Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2018, 5(12):1034-1042

Research Article ISSN: 2394 - 658X

Balancing Complexity and Efficiency Strategies for Multi-Application Test Automation Using Shared Code Frameworks

Amit Gupta

Software Engineer/Leader, San Jose, CA, USA
gupta25@gmail.com

ABSTRACT
The need for reliable and effective testing processes has grown rapidly due to the dynamic growth of software development. Organizations may now deliver products more quickly while upholding high standards to test automation, which has become an essential tool for ensuring software quality, stability, and scalability. Ensuring the quality of each code swift and smooth integration, however, offers special issues in multi-application systems, where multiple software entities interact and share data. To facilitate test automation in such systems, this study intends to investigate and clarify the approaches for creating and putting into practice a shared code framework.
The first part of the paper explains the balance between efficiency and complexity that arises when testing several applications that communicate with complex architecture (On-prem and cloud). It looks into the trade-offs between test automation solutions effectiveness in producing trustworthy results and their level of complexity. This paper explores the design concepts of a modular and flexible shared code framework and discusses the integration of application-specific modules within the common framework. It identifies fundamental functionality modules that enable cross-application test automation. Various approaches are suggested to strike a balance between efficiency and complexity. These include classifying test suites according to priority, abstracting and reusing test cases, and parallelizing and distributing tests to mitigate execution timeframes. The implementation process and case study illustrate the advantages and effectiveness of the shared code framework. The study examines the advantages of using a shared code framework, such as enhanced maintainability and reusability, and discusses possible drawbacks like the customization complexity and learning curve. The benefits of the shared code architecture in terms of code reuse, uniform testing, and maintainability are highlighted by a comparison with current methods.

The connection with CI/CD pipelines, AI-driven testing, cross-domain application testing, and utilizing microservices and containerization are some of the future approaches for multi-application test automation. The shared code architecture for multi-application test automation offers a flexible way to negotiate the complexities of contemporary software ecosystems, enabling businesses to produce high-caliber apps quickly and effectively.

Key words: Test Automation, Scalability, Abstraction, Software Testing, Mobile Testing, Software Quality

INTRODUCTION

In the realm of software development, applications have become integral components of various industries mapped with horizontal and vertical service lines, ranging from finance and healthcare to entertainment and communication. As the demand for robust and feature-rich software applications increases, so does the need for robust and efficient testing procedures. Test automation has emerged as a crucial solution to ensure software Quality, reliability and scalability enable organizations to deliver products faster while maintaining high performance and security standards. However, the landscape of software applications is evolving rapidly, with applications becoming more interconnected and complex. In multi-application environments, where multiple software entities interact and share data, ensuring the quality of each application and the seamless integration presents unique challenges.

OBJECTIVE

The world is progressing toward automation, resulting in an escalating demand for test automation. Test automation frameworks enable concurrent testing on various mobile devices with simulating devices.
This paper aims to explore and elucidate the strategies for designing and implementing a shared code framework for test automation in multi-application environments.

Understanding Complexity and Efficiency Balance:

- Analyzing the inherent complexities of testing multiple applications that interact and collaborate.
- They are investigating the trade-offs between the complexity of test automation solutions and their efficiency in delivering reliable results.

Shared Code Framework Design:

- Exploring the design principles of a modular and adaptable shared code framework.
- Identifying the core functionality modules that facilitate cross-application test automation.
- Discussing the integration of application-specific modules within the shared framework.

Strategies for Balancing Complexity and Efficiency:

- Proposing strategies to abstract and reuse test cases across diverse applications.
- Examining methods for prioritizing test suites based on critical functionalities and integration points.
- Investigating techniques for parallelizing and distributing tests to optimize execution time.

Implementation and Case Study:

- Detailing the step-by-step process of developing and implementing the shared code framework.
- Presenting a real-world case study where the framework is applied to multiple applications, showcasing its efficacy and benefits.

Benefits, Challenges, and Future Directions:

- Evaluating the benefits of adopting a shared code framework, including improved reusability and maintainability.
- Addressing the potential challenges and learning curves of implementing such a framework.
- Discussing potential avenues for future research and development, such as integrating the framework into CI/CD pipelines and incorporating AI-driven automation.

