**Introducción**

El concepto básico en el que se basa este patrón es el de representar cada una de las pantallas que componen el sitio web o la aplicación que nos interesa probar, como una serie de objetos que encapsulan las características y funcionalidades representadas en la página. De esta manera, nos permite consolidar el código para interactuar con los elementos de una página en cada uno de los **PageObjects**.

Al crear un **PageObject**, lo que estamos consiguiendo es crear una capa de abstracción entre el «¿Qué podemos hacer/ver en la página? », y el « Cómo » se realiza esta acción, simplificando enormemente la creación de nuestras pruebas y reutilizando el código con el que interactuamos con la página en concreto. Y a la vez, cualquier cambio que se produzca en la UI únicamente afectará al **PageObject** en cuestión, no a los test ya implementados.

Esto se debe a que un test nunca debe manipular directamente elementos de la página (UI), si no que este manejo debe realizarse a través del **PageObject** correspondiente que representa la página. Para entendernos, el **PageObject**se convierte en una API con la que fácilmente podemos encontrar y manipular los datos de la página.

[](https://qanewsblog.files.wordpress.com/2014/07/solde1.jpg)

Podemos encontrarnos con que una página corresponda a más de un **PageObject**, si se da el caso de que algunas áreas de la página son lo suficientemente significativas. Por ejemplo, en una página web, podemos tener un **PageObject**para el header y otro para el body.

**Detalles de la implementación**

Una de las opciones de las que disponemos es crear, en primer lugar, una clase “básica”, que posteriormente será la que extenderán cada uno de los distintos **PageObjects** que vayamos implementando. Esta **PageBase** nos aporta la estructura básica y las propiedades generales que utilizaremos:

[](https://qanewsblog.files.wordpress.com/2014/07/solde2.jpg)

Y una vez creada esta página, crear los **PageObjects** necesarios. Un ejemplo de una página de “Login” podría ser:

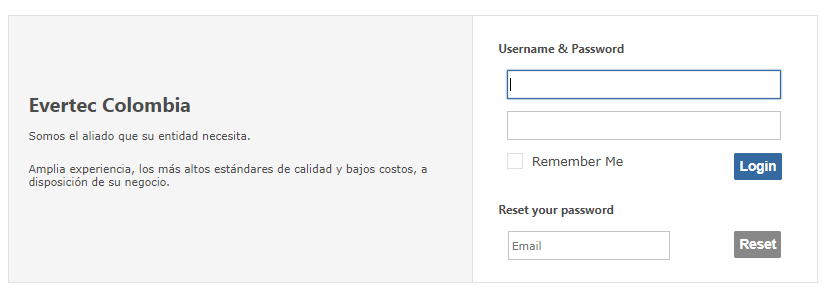
[](https://qanewsblog.files.wordpress.com/2014/07/solde3.jpg)

Lógicamente, si lo creemos conveniente nos podemos saltar la creación de la PageBase, y crear directamente los distintos**PageObjects**con todas las propiedades y métodos necesarios.

Como podemos ver en este simple ejemplo, los métodos que interactúan con la página en si, normalmente o bien no nos devolverán ningún parámetro, o bien serán de tipo sencillo, como boolean, string, etc (ej. returnErrorMessage() nos devuelve un String con el mensaje de error de la página de login). Pero cuando se trata de funciones de **navegación** entre páginas, estas nos devolverán un nuevo objeto **PageObject** del tipo correspondiente a la página donde navegamos. En el ejemplo superior, si desde la página LoginPage ejecutamos el método loginAs, se realizará el login y consecuentemente se nos devolverá un objeto de HomePage, que corresponde al **PageObject**de la página principal.

Como consecuencia de este enfoque, será necesario crear métodos en los que la acción a realizar acabe con éxito (ej: loginAs, que nos devuelve la HomePage), y otros en los que la acción no sea correcta (ej: loginWithError, que se mantiene en la LoginPage).

