SSRG International Journal of Computer Science and Engineering
ISSN: 2348–8387 / https://doi.org/10.14445/23488387/IJCSE-V11I8P106

Volume 11 Issue 8, 45-49, August 2024

Original Article

Page Object Design Pattern for IOS UI Automation

Naveen Chikkanayakanahalli Ramachandrappa

Lead Software Quality Assurance Engineer, Texas, USA.

Corresponding Author : accessnaveen@gmail.com

Received: 24 June 2024 Revised: 31 July 2024 Accepted: 20 August 2024 Published: 31 August 2024

Abstract *- The Page Object Model (POM) design pattern is a robust and widely adopted approach in UI automation that enhances test maintenance and reduces code duplication. This research paper explores the application of the POM design pattern for iOS UI automation, examining its benefits, implementation strategies, and best practices. By integrating detailed examples and referencing relevant literature, this paper aims to provide a comprehensive guide for developers and testers seeking to leverage POM for efficient and robust iOS UI automation.*

Keywords *- Page Object Model (POM), Test abstraction, UI encapsulation, Automated testing frameworks, Code reusability.*

1. Introduction

UI automation plays a critical role in software testing by verifying that applications function correctly from the end-user's perspective. However, as iOS applications grow increasingly complex, the maintenance of automated UI tests presents significant challenges. These challenges include the difficulties of managing test scripts as the application's user interface evolves and ensuring that tests remain reliable amidst frequent updates. The Problem of maintaining automated UI tests in iOS applications stems from the need to adapt to constant platform updates and shifting design guidelines.

Traditional testing frameworks often struggle with the rapid pace of changes, leading to brittle tests and increased maintenance efforts. This issue highlights a significant gap in the testing process: the lack of an effective method to decouple test scripts from the UI components they interact with, which affects both readability and maintainability. The Page Object Model (POM) design pattern addresses this gap by introducing a structured approach to representing UI elements and interactions. By separating the test logic from the UI details, POM enhances the scalability and manageability of test automation frameworks. This paper will explore the implementation of the POM design pattern within the iOS platform, examining its effectiveness in overcoming the challenges associated with UI automation and improving testing efficiency.

2. Background

The concept of POM originated in the context of web testing but has since been adapted for mobile platforms, including iOS. POM involves creating an abstraction layer over the UI, where a corresponding class represents each page or screen of the application. These classes encapsulate the elements and actions that can be performed on the respective pages, allowing test scripts to interact with the pages through these objects [1].

3. Benefits of POM in iOS UI Automation

3.1. Improved test maintenance

POM reduces code duplication by centralizing UI elements and actions. Changes to the UI need only be updated in one place, the page object, rather than across multiple test scripts. This significantly lowers the maintenance overhead and ensures consistency across tests [1]. For instance, if a button's identifier changes, the update is required only in the page object class, leaving the test scripts unaffected.

3.2. Enhanced Readability and Reusability

By abstracting the UI interactions into page objects, test scripts become more readable and reusable. Testers can write tests in a high-level language that focuses on business logic rather than UI intricacies [2]. This abstraction allows test scripts to be written in a more natural language, making them accessible to non-technical stakeholders and improving collaboration between developers and testers.

3.3. Separation of Concerns

POM promotes the separation of test logic from UI structure, making the tests more modular and easier to manage [2]. This separation ensures that changes in the UI structure do not necessitate changes in the test logic, thus making the test suite more resilient to UI changes.

4. Implementation of POM for IOS

Implementation of POM for iOS involves several key steps, from setting up the project to writing test scripts that utilize the page objects.

4.1. Setting up the project

To start, create an iOS UI automation project using a testing framework such as XCTest. XCTest is integrated into Xcode and provides a robust environment for writing and running UI tests.

4.2. Creating Page Objects

Each page object should represent a distinct screen or a significant component of the app. For instance, a login screen would have a LoginPage class encapsulating the username and password fields, and the login button. The following example demonstrates a basic LoginPage class, encapsulating the elements and actions related to the login process [1]. The login method abstracts the sequence of actions required to perform a login, simplifying the test scripts that interact with this page.

4.3. Writing Test Scripts

Test scripts can now interact with the page objects, making the tests more readable and maintainable. The following example test script focuses on the business logic, using the LoginPage class to perform the login action [4]. The abstraction provided by the page object allows the test to be concise and focused on the expected outcomes rather than the details of the UI interactions.

```
class LoginPage {
    let app = XCUIApplication()

    var usernameField: XCUIElement {
        return app.textFields["username"]
    }

    var passwordField: XCUIElement {
        return app.secureTextFields["password"]
    }

    var loginButton: XCUIElement {
        return app.buttons["login"]
    }

    func login(username: String, password: String) {
        usernameField.tap()
        usernameField.typeText(username)

        passwordField.tap()
        passwordField.typeText(password)

        loginButton.tap()
    }
}
```

Fig. 1

```
func testLogin() {
    let loginPage = LoginPage()

    loginPage.login(username: "testuser",
                    password: "password123")

}
```

Fig. 2

5. Best practices for POM in IOS

5.1. Consistent Naming Conventions

Use clear and consistent naming conventions for page objects and UI elements to enhance readability and maintainability [1]. Consistency in naming helps in understanding the role and functionality of each page object and its components, facilitating easier navigation and updates.