Through this exploration, the paper seeks to provide insights into creating a versatile and adaptable test automation framework that balances the complexity inherent in multi-application testing environments and the efficiency required for adequate software quality assurance.

CHALLENGES

The transition from testing individual applications to testing multiple applications within a complex ecosystem introduces challenges that traditional test automation approaches may struggle to address such as version control, testing multiple layers of software, security threads, etc. These challenges arise from the intricacies of interactions, diverse application architectures, and the potential maintenance overhead. Understanding and mitigating these challenges are crucial for designing a successful shared code framework for multi-application test automation.

Complexity of Interactions: In multi-application environments, applications often communicate, share data, and interact in subtle ways. The testing process must simulate these interactions accurately to ensure the applications function harmoniously. However, accurately modeling and testing these interactions can lead to increased complexity. Test scenarios need to cover various possible interaction paths, increasing the number of test cases and complicating test orchestration.

Diverse Application Architectures: Different applications within a multi-application ecosystem can have diverse architectures, technologies, and coding standards. This diversity poses challenges for creating a uniform testing framework that seamlessly accommodates these variations. This framework must be flexible enough to accommodate different communication protocols, data formats, and authentication mechanisms while maintaining a consistent testing approach.

Maintenance Overhead: As the number of applications in the ecosystem grows, maintaining individual test automation scripts for each application can become unwieldy and resource-intensive. Any changes in the applications functionalities or interfaces can lead to cascading updates across multiple test scripts. Ensuring that the test automation remains up-to-date and synchronized with the evolving applications can be a significant challenge, often resulting in increased maintenance overhead.

Test Data Management: Multi-application environments often involve the use of shared databases or data repositories. Coordinating and managing the test data across applications becomes critical to ensure accurate testing. Effective handling of data consistency, privacy, and integrity is necessary to keep data-related problems from interfering with the testing procedure.
Addressing these challenges requires a comprehensive approach that combines architectural design, testing strategies, and efficient tools to create a shared code framework that can effectively manage the complexities of multi-application test automation while optimizing efficiency.

CURRENT APPROACH

Comparing the shared code framework for multi-application test automation with existing approaches provides valuable insights into the advantages and unique features of the proposed solution. This section evaluates the framework against two common existing approaches:

Proprietary Test Automation Tools

Advantages	Disadvantages
User-friendly Interfaces: Proprietary tools often have user-friendly interfaces, making them accessible to testers with varying technical expertise. Built-in Features: These tools offer built-in features for test case management, test execution, and reporting. Vendor Support: Vendors provide support, updates, and assistance for using their tools effectively. Ease of Setup: Many proprietary tools are designed to be easily set up and integrated into existing workflows.	Vendor Lock-in: Organizations may depend on a specific vendor's tools, limiting flexibility. Cost: Proprietary tools can be expensive, especially as the number of applications and users grows. Limited Customization: Customizing these tools To meet specific multi-application requirements can be challenging.

Application-Specific Test Frameworks

Advantages	Disadvantages
Tailored Approach: Application-specific frameworks are designed with the specific needs of a particular application in mind, ensuring a precise fit. **Focused Testing**: These frameworks can intensely focus on the application's architecture and functionality nuances. **Optimized Performance:** Application-specific frameworks can be optimized for performance and specific use cases.	**Reusability Challenges:** Reusing test cases across different applications can take time, leading to redundant effort. **Fragmented Maintenance:** Each application might have its own framework, leading to maintenance fragmentation. **Learning Curve:** Teams need to learn and adapt to different frameworks for each application.

Comparative Considerations

- Organizations seeking tailored solutions for specific applications might find application-specific frameworks appealing.

- Proprietary tools can be advantageous for smaller-scale testing efforts or when user-friendly interfaces are a priority.
- For complex multi-application environments with the need for efficiency, consistency, and code reusability, the shared code framework presents a strong case.

In conclusion, while each approach has merits, the shared code framework balances the advantages of application-specific frameworks and the flexibility of proprietary tools. Its ability to cater to multi-application testing needs, promote code reusability, and streamlined maintenance efforts make it a robust solution for organizations facing the challenges of testing multiple applications within a complex ecosystem.