**Un ejemplo práctico:**



Vamos a plantear un ejemplo real: la página de Login de la web de GEMINI. El ejemplo a continuación se describe como un ejemplo de Selenium, en Java.

**PageObject:**

El login, el PageObject que modela la página podría implementarse de la siguiente forma:

package PageObject.Pages;  
  
import PageObject.Actions.actions;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
  
public class login extends actions {  
  
 public login(WebDriver driver) {  
 super(driver);  
 }  
  
 By user = By.*xpath*("//input[@id='Username']");  
 By pass = By.*xpath*("//input[@id='regular-password']");  
 By login = By.*xpath*("//input[@id='regular-signin']");  
 By dashBoard = By.*xpath*("//td[contains(.,'Jason Andres Fierro Hernandez')]");  
  
 public void login() throws InterruptedException {  
 waitImplicit(user);  
 input("User",user);  
 waitImplicit(pass);  
 input("Password",pass);  
 waitImplicit(login);  
 click(login);  
 waitImplicit(dashBoard);  
 Thread.*sleep*(2000);  
 getText(dashBoard);  
 Thread.*sleep*(2000);  
 textEquals("Jason Andres Fierro Hernandez", dashBoard);  
 }  
}

Como podemos ver, en este código estamos implementando algunos de los métodos para interactuar con la página (introducir usuario, contraseña, click en login, etc), en los que enlazamos los elementos de la página de GEMINI en si misma.

**Tests:**

Una prueba sencilla que simplemente ejecute un login y verifique que se accede al “Inbox” correctamente podría implementarse de la siguiente forma:

package Tests;  
  
  
import org.junit.Ignore;  
import org.junit.Test;  
  
public class MejoraInterna extends TestSuite{  
  
 @Test  
 public void loginGemini() throws InterruptedException {  
 login.login();  
 }  
  
 @Ignore  
 public void dashBoard() throws InterruptedException {  
 pageDashBoard.dashBoard();  
 }  
}

En esta prueba básico, utilizamos la función de loginGemini que nos ofrece el **PageObject** de la página de Login de GEMINI.

Tal y como podemos ver, al utilizar el patrón de**PageObjects**, las pruebas que debemos implementar son sencillos y a la vez intuitivos y entendibles, y una vez realizado el esfuerzo inicial de desarrollar los distintos métodos para interactuar con el objeto, la incorporación de nuevas pruebas se puede realizar en un tiempo razonable.

Como vemos en el ejemplo anterior se están utilizando unas anotaciones como @Test y @Ignore, en este caso solo serán ejecutados los test que tengan la anotación @Test dependiendo de la librería que se esta utilizando para poder ejecutar estos casos automatizados.

**Acciones**

Se pueden crear acciones para que el código sea más limpio al momento de realizar cada prueba automatizada, y este puede ser utilizada en el *PAGEOBJECT* al momento de enviar las acciones extendiendo la clase JAVA y en la página se debe realizar el llamado de la clase acciones como se verá a continuación:

**Clase actions**

public class actions{  
  
 protected WebDriver driver;  
  
 public actions(WebDriver driver) {  
 this.driver = driver;  
 }  
  
 protected WebElement findElement(By locator) {  
 return driver.findElement(locator);  
 }  
  
 protected List<WebElement> findElements(By locator) {  
 return driver.findElements(locator);  
 }  
   
  
 protected String getText(By locator) {  
 String getText = findElement(locator).getText();  
 System.*out*.println("El texto guardado es: " + getText);  
 return null;  
 }  
  
 protected void input(String inputText, By locator) {  
 driver.findElement(locator).sendKeys(inputText);  
 }  
  
 protected void getAttribute(String inputText, By locator) {  
 String getAttribute = driver.findElement(locator).getAttribute(inputText);  
  