5.2. Encapsulation of Actions

Encapsulate all possible actions on a page within the corresponding page object, reducing the need to manipulate UI elements directly in test scripts [2]. This encapsulation ensures that the test scripts remain focused on the test logic while the details of the UI interactions are managed within the page objects.

5.3. Utilize Extensions

Swift extensions are used to add functionalities to existing classes, improving code modularity and reuse [3]. Extensions can be employed to add common functionalities across multiple-page objects, reducing redundancy and enhancing maintainability.

5.4. Error Handling

Implement robust error handling within page objects to manage unexpected UI changes gracefully [2]. By anticipating potential issues and incorporating error-handling mechanisms, the robustness and reliability of the test suite can be significantly enhanced.

6. Advanced Techniques in POM for IOS

6.1. Lazy Initialization

Utilize lazy initialization for UI elements within page objects to improve performance and avoid unnecessary element lookups [3]. Lazy initialization ensures that UI elements are only accessed when needed, reducing the overhead during test execution.

```
class LoginPage {
    let app = XCUIApplication()

    lazy var usernameField: XCUIElement = {
        return app.textFields["username"]
    }()

    lazy var passwordField: XCUIElement = {
        return app.secureTextFields["password"]
    }()

    lazy var loginButton: XCUIElement = {
        return app.buttons["login"]
    }()

    func login(username: String, password: String) {
        usernameField.tap()
        usernameField.typeText(username)
        passwordField.tap()
        passwordField.typeText(password)
        loginButton.tap()
    }
}
```

Fig. 3

6.2. Custom Assertions

Create custom assertions within page objects to validate UI states, improving the clarity and expressiveness of test scripts [2]. Custom assertions encapsulate the verification logic within the page objects, making the test scripts more readable.

```
class LoginPage {
    let app = XCUIApplication()

    var usernameField: XCUIElement {
        return app.textFields["username"]
    }

    var passwordField: XCUIElement {
        return app.secureTextFields["password"]
    }

    var loginButton: XCUIElement {
        return app.buttons["login"]
    }

    var errorMessageLabel: XCUIElement {
        return app.staticTexts["errorMessage"]
    }

    func login(username: String, password: String) {
        usernameField.tap()
        usernameField.typeText(username)
        passwordField.tap()
        passwordField.typeText(password)
        loginButton.tap()
    }

    func assertErrorMessage(_ message: String) {
        XCTAssertEqual(errorMessageLabel.label, message)
    }
}
```

Fig. 4

6.3. Page Object Inheritance

Use inheritance to create base page objects with common functionalities, promoting code reuse and reducing duplication [4]. Base page objects can define shared actions and elements, which specific page objects can inherit.

```
class BasePage {
    let app = XCUIApplication()
    var backButton: XCUIElement {
        return app.buttons["back"]
    }

    func goBack() {
        backButton.tap()
    }
}

class LoginPage: BasePage {
    //Login page methods and properties
}
```

Fig. 5

7. Case Study: POM in a Real-World iOS Application

To illustrate the effectiveness of POM, consider a case study of a banking application. The application includes various screens for login, account summary, and fund transfer. By implementing POM, the test automation team created separate page objects for each screen, significantly reducing test script complexity and improving maintainability.

7.1. LoginPage

The LoginPage class encapsulates the elements and actions related to the login process, providing a clear and maintainable interface for the test scripts [1].

```
class LoginPage {
    let app = XCUIApplication()

    var usernameField: XCUIElement {
        return app.textFields["username"]
    }

    var passwordField: XCUIElement {
        return app.secureTextFields["password"]
    }

    var loginButton: XCUIElement {
        return app.buttons["login"]
    }

    func login(username: String, password: String) {
        usernameField.tap()
        usernameField.typeText(username)
        passwordField.tap()
        passwordField.typeText(password)
        loginButton.tap()
    }
}
```

Fig. 6

7.2. AccountSummaryPage

The AccountSummaryPage class provides methods to interact with the account summary screen, abstracting the UI interactions required to retrieve the account balance [1].

```
class AccountSummaryPage {
    let app = XCUIApplication()

    var accountBalanceLabel: XCUIElement {
        return app.staticTexts["accountBalance"]
    }

    func getAccountBalance() -> String {
        return accountBalanceLabel.label
    }
}
```

