Journal of Technological Innovations

Est. 2020

Designing a Robust Page Object Model (POM) for Cross-Browser and Cross-Platform Testing

Asha Rani Rajendran Nair Chandrika
Email: ashaadarsh2010@gmail.com

Abstract

In the dynamic landscape of web and mobile application testing, ensuring that test automation frameworks are scalable, maintainable, and capable of supporting diverse environments is a critical challenge. The Page Object Model (POM) has emerged as a highly effective design pattern for addressing these concerns. This article explores advanced strategies and best practices for designing a POM that not only adheres to the principles of maintainability and scalability but also supports **cross-browser** and **cross-platform** testing. The strategies outlined here aim to ensure efficient, reliable, and streamlined test execution across varying environments. This approach involves leveraging a variety of techniques such as modularization, abstraction, parallel execution, and cloud-based tools. By following these methodologies, testing teams can maximize test coverage and performance, ultimately delivering high-quality software with confidence.

Keywords: Page Object Model (POM), Test Automation, Cross-Browser Testing, TestNG, Selenium WebDriver, Factory Design Pattern, Configuration Files, Dynamic Test Environments, Locator Strategies, Parallel Test Execution, Cross-Platform Testing, BrowserStack, Sauce Labs, Mobile Testing, Test Framework Scalability, Test Maintenance, Automation Best Practices, Web Automation Framework.

Introduction

In modern software development, test automation plays a pivotal role in ensuring high-quality releases. However, with the growing need to test applications across various browsers and platforms, it is vital that test automation frameworks are designed to be adaptable and resilient. The Page Object Model (POM) design pattern has long been a popular choice for structuring test automation code. This pattern focuses on separating the test scripts from the user interface (UI) elements and behaviors, creating an abstraction layer that simplifies both writing and maintaining tests [1].

While POM provides significant advantages in organizing code, it also needs to evolve to handle the increasing complexity of testing across multiple browsers, devices, and platforms. Traditional POM may not suffice when facing these challenges. As such, a more sophisticated approach to the POM is required—one that incorporates principles of scalability, flexibility, and reusability across different environments.

This article delves into how to design a robust and scalable Page Object Model (POM) for cross-browser and cross-platform testing. By using a series of best practices and advanced strategies, it is possible to build a framework that adapts to various devices, browsers, and operating systems while keeping the automation code clean and maintainable.

Base Page Class for Shared Functionality

A well-structured Page Object Model (POM) relies heavily on the concept of a **Base Page Class**. This class acts as a central hub, consolidating shared methods, properties, and utilities that are commonly used across various page objects in the framework. By placing these functionalities in a single class, you eliminate redundant code and promote consistency in how page objects interact with the browser and web elements. This centralized approach not only simplifies the maintenance of the test automation code but also enhances scalability and flexibility when expanding the framework [2].

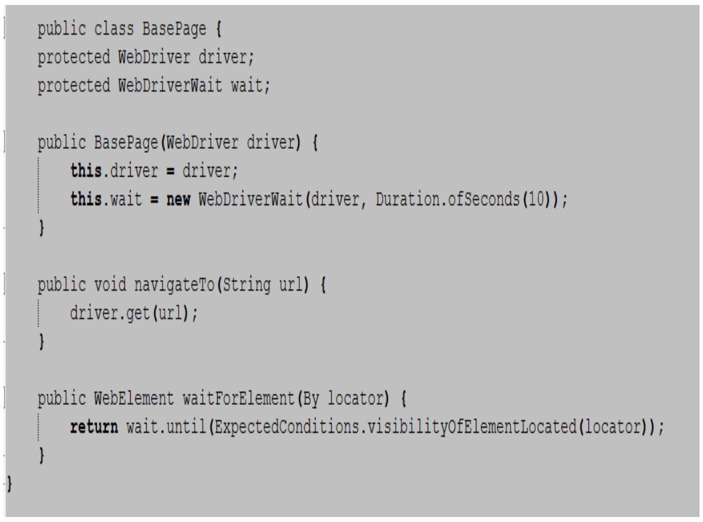
The **BasePage** class is designed to provide essential, reusable functionality for all page objects. It consists of three main components:

- **WebDriver**: This is used to interact with the browser, enabling actions like navigating to pages, clicking buttons, filling out forms, and retrieving element data.

- **WebDriverWait**: This utility helps the script to wait for specific conditions (such as element visibility) before interacting with web elements. This ensures synchronization and avoids issues where elements are not ready for interaction.