 System.*out*.println("El texto guardado es: " + getAttribute);  
 }  
  
 protected String textEquals(String textExpected, By locator){  
 String getText = findElement(locator).getText();  
 Assert.*assertEquals*(textExpected,getText);  
 System.*out*.println("El texto esperado es: " + textExpected);  
 System.*out*.println("El texto actual es: " + getText);  
 return null;  
 }  
  
 protected void typeLeft(By locator) {  
 driver.findElement(locator).sendKeys(Keys.*ARROW\_LEFT*);  
 }  
  
 protected void clear(By locator) {  
 driver.findElement(locator).clear();  
 }  
  
 protected void click(By locator) {  
 driver.findElement(locator).click();  
 }  
  
 protected Boolean isDisplayed(By locator) {  
 try {  
 return driver.findElement(locator).isDisplayed();  
 } catch (org.openqa.selenium.NoSuchElementException e) {  
 return false;  
 }  
 }  
  
 protected void scrollDown(By locator) {  
 JavascriptExecutor executor = (JavascriptExecutor) driver;  
 executor.executeScript("window.scrollBy(0,1000)");  
 }  
  
 protected void scrollUp(By locator) {  
 JavascriptExecutor executor = (JavascriptExecutor) driver;  
 executor.executeScript("window.scrollBy(0,-1000)");  
 }  
  
 protected void visit(String url) {  
 driver.get(url);  
 }  
  
 protected void waitImplicit(By locator) {  
 try {  
 WebDriverWait myWaitVar = new WebDriverWait(driver, 10);  
 myWaitVar.until(ExpectedConditions.*visibilityOfElementLocated*(locator));  
 } catch (Exception e) {  
 System.*out*.print("No se encontro el localizador: " + locator);  
 driver.quit();  
 }  
 }  
  
 protected void verifyLogin(String texto,By locator) {  
 try{  
 driver.findElement(locator).equals(texto);  
 System.*out*.println("Error al iniciar sesión, usuario o contraseña incorrecto");  
 driver.close();  
 } catch (Exception e){  
 System.*out*.println("Ingreso correctamente al login " + locator);  
 }  
 }  
  
 protected void selectElement(String texto,By locator){  
 Select select = new Select(driver.findElement(locator));  
 select.selectByVisibleText(texto);  
 }  
  
 protected void takeScreenShot(String nombre) throws Exception {  
 File scrFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.*FILE*);  
 String imageFileDir = System.*getProperty*("user.dir") + "\\imagenes\\test";  
 if (imageFileDir == null)  
 imageFileDir = System.*getProperty*("user.dir") + "\\imagenes\\test";  
 FileUtils.*copyFile*(scrFile, new File(imageFileDir, nombre));  
 }  
  
}

**Clase login**, en la clase debe indicar extends actions para que tome las acciones indicadas y poder utilizarlas en el método **public void login().**

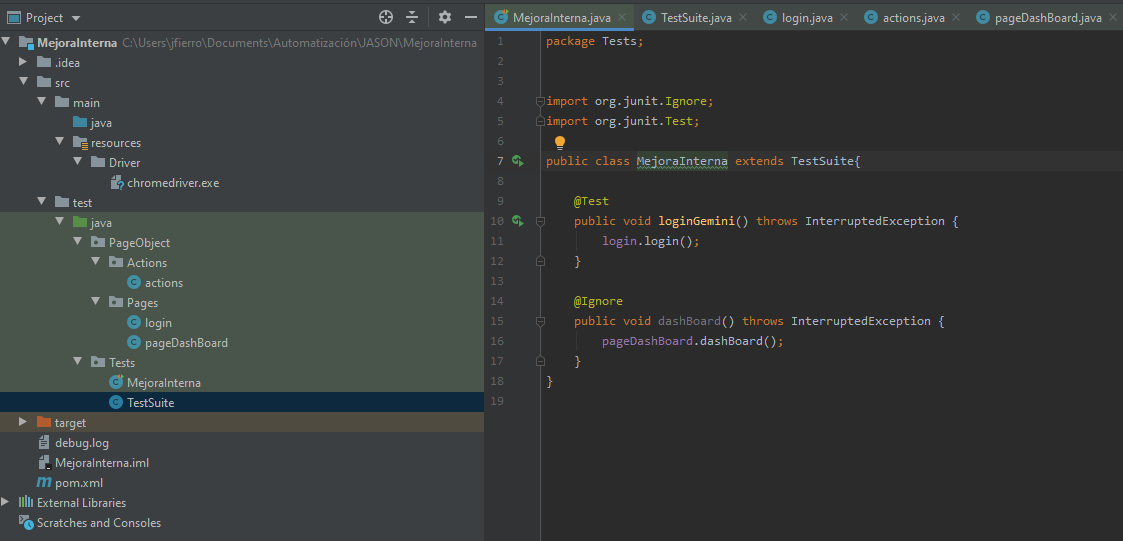
package PageObject.Pages;  
  