Proposed Framework

A well-designed shared code framework forms an efficient multi-application test automation. This section outlines the principles and components of proposed framework that enables seamless testing of multiple applications while promoting code reusability, maintainability, and adaptability.

Modular Architecture: It is advised that a modular design with numerous functional layers be used to build the shared code framework. These modules contain integrated reporting tools, communication interfaces, application-specific features, and fundamental functionality. A modular approach promotes scalability because new applications can be easily integrated by adding or extending modules.

Core Functionality Modules: Core modules provide the foundational building blocks for test automation across applications. These modules often encompass functionalities like test case management, test data provisioning, and interaction with external tools and libraries. A robust core module ensures consistent execution of tests and provides a unified interface for interacting with different application-specific modules.

Application Specific Modules: Application specific modules cater to the unique requirements of individual applications. These modules handle tasks such as interacting with application interfaces, simulating user interactions and verifying application-specific behaviors. By abstracting application-specific logic into modules, the framework enhances code reusability and simplifies test case development for each application.

Test Configuration and Parameters: A flexible configuration mechanism is essential to adapt the framework to varying application environments. Parameters such as environment URLs, authentication credentials, single sign-on and test data sources should be configurable externally, enabling easy setup and reconfiguration for different applications and testing stages.

Version Control and Collaboration: Version control becomes essential as the framework evolves and matures. Managing the framework's codebase using version control systems enables collaboration among team members, facilitates code reviews, and ensures a well-maintained and documented codebase based on version change.

By adopting a modular, adaptable, and extensible framework design, organizations can create a robust foundation for multi-application test automation. This design approach balances complexity and efficiency by providing a structured environment that manages the intricacies of multi-application interactions while promoting streamlined test case development and execution.

Illustration: Consider the mobile applications automation testing as an example. This architecture and design, however, can be used with any software test automation framework that calls for multiple application test automation.

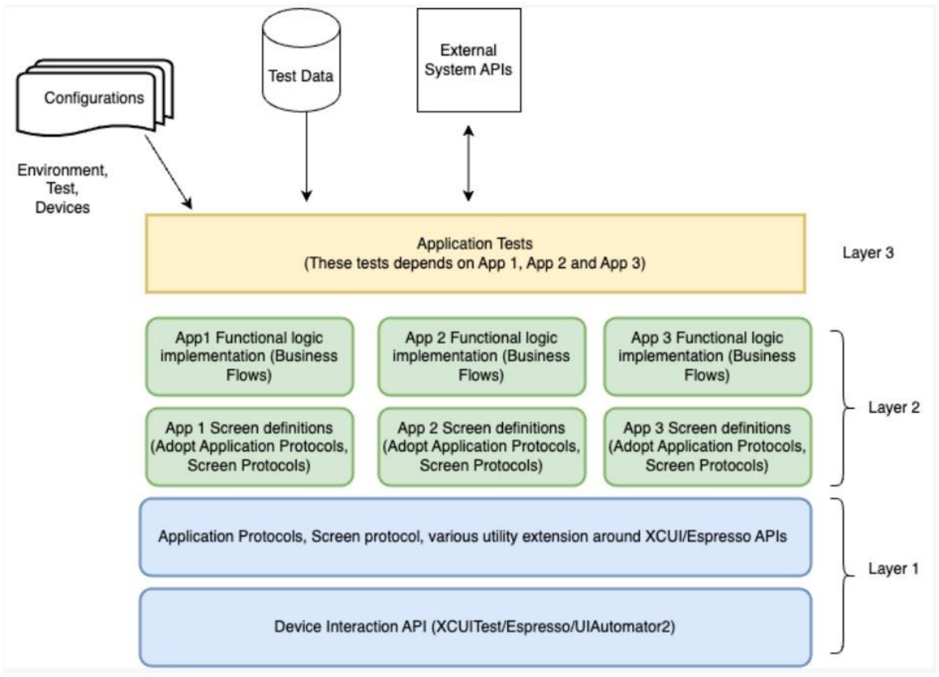
Configurations
Test Data
External System APIs
Environment, Test, Devices
Application Tests (These tests depends on App 1, App 2 and App 3)
Layer 3
App1 Functional logic implementation (Business Flows)
App 2 Functional logic implementation (Business Flows)
App 3 Functional logic implementation (Business Flows)
App 1 Screen definitions (Adopt Application Protocols, Screen Protocols)
App 2 Screen definitions (Adopt Application Protocols, Screen Protocols)
App 3 Screen definitions (Adopt Application Protocols, Screen Protocols)
Layer 2
Application Protocols, Screen protocol, various utility extension around XCUI/Espresso APIs
Device Interaction API (XCUITest/Espresso/UIAutomator2)
Layer 1