Fig. 7

7.3. FundTransferPage

The FundTransferPage class encapsulates the elements and actions required for fund transfer, ensuring that the test scripts remain focused on the business logic [2].

```
class FundTransferPage {
    let app = XCUIApplication()

    var fromAccountField: XCUIElement {
        return app.textFields["fromAccount"]
    }

    var toAccountField: XCUIElement {
        return app.textFields["toAccount"]
    }

    var amountField: XCUIElement {
        return app.textFields["amount"]
    }

    var transferButton: XCUIElement {
        return app.buttons["transfer"]
    }

    func transferFunds(from: String, to: String, amount: String) {
        fromAccountField.tap()
        fromAccountField.typeText(from)
        toAccountField.tap()
        toAccountField.typeText(to)
        amountField.tap()
        amountField.typeText(amount)
        transferButton.tap()
    }
}
```

Fig. 8

By using POM, the banking app's UI tests were streamlined, and the maintenance overhead was significantly reduced. Testers could quickly update page objects in response to UI changes without modifying the test scripts.

8. Tools and Frameworks Supporting POM in iOS

Several tools and frameworks support the implementation of POM in iOS UI automation, including:

8.1. XCTest

Apple's native testing framework, integrated into Xcode, provides comprehensive support for UI testing with POM [3]. XCTest offers a seamless experience for iOS developers, allowing them to write and run tests directly within the Xcode environment.

8.2. Appium

An open-source tool that supports iOS and Android automation, allowing the use of POM with various languages and frameworks [2]. Appium provides flexibility in writing tests using languages such as Java, Python, and JavaScript, making it a versatile choice for cross-platform testing.

8.3. EarlGrey

Developed by Google, EarlGrey is another powerful tool for iOS UI testing, supporting POM implementation with Swift or Objective-C [2]. EarlGrey integrates well with XCTest, providing additional functionalities such as synchronization and improved element interaction capabilities.

9. Challenges and Limitations

While POM offers numerous advantages, it also presents some challenges.

9.1. Initial Setup Overhead

Setting up page objects requires an initial investment of time and effort, which can be significant for large applications. The creation of page objects and the encapsulation of actions need careful planning and execution, which might be daunting for large and complex applications.

9.2. Learning Curve

Testers need to be familiar with the POM design pattern and the specific tools and frameworks used, which may require additional training [11]. The transition from traditional testing approaches to POM might involve a learning curve, especially for teams new to the concept.

9.3. Maintenance of Page Objects

As the application evolves, page objects need to be maintained and updated to reflect UI changes, which can become a continuous effort [2]. The dynamic nature of UI design and frequent updates necessitate constant vigilance and updates to the page objects, which might be resource-intensive.

10. Research Result

The research on employing the Page Object Model (POM) design pattern for iOS UI automation highlights its significant advantages in optimizing test efficiency and maintainability. By decoupling UI elements from test logic, POM not only simplifies complex test scenarios but also reduces maintenance overhead. This study outperforms previous research by demonstrating how POM effectively addresses challenges unique to iOS environments, such as frequent updates and evolving design guidelines. Unlike earlier approaches that often struggled with adaptability and scalability, this research shows that POM enhances readability and makes it easier to manage and update tests as the UI changes. By providing a structured framework that accommodates the dynamic nature of iOS development, this approach results in a more robust, scalable, and efficient automation process. The improved results in test efficiency and flexibility underscore POM's superior ability to handle the complexities of modern iOS applications compared to traditional methodologies.

11. Conclusion

The Page Object Model design pattern is a valuable approach for iOS UI automation, offering improved test maintenance, readability, and separation of concerns. By encapsulating UI elements and actions within page objects, testers can create modular and maintainable test scripts. As tools and frameworks continue to evolve, the application of POM is expected to become even more efficient and widespread. The implementation of POM in real-world applications, as illustrated by the banking app case study, demonstrates its potential to reduce complexity and enhance test maintainability. Future advancements in AI and machine learning are likely to augment the POM design pattern further, making it an even more powerful tool in the arsenal of iOS testers.

References

[1] Lisa Crispin, and Janet Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*, Addison-Wesley, 2009. [Google Scholar] [Publisher Link]

[2] Jerry Gao et al., "Mobile Application Testing: A Tutorial," *IEEE Software*, vol. 47, no. 2, pp. 46-55, 2014. [CrossRef] **[**Google Scholar**]** **[**Publisher Link**]**

[3] Apple Inc., Xcode, Apple Developer Documentation, 2020. [Online]. Available: https://developer.apple.com/documentation/xctest

[4] Andrew Hunt, and David, *The Pragmatic Programmer*, Addison-Wesley, 1999. **[**Google Scholar**]** **[**Publisher Link**]**