import PageObject.Actions.actions;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
  
public class login extends actions {  
  
 public login(WebDriver driver) {  
 super(driver);  
 }  
  
 By user = By.*xpath*("//input[@id='Username']");  
 By pass = By.*xpath*("//input[@id='regular-password']");  
 By login = By.*xpath*("//input[@id='regular-signin']");  
 By dashBoard = By.*xpath*("//td[contains(.,'Jason Andres Fierro Hernandez')]");  
  
 public void login() throws InterruptedException {  
 waitImplicit(user);  
 input("User",user);  
 waitImplicit(pass);  
 input("Password",pass);  
 waitImplicit(login);  
 click(login);  
 waitImplicit(dashBoard);  
 Thread.*sleep*(2000);  
 getText(dashBoard);  
 Thread.*sleep*(2000);  
 textEquals("Jason Andres Fierro Hernandez", dashBoard);  
 }  
}

**Iniciar Proceso de automatización**

Para que este Desarrollo funcione correctamente, debe tener ciertos parámetros como la apertura del navegador que va realizar las pruebas automatizadas, en este caso se esta utilizando Google Chrome, para esto se debe tener en cuenta una clase JAVA con el fin de poder llamar todas las funciones solicitadas al abrir la página, como por ejemplo donde está ubicado el driver para abrir el navegador, el antes y después de abrir el navegador etc, a continuación un ejemplo para abrir la pagina GEMINI que debe hacer antes y después de ejecutar el @Test:

public class TestSuite {  
  
 String chromePath = System.*getProperty*("user.dir") + "\\src\\main\\resources\\Driver\\chromedriver.exe"; //Indicar la ruta del archivo ChromeDriver.exe en la propierdad webdriver.chromer.exe  
 protected WebDriver driver;  
 static String *baseURL* = "http://processa6.ongemini.com/account/login";  
 login login;  
 pageDashBoard pageDashBoard;  
  
  
 private void inicializarPaginas(){  
 login = new login(driver);  
 pageDashBoard = new pageDashBoard(driver);  
 }  
  
 @Before  
 public void chromeDriverConnection() throws InterruptedException {  
 System.*setProperty*("webdriver.chrome.driver", chromePath);//Indicar la ruta del archivo ChromeDriver.exe  
 driver = new ChromeDriver();  
 driver.manage().timeouts().implicitlyWait(10, TimeUnit.*SECONDS*);  
 driver.manage().window().maximize();  
 driver.get(*baseURL*);  
 inicializarPaginas();  
 }  
  
 @After  
 public void cerrarChrome(){  
 driver.quit();  
 }  
}

**Estructura Proyecto**



**Conclusiones:**

Sin duda alguna, los patrones de diseño en desarrollo, nos ofrecen una solución previamente utilizada y de efectividad probada, así como la correspondiente documentación, con  la que enfrentarse a una serie de problemas comunes dentro del mundo del desarrollo y la automatización de pruebas. Y en concreto, el **Page Object**nos ofrece las ventajas de:

* Reducir la duplicidad de código.
* Conseguir pruebas más robustas y fácilmente entendibles
* Mejorar la mantenibilidad de nuestras **pruebas funcionales automatizadas**, punto que se pone de relevancia cuando tenemos una web o aplicación que evoluciona rápidamente.