Fig. 1 Proposed design for Automation

Layer 1: Device Interaction APIs (Wrapper around XCUITest and UIAutomator2)
At the framework's foundation lies a set of APIs that abstract the interactions with mobile device functionalities. These APIs encapsulate tapping, scrolling, clicking, and other essential operations. For iOS, these APIs wrap around XCUITest [1], while for Android, they utilize UIAutomator2.
iOS Illustration:

```
// DeviceInteraction.swift import XCTest
class DeviceInteraction {
static    func    tap(element:    XCUIElement){
element.tap()
}

static    func    scroll (element:    XCUIElement, direction: XCUIElement.Direction){
element.swipe (direction)
}
// More interaction methods
}
```

Layer 2: Test Framework Base Classes and Common Utilities
Building upon the interaction APIs, this layer includes test framework base classes that define common utilities for mobile test automation. It abstracts shared functionalities including element identification algorithms and device navigation logic. This layer is version-controlled to align with OS versions and ensure compatibility.

```
// iOS Example
// BaseScreen.swift

import XCTest class BaseScreen {
let app = XCUIApplication() func navigateBack() {
app.navigationBars.buttons.element(boundBy: 0).tap()
}
// More common functionalities
}
```

Layer 3: Application-Specific Libraries

This layer utilizes the Device Interaction APIs (Layer 1) and Test Framework Base Classes (Layer 2) into a single, application-specific library. Each mobile application has its library version containing screen definitions, feature navigation logic, and business flows specific to that application. Layer 1 can be consumed by Layer 2 and declared as dependency via cocoapods [4]/swift package manager [5] (iOS) Or whatever dependency management system your solution uses.

```
// MyAppLibrary.swift import Layer2
class MyApp: BaseApplication {
// Define screen elements using XCUIElement func performFeatureA() {
// Implement navigation and interactions
}
// More screen definitions and business logic
}
```

Layer 4: Test Scripts and Validations

In the final layer, independent test scripts are created. These scripts leverage the application-specific library from Layer 3 to perform feature validations. They write test cases using the application's screen definitions, navigation logic, and business flows. The tests remain independent of each application's internal details.
Shared Code Framework Comparison

Advantages	Disadvantages
Code Reusability: The shared framework allows for substantial code reusability across applications, saving effort and time.	**Initial Investment:** Developing and implementing the shared framework requires an upfront investment of time and resources.
Consistent Testing: The uniform approach ensures consistent testing standards across applications.	**Customization Complexity:** Addressing unique application requirements within the framework require careful design and implementation.
Maintenance Efficiency: A single framework reduces maintenance overhead by centralizing standard functionalities.	**Learning Curve:** Teams must familiarize themselves with the framework's architecture and usage.
Adaptability: The framework can accommodate diverse application architectures and requirements.	
Cost-Effectiveness: Compared to proprietary tools, building a shared framework can offer cost savings in the long term.	

This multi-layered approach ensures a structured, scalable, and reusable test automation framework for mobile applications. By leveraging device interaction APIs, standard utilities, application-specific libraries, and independent test scripts, organizations can efficiently test their mobile applications across iOS and Android platforms while promoting code reusability, streamlined maintenance, and consistent testing practices.

Benefits and Challenges: This section outlines the advantages gained from adopting shared framework and addresses the potential hurdles organizations might encounter

Benefits:

Improved Code Reusability: The modular architecture of the framework promotes reusable test cases and modules across multiple applications. It reduces duplicate effort and accelerates test case development.

Efficient Test Maintenance: Maintenance efforts are streamlined by centralizing core functionalities and abstracting application-specific details. Updates to shared modules have a cascading effect on multiple applications, enhancing maintainability.

Consistent Testing Approach: The framework enforces consistent testing across applications, reducing discrepancies and ensuring uniform quality standards.

