Design Patterns - Page Object Model

Introduction

Creating tests for UI based applications using Selenium/Appium libraries have two parts,

- Locating the UI elements.
- Perform Operations on these elements and additionally assert some specific conditions.

But as the UI Screens become complex with many pages and elements, corresponding automated tests also become more complex and un-maintainable.

Adding to this misery is when more automation engineers continue to add and extend tests, a lot of code gets duplicated and any change in UI functionality can lead to substantial changes in tests.

To address these issues, developers have taken the approach of formalizing a Design Pattern called Page Object Model (POM)

The POM design pattern dictates that there should be a separation between test classes, and pages (business objects), this methodology allows test projects to decouple responsibilities (tests VS. page logic) and expand their projects rapidly.

Quick and Easy - Clone our Page Object Model Repository

Now that you are all set, with basics of Page Object Model. Let's try out a sample to demonstrate POM in action.

1. Find the sample test in our git repository.
2. Fork the repository.
3. Edit the test and hit Run

Before you proceed with fetching the sample tests, you will have to do two more things, very simple but important ones:

1. If you plan to test a native or hybrid app - Upload your app to your project
2. Fetch your access key

Example (Legacy Style Tests)

The code snippet below shows Java-based test classes implemented using TestNG framework.

```java
public class TestClass1 {
    @BeforeClass
    public void init() {
        // Initialize driver
    }

    @Test
    public void testLogin() {
        // LOGIN
        driver.findElement(By.xpath("//*[@id='username']")).sendKeys();
        driver.findElement(By.xpath("//*[@id='password']")).sendKeys();
        driver.findElement(By.xpath("//*[@id='loginButton']")).click();
    }
}
```
As you can see, login operation is duplicated in both test classes i.e. in TestClass1 and TestClass2. Over a period of time, such duplicate code will increase leading affecting maintainability of test software.

Imagine if there is a slight change in the UI screen, all the affected tests have to be modified and reviewed for validity. One more drawback in the test is that the UI functionality and tests are clustered together at one place which makes code unorganized. It is easy to kick-start creating tests by using an approach discussed above, but in the long term, this is not feasible and is not a great way to develop tests.

**Page Object Model (POM)**

**Page Object Model** is a design pattern which separates the UI elements and tests or operations performed on those elements. Typically the UI elements and all its associated logic is implemented as Page Objects. Tests are completely independent and perform operations on these Page Objects.

The diagram below depicts the interaction between Test and a PageObject.
Implementation involves following steps,

1. Review the overall flow of UI Screens.
2. Create a Page class for every UI screen.
3. A Page Class should return another Page class (via an operation) which represents the next screen in a flow.
4. Create Test classes and test methods which perform operations on Page Objects.

Here is an example using the TestNG framework to develop a simple Test using POM.

1. Create a Page Object Base Class

```java
package pageobjects;

public abstract class PageBase {
    protected <AndroidElement> driver = null;

    public PageBase(AndroidDriver androidDriver) {
        driver = androidDriver;
    }
}
```

2. Create a concrete Object Page Class
public class LoginPage extends PageBase {
    private AndroidElement usernameElement;
    private AndroidElement passwordElement;
    private AndroidElement loginButtonElement;

    /**
     * Constructor
     */
    public LoginPage(AndroidDriver androidDriver) {
        super(androidDriver);
        usernameElement = driver.findElement(By.xpath("//*[@id='usernameTextField']"));
        passwordElement = driver.findElement(By.xpath("//*[@id='passwordTextField']"));
        loginButtonElement = driver.findElement(By.xpath("//*[@id='loginButton']"));
    }

    /**
     * Login
     * @param userName
     * @param password
     */
    public Page login(String userName, String password) {
        usernameElement.sendKeys(userName);
        passwordElement.sendKeys(password);
        loginButtonElement.click();
        return HomePage(driver);
    }
}

3. Create Test Class
package tests;
/**
 * Tests for Example Application.
 */
public class ExampleAppTests {
    private <AndroidElement> driver = null;

    @Before
    public void setUp() {
        // Init Driver
        driver = ....;
    }

    @Test
    public void loginTest() {
        try {
            LoginPage loginPage = new LoginPage(driver);
            loginPage.initialize(driver);
            loginPage.login("company1121", "company1121");
        } catch(Exception e) {
            Assert.fail("Should not have thrown any exception" + e.getStackTrace());
        }
    }

    @Test
    public void loginTest() {
        try {
            LoginPage loginPage = new LoginPage(driver);
            HomePage homePage = loginPage.login("company1121", "company1121");
            homePage.buyTickets(....);
        } catch(Exception e) {
            Assert.fail("Should not have thrown any exception" + e.getStackTrace());
        }
    }
}

From the example above, Tests (ExampleAppTests) are separated from the Page Objects and their associated Elements (LoginPage).

**Structure for a POM based project**

**Summary**
Page Object Model is a great way to develop tests for UI applications.

- Make Test Suites Structured, Maintainable and Reusable.
- The core strategy is to separate UI Locator logic to Page Objects and Test logic to Test Objects