- **Base Methods**: Methods like navigateTo and waitForElement simplify common tasks across multiple page objects. navigateTo is responsible for navigating to a given URL, while waitForElement ensures that an element is visible before interacting with it.

The constructor initializes these components, ensuring they are available for any page class that extends **BasePage**. By centralizing repetitive actions such as navigation and element waiting in this base class, you reduce code duplication and ensure a consistent approach to interacting with web elements, ultimately streamlining test creation and maintenance [3].



```
public class BasePage {
protected WebDriver driver;
protected WebDriverWait wait;

public BasePage(WebDriver driver) {
    this.driver = driver;
    this.wait = new WebDriverWait(driver, Duration.ofSeconds(10));
}

public void navigateTo(String url) {
    driver.get(url);
}

public WebElement waitForElement(By locator) {
    return wait.until(ExpectedConditions.visibilityOfElementLocated(locator));
}
}
```

Figure 1: Example BasePage Implementation

Abstracting Page Object Creation with a Factory Method

A factory method is a design pattern that provides a streamlined way to dynamically create page objects based on specific contexts such as browser type, platform, or device. This pattern ensures flexibility and scalability in test automation by abstracting page object creation, enabling the framework to adapt seamlessly to varying environments [4].

Benefits of the Factory Method

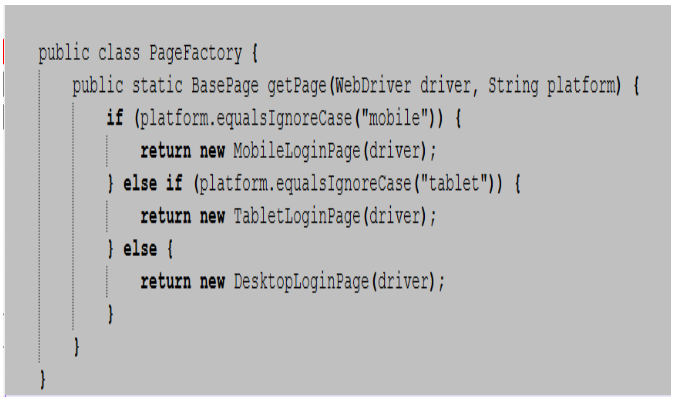
- **Separation of Concerns**: Test scripts are simplified as the logic for creating page objects is moved to a centralized factory class.

- **Extensibility**: Supporting new platforms or browsers only requires modifying the factory logic, without altering existing tests.

- **Dynamic Instantiation**: Page objects are created dynamically at runtime based on configuration settings like platform or browser type, promoting adaptability.

Code Implementation

The PageFactory class encapsulates the logic for instantiating platform-specific page objects. The getPage method dynamically creates the appropriate subclass of BasePage (such as MobileLoginPage, TabletLoginPage, or DesktopLoginPage) based on the provided platform parameter:



```
public class PageFactory {
    public static BasePage getPage(WebDriver driver, String platform) {
        if (platform.equalsIgnoreCase("mobile")) {
            return new MobileLoginPage(driver);
        } else if (platform.equalsIgnoreCase("tablet")) {
            return new TabletLoginPage(driver);
        } else {
            return new DesktopLoginPage(driver);
        }
    }
}
```

Figure 2: Example Factory Method Implementation

Leveraging Abstract Classes or Interfaces for Consistency

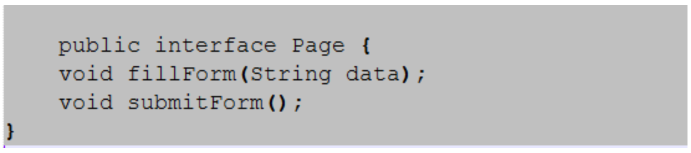
To ensure all page objects follow a consistent structure, using abstract classes or interfaces is a best practice. These constructs establish a contract that all implementing classes must adhere to, ensuring uniformity and better reusability. This approach simplifies test automation, especially when dealing with platform-specific behaviors or layouts [5].

Advantages of Using Abstract Classes or Interfaces:

- **Standardizes Method Signatures**: Guarantees all page objects implement the same set of methods, ensuring consistency across different platforms.

