracle. “What is Java technology and why do I need it?” Oracle. Accedido el 5 de octubre de 2023. [En línea]. Disponible: https://www.java.com/en/download/help/whatis_java.html

[11] Selenium. “Page object models”. Selenium. Accedido el 8 de agosto de 2023. [En línea]. Disponible: https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/

[12] E. Díaz Asencio, “Automatización de pruebas de regresión”, trabajo de grado, Univ. Sevilla, Sevilla, 2021. Accedido el 4 de octubre de 2023. [En línea]. Disponible: https://idus.us.es/bitstream/handle/11441/125114/TFG-3425-DIAZ%20ASENCIO.pdf?sequence=1&isAllowed=y

[13] testng org. “TestNG - Welcome”. TestNG. Accedido el 5 de octubre de 2023. [En línea]. Disponible: https://testng.org/doc/