Enhanced Collaboration: The shared framework encourages collaboration among testers and developers, providing a unified platform for testing across teams working on different areas such as development, testing, security, devops, infrastructure and admins.

Scalability: The framework's modular design enables easy integration of new applications, allowing the organization to scale its testing efforts seamlessly.

Challenges

Initial Setup and Learning Curve: Implementing the shared framework requires the initial attempt to design the architecture and develop core modules. Testers and developers may also need time to use the framework proficiently.

Adapting to Unique Application Requirements: While the framework aims to abstract application-specific details, some applications might have unique requirements that demand additional customization based on client needs. Integration

Complexity: Integrating the framework into existing CI/CD pipelines and toolchains may require adjustments and integration efforts.

Maintenance of Framework Itself: Like any software project, the framework requires ongoing maintenance to stay current with evolving technologies, new application features, and changing testing needs.

Potential Performance: Bottlenecks: Performance bottlenecks require careful optimization in specific scenarios, especially those involving complex inter-application interactions.

Mitigation Strategies

To address the challenges, organizations can adopt the following strategies:

Comprehensive Documentation: Provide thorough documentation to facilitate the onboarding process and minimize the learning curve.

Flexibility in Framework Design: Build flexibility into the framework to accommodate unique application requirements while maintaining a consistent testing approach. Regular Maintenance: Allocate resources for ongoing maintenance and updates to align the framework with technological advancements

Gradual Implementation: Start with a subset of applications for initial implementation and gradually expand to cover the entire ecosystem, allowing for gradual adaptation and learning.

Performance Optimization: Profile and optimize the framework's performance to identify and mitigate potential bottlenecks, ensuring efficient execution of test suites.

Collaboration and Feedback: Encourage cooperation between testers, developers, and stakeholders to gather feedback on the framework's usability and effectiveness and use this feedback to refine the framework.

In conclusion, while adopting a shared code framework for multi-application test automation offers numerous benefits, organizations should be prepared to address the challenges through careful planning, continuous improvement, and a willingness to adapt the framework to evolving needs. The advantages of efficiency, maintainability, and code reusability can far outweigh the initial challenges, ultimately contributing to higher software quality and faster application delivery.

FUTURE DIRECTION

The continuous evolution of technology and software development practices presents several exciting future directions for multi-application test automation using shared code frameworks. This section explores potential avenues for extending and enhancing the framework's capabilities, addressing challenges, and leveraging emerging trends.

a. Integration with Continuous Integration

/Continuous Deployment (CI/CD) Pipelines:

Many organizations across the globe have already started integrating with continuous integration through pipelines.

- Seamless Integration: Develop tighter integration with CI/CD pipelines to enable automated testing in the software delivery process.
- Automated Deployment Verification: Automate the verification of application deployments in the multi-application ecosystem, ensuring compatibility and stability

b. AI-Driven Test Automation:

AI-powered testing is a newer form of test automation. It catches visual UI bugs that were impossible to see with manual testing.

- Test Case Generation: Utilize AI algorithms to generate test cases automatically, exploring various application scenarios and uncovering edge cases.
- Predictive Analysis: Leverage AI-powered predictive analysis to identify potential areas of failure and focus testing efforts accordingly.

c. Cross-Domain Application Testing:

Tests take responsibility for launching their processes and building application domains while often making assumptions about reference
.exe's and .dll's.

- Inter-Domain Testing: Extend the framework's capabilities to test applications that span multiple domains or industries, such as IoT, healthcare, and smart cities.
- Data Sharing and Privacy: Address data sharing and privacy challenges in cross-domain testing scenarios.

d. Microservices and Containerization:

Software architecture approach that involves breaking down an application into small, independently deployable services.

Microservices Testing: Enhance the framework to support testing applications built on microservices architectures.

- Containerized Testing: Develop mechanisms to test applications deployed in containerized environments, such as Docker and Kubernetes.

e. App Simulator

“App simulation for mobile testing" is indeed emerging as a significant future need in software testing, driven by the increasing complexity of mobile applications, the diversity of mobile devices and operating systems, and efficient and cost-effective testing solutions.