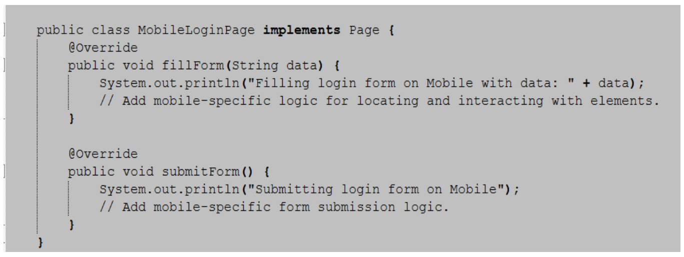
- **Encapsulates Platform-Specific Behavior**: Allows flexibility to define platform-specific implementations while maintaining a common structure for all page objects.

- **Enhances Maintainability**: Simplifies updates since shared behavior is centralized in the interface or abstract class.



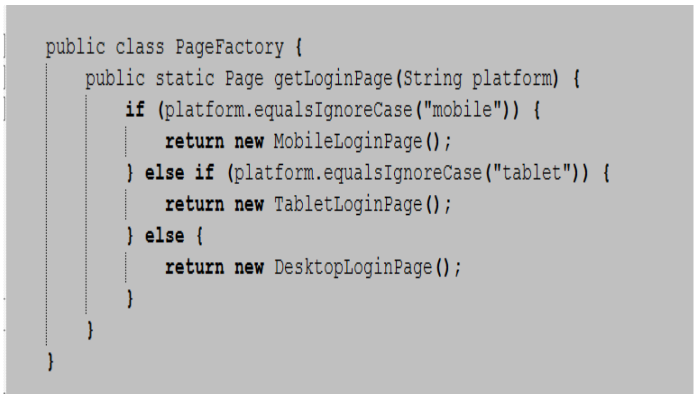
```
public interface Page {
    void fillForm(String data);
    void submitForm();
}
```

Figure 3: Page Interface



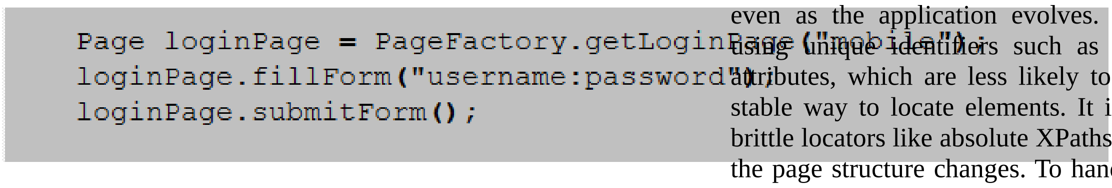
```
public class MobileLoginPage implements Page {
    @Override
    public void fillForm(String data) {
        System.out.println("Filling login form on Mobile with data: " + data);
        // Add mobile-specific logic for locating and interacting with elements.
    }

    @Override
    public void submitForm() {
        System.out.println("Submitting login form on Mobile");
        // Add mobile-specific form submission logic.
    }
}
```

Figure 4: Platform specific Page Implementation



```
public class PageFactory {
    public static Page getLoginPage(String platform) {
        if (platform.equalsIgnoreCase("mobile")) {
            return new MobileLoginPage();
        } else if (platform.equalsIgnoreCase("tablet")) {
            return new TabletLoginPage();
        } else {
            return new DesktopLoginPage();
        }
    }
}
```

Figure 5:Factory Method Implementation



```
Page loginPage = PageFactory.getLoginPage("mobile");
loginPage.fillForm("username:password");
loginPage.submitForm();
```

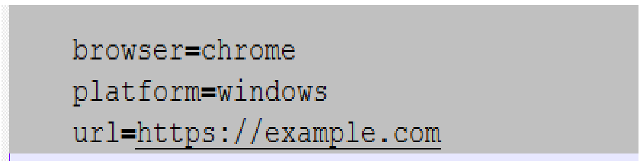
Figure 6:Testcase Example

Using Configuration Files for Dynamic Test Environments

A configuration file centralizes environment-specific settings, such as browser types, platform names, and application URLs, making the test framework highly adaptable. By externalizing these details, you can easily switch between environments like staging, production, or testing without modifying the underlying codebase. This approach promotes maintainability and eliminates hardcoding of sensitive information [6].

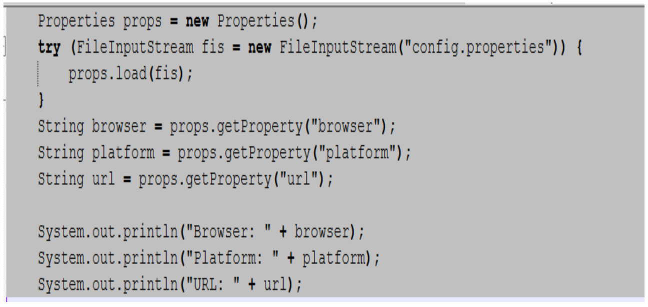
Advantages of Using Configuration Files:

- **Avoids Hardcoding**: URLs, browsers, and other environment-specific details are externalized for flexibility.
- **Streamlined Environment Switching**: Makes switching between test environments seamless.
- **Enhances Maintainability**: Updates to the environment require changes only in the configuration file, not in the code.



```
browser=chrome
platform=windows
url=https://example.com
```

Figure **7:** Configuration File

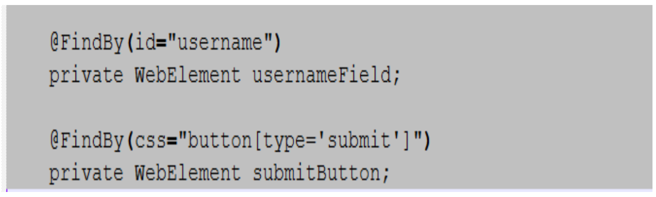


```
Properties props = new Properties();
try (FileInputStream fis = new FileInputStream("config.properties")) {
    props.load(fis);
}
String browser = props.getProperty("browser");
String platform = props.getProperty("platform");
String url = props.getProperty("url");

System.out.println("Browser: " + browser);
System.out.println("Platform: " + platform);
System.out.println("URL: " + url);
```

Figure **8:** Load Configuration

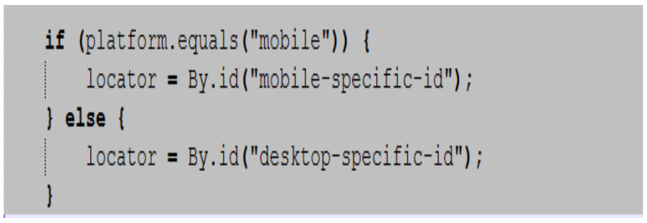
Robust Locator Strategies for Cross-Browser Testing

A robust locator strategy is essential for ensuring the reliability of tests, especially when working across multiple browsers and platforms. By choosing the right locators, your test automation code can remain adaptable even as the application evolves. Best practices include using unique identifiers such as id, name, or data-test attributes, which are less likely to change and provide a stable way to locate elements. It is also critical to avoid brittle locators like absolute XPaths that can easily break if the page structure changes. To handle elements that differ based on the platform (mobile, tablet, desktop), dynamic locators can be employed using conditional logic [7].



```
@FindBy(id="username")
private WebElement usernameField;

@FindBy(css="button[type='submit']")
private WebElement submitButton;
```

Figure 9: Robust Locator Strategy

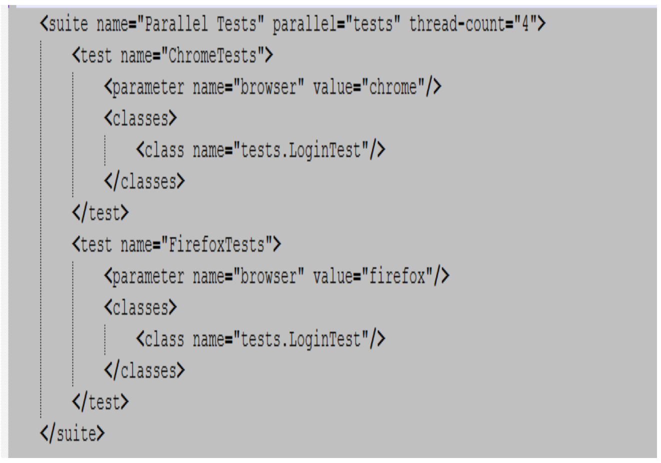


```
if (platform.equals("mobile")) {
    locator = By.id("mobile-specific-id");
} else {
    locator = By.id("desktop-specific-id");
}
```

Figure 10: Multiplatform dynamic locator

Parallel Test Execution for Optimizing Test Time

Parallel test execution is a powerful strategy for optimizing test time, enabling multiple tests to run concurrently across different platforms and browsers. This approach significantly reduces overall execution time, particularly when covering various devices, operating systems, and browser combinations. By implementing parallel execution, your test suite can run tests simultaneously, enhancing efficiency and accelerating feedback. The key benefits include faster feedback with quicker bug detection, optimized resource utilization, and scalability to handle more complex tests as your system evolves [8].