- Rapid Release Cycles: Mobile app development follows agile and continuous delivery practices, with frequent updates and releases. Traditional testing methods, such as manual testing on physical devices, may not keep pace with the rapid release cycles. App simulation enables automated testing in virtual environments, facilitating faster feedback and regression testing across iterations.
- Cross-Platform Testing: With the proliferation of hybrid and cross-platform mobile development frameworks, ensuring compatibility and consistency across different platforms is essential. App simulation facilitates cross-platform testing by emulating the behavior of various operating systems and environments, enabling testers to validate the application's functionality and user experience across multiple platforms without needing separate physical devices.

CONCLUSION

The Software development is evolving rapidly, driven by the demand for interconnected and efficient applications. In this context, multi-application test automation presents both challenges and opportunities. This paper explored the strategy for designing a shared code framework that balances the complexities of testing multiple applications with the imperative of efficiency.

This paper demonstrated that multi-application testing involves intricate interactions, diverse application architectures, and potential maintenance overhead. To address these challenges, the shared code framework design emphasizes modularity, core functionality modules, application-specific modules, and effective communication mechanisms.

The strategies for balancing complexity and efficiency underscored the significance of test case abstraction, prioritization of test suites, parallelization, and comprehensive error handling. The implementation and case study provided a practical illustration of how the shared framework is developed, integrated, and applied in real-world multi-application testing scenarios, highlighting its benefits and impact.

This paper also discussed the benefits and challenges of adopting a shared code framework, emphasizing improved code reusability, consistent testing standards, and streamlined maintenance as advantages. Moreover, it addressed potential challenges such as the learning curve and customization complexity, offering mitigation strategies.

In a comparative analysis, the shared code framework was juxtaposed against proprietary test automation tools and application-specific frameworks, showcasing its strengths in code reusability, consistent testing, and maintainability.

Lastly, the future directions outlined potential pathways for advancing multi-application test automation. In conclusion, the shared code framework for multi-application test automation represents a dynamic solution that empowers organizations to navigate the complexities of modern software ecosystems. By balancing complexity and efficiency, this framework enables organizations to deliver high-quality, intercon applications while accelerating testing processes and reducing maintenance overhead. As technology advances, embracing such frameworks will ensure software quality in an interconnected world.

With the proposed test automation framework, an organization can implement the "shift-left" concept, which emphasizes moving testing to the earliest phases of the software development life cycle. This will help in early-stage bug detection and a fix for the same.

REFERENCES

[1]. Humble, J. (2011). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.

[2]. Graham, D., Fewster, M., Copeland, L. and Addison-Wesley (2012). Experiences of test automation: case studies of software test automation. Upper Saddle River Etc.: Addison-Wesley, Cop.

[3]. Holmes, R. and Walker, R.J. (2012). Systematizing pragmatic software reuse. ACM Transactions on Software Engineering and Methodology, 21(4), pp.1–44. doi:https://doi.org/10.1145/2377656.2377657.

[4]. Pressman, R.S. and Maxim, B.R. (2014). Software engineering: a practitioner's approach. New York: Mcgraw-Hill Education.

[5]. 2014 Index IEEE Transactions on Software Engineering Vol. 40. (2015). IEEE Transactions on Software Engineering, 41(1), pp.104–112. doi:https://doi.org/10.1109/tse.2014.2382474.

[6]. António, C. (2016). Modern Software Engineering Methodologies for Mobile and Cloud Environments. IGI Global.

[7]. Rojas, J.M., Fraser, G. and Arcuri, A. (2016). Seeding strategies in search-based unit test generation. Software Testing, Verification and Reliability, 26(5), pp.366–401. doi:https://doi.org/10.1002/stvr.1601.

[8]. Codekeeper Magazine. [online] blog.codekeeper.co. Available at: https://blog.codekeeper.co/tag/third-party-depen dency-escrow.

[9]. Dependency Manager; Cocoapods. https://cocoapods.org/

[10]. Release high-quality apps, with confidence; Perfecto. https://www.perfecto.io/

[11]. Test your mobile app like your customers depend on it; SmartBear. Available at: https://smartbear.com/product/testcomplete/mobi le-testing/

[12]. XCTest; Apple Developer Documentation. Available at: https://developer.apple.com/documentation/xctes