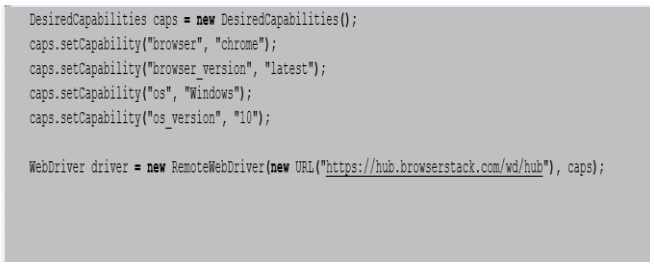
```xml
<suite name="Parallel Tests" parallel="tests" thread-count="4">
    <test name="ChromeTests">
        <parameter name="browser" value="chrome"/>
        <classes>
            <class name="tests.LoginTest"/>
        </classes>
    </test>
    <test name="FirefoxTests">
        <parameter name="browser" value="firefox"/>
        <classes>
            <class name="tests.LoginTest"/>
        </classes>
    </test>
</suite>
```

Figure 11: Parallel Execution using TestNG

In this configuration, TestNG runs tests for Chrome and Firefox browsers in parallel, utilizing multiple threads (in this case, up to 4 threads). This setup allows you to efficiently test across different environments and get faster results while optimizing your testing resources.

Using Cross-Browser Testing Tools for Multiple Environments

Cross-browser testing tools like Selenium Grid, BrowserStack, and Sauce Labs are essential for ensuring compatibility across different platforms, browsers, and devices. These tools allow for parallel test execution on real browsers and devices, without requiring the maintenance of physical infrastructure. They help streamline the testing process by providing access to multiple browsers, operating systems, and devices from a centralized, cloud-based platform.



```java
DesiredCapabilities caps = new DesiredCapabilities();
caps.setCapability("browser", "chrome");
caps.setCapability("browser_version", "latest");
caps.setCapability("os", "Windows");
caps.setCapability("os_version", "10");

WebDriver driver = new RemoteWebDriver(new URL("https://hub.browserstack.com/wd/hub"), caps);
```

Figure 12: Remote Webdriver Instance with Browserstack

In this code, **DesiredCapabilities** are used to define the browser, browser version, and operating system version. A **RemoteWebDriver** is initialized with these capabilities, which connects to BrowserStack's cloud server to run the test on a remote browser instance. This setup ensures that tests are executed across different platforms without the need to configure physical machines or environments, making the testing process more efficient and scalable.

Conclusion

- **Page Object Model (POM) Design**: A robust POM ensures maintainable, scalable, and flexible test automation code across multiple browsers and platforms.
- **Base Page Class**: Centralizes shared functionality, reducing redundancy and improving maintainability.
- **Factory Method**: Dynamically creates page objects based on the environment, promoting flexibility and scalability.
- **Abstract Classes or Interfaces**: Enforce consistency and structure across all page objects, making the framework extensible and easy to maintain.
- **Configuration Files**: Externalize environment-specific settings, simplifying environment switching and improving test maintainability.
- **Locator Strategies**: Use reliable and platform-independent locators, ensuring consistent element identification across different browsers and platforms.
- **Parallel Test Execution**: Reduces test execution time by running tests concurrently, enabling faster feedback and better resource optimization.
- **Cloud-based Cross-Browser Testing Tools**: Leverage tools like Selenium Grid, BrowserStack, and Sauce Labs to access a wide range of browsers and devices without needing local infrastructure.
- **Scalable Framework**: The strategies outlined allow for the seamless addition of new browsers, devices, or platforms, ensuring long-term sustainability as your application grows.
- **Overall Goal**: By implementing these practices, testing teams can build adaptable frameworks that provide high-quality, reliable test results across diverse environments, ultimately ensuring the application's robustness and performance.

REFERENCES

[1] https://saucelabs.com/resources/blog/getting-started-with-cross-platform-testing

[2] https://www.softwaretestinghelp.com/testng-example-to-create-testng-xml/
[3] https://www.toolsqa.com/selenium-webdriver/page-object-model/
[4] https://www.guru99.com/testng-execute-multiple-test-suites.html
[5] https://www.softwaretestinghelp.com/automate-web-app-on-chrome-browser/
[6] https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/
[7] https://www.softwaretestingmaterial.com/page-object-model/#Page-Object-Model-Design-Pattern
[8] https://www.browserstack.com/guide/design-patterns-